

Initiation à la programmation python

Dr. Séna APEKE

UL/BERIIA

17/08/2021

1 Introduction

- Premier programme
- Les commentaires
- Notion de bloc d'instructions et d'indentation

2 Variables

- Définition
- Les types de variables
- Nommage
- Écriture scientifique

3 Opérations

- Opérations sur les types numériques
- Opérations sur les chaînes de caractères
- Opérations illicites
- La fonction `type()`
- Conversion de types
- Division de deux nombres entiers

4 Affichage

- La fonction `print()`
- Écriture formatée
 - Prise en main des f-strings
 - La méthode `.format()`

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre compte, ouvrez un shell puis lancez la commande (après installation de python évidemment) :

Python ou python3

La commande précédente va lancer l'interpréteur Python. Vous devriez obtenir quelque chose de ce style pour ubuntu.

**Python 3.6.9 (default, Jan 26 2021, 15:33:00) [GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.**

Les blocs

- `C : \Users \ yawa >` pour Windows,
- `Mac — de — yawa : Downloads$` pour Mac OS,
- `sena@sena — apeke :~ $` pour Linux.

représentent l'invite de commande de votre shell. Par la suite, cette invite de commande sera représentée simplement par le caractère \$, que vous soyez sous Windows, Mac ou Linux.

Le triple chevron `>>>` est l'invite de commande (prompt en anglais) de l'interpréteur Python. Ici, Python attend une commande que vous devez saisir au clavier. Tapez par exemple l'instruction : `print("Hello world!")` puis validez cette commande en appuyant sur la touche Entrée.

Python a exécuté la commande directement et a affiché le texte Hello world!. Il attend ensuite votre prochaine instruction en affichant l'invite de l'interpréteur Python (`>>>`). En résumé, voici ce qui a dû apparaître sur votre écran :

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

À ce stade, vous pouvez entrer une autre commande ou bien quitter l'interpréteur Python, soit en tapant la commande `exit()` puis en validant en appuyant sur la touche Entrée, soit en pressant simultanément les touches Ctrl et D sous Linux et Mac OS X ou Ctrl et Z puis Entrée sous Windows.

Premier programme

L'interpréteur présente vite des limites dès lors que l'on veut exécuter une suite d'instructions plus complexe. Comme tout langage informatique, on peut enregistrer ces instructions dans un fichier, que l'on appelle communément un script (ou programme) Python. Pour reprendre l'exemple précédent, ouvrez un éditeur de texte et entrez le code suivant : `print("Hello world!")` Ensuite, enregistrez votre fichier sous le nom `test.py`, puis quittez l'éditeur de texte.

Remarque : l'extension de fichier standard des scripts Python est `.py`. Pour exécuter votre script, ouvrez un shell et entrez la commande : `python test.py` Vous devriez obtenir un résultat similaire à ceci :

```
$ python test.py
```

```
Hello world !
```

Si c'est bien le cas, bravo ! Vous avez exécuté votre premier programme Python.

Les commentaires

Dans un script, tout ce qui suit le caractère `#` est ignoré par Python jusqu'à la fin de la ligne et est considéré comme un commentaire. Pour les commentaires sur plusieurs lignes, on utilise `""" """`

Voici un exemple :

`"""`

Blablabla... ligne 1

Blablabla... ligne 2

Blablabla... ligne 3

Blablabla... ligne 4

`"""`

Notion de bloc d'instructions et d'indentation

En programmation, il est courant de répéter un certain nombre de choses (avec les boucles) ou d'exécuter plusieurs instructions si une condition est vraie (avec les tests,). Par exemple, imaginons que nous souhaitions afficher chacune des bases d'une séquence d'ADN, les compter puis afficher le nombre total de bases à la fin. Nous pourrions utiliser l'algorithme présenté en pseudo-code dans la figure 1.

Pour chaque base de la séquence ATCCGACTG, nous souhaitons effectuer deux actions : d'abord afficher la base puis compter une base de plus. Pour indiquer cela, on décalera vers la droite ces deux instructions par rapport à la ligne précédente (pour chaque base [...]). Ce décalage est appelé **indentation**, et l'ensemble des lignes indentées constitue un bloc d'instructions.

Une fois qu'on aura réalisé ces deux actions sur chaque base, on pourra passer à la suite, c'est-à-dire afficher la taille de la séquence.


```
taille <- 0
séquence <- "ATCCGACTG"
pour chaque base dans séquence:
    afficher(base)
    taille <- taille + 1
afficher(taille)
```

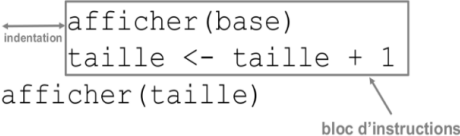


Figure 1: Notion d'indentation et de bloc d'instructions.

Pratiquement, l'indentation en Python doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une indentation avec 4 espaces est le style d'indentation recommandé.

Définition

Une variable est une zone de la mémoire de l'ordinateur dans laquelle une valeur est stockée. Aux yeux du programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

En Python, la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps. Pour vous en convaincre, testez les instructions suivantes après avoir lancé l'interpréteur :

```
>>> x = 4
```

```
>>> x
```

```
4
```

Ligne 1. Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 4. Notez bien qu'en réalité, il s'est passé plusieurs choses :

- Python a « deviné » que la variable était un entier. On dit que Python est un langage au typage dynamique
- Python a alloué (réservé) l'espace en mémoire pour y accueillir un entier. Chaque type de variable prend plus ou moins d'espace en mémoire. Python a aussi fait en sorte qu'on puisse retrouver la variable sous le nom `x`
- Enfin, Python a assigné la valeur 4 à la variable `x`

Dans d'autres langages (en C par exemple), il faut coder ces différentes étapes une par une. Python étant un langage dit de haut niveau, la simple instruction `x = 4` a suffi à réaliser les 3 étapes en une fois !

Lignes 2 et 3. L'interpréteur nous a permis de connaître le contenu de la variable juste en tapant son nom. Retenez ceci car c'est une spécificité de l'interpréteur Python, très pratique pour chasser (debugger) les erreurs dans un programme.

Les types de variables

Le type d'une variable correspond à la nature de celle-ci. Les trois principaux types dont nous aurons besoin dans un premier temps sont les entiers (integer ou int), les nombres décimaux que nous appellerons floats et les chaînes de caractères (string ou str). Bien sûr, il existe de nombreux autres types (par exemple, les booléens, les nombres complexes, etc.). Si vous n'êtes pas effrayés, vous pouvez vous en rendre compte ici. Dans l'exemple précédent, nous avons stocké un nombre entier (int) dans la variable `x`, mais il est tout à fait possible de stocker des floats, des chaînes de caractères (string ou str) ou de nombreux autres types de variable que nous verrons par la suite :

```
>>> y = 3.14
```

```
>>> y
```

```
3.14
```

```
>>> a = " bonjour"
```

```
>>> a
```

```
' bonjour '
```

```
>>> b = 'salut'
>>> b
'salut '
>>> c = """girafe"""
>>> c
'girafe '
>>> d = '''lion'''
>>> d
'lion '
```

Nommage

Le nom des variables en Python peut être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de nombres (0 à 9) ou du caractère souligné (_). Vous ne pouvez pas utiliser d'espace dans un nom de variable. Par ailleurs, un nom de variable ne doit pas débuter par un chiffre et il n'est pas recommandé de le faire débuter par le caractère _ (sauf cas très particuliers). De plus, il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par exemple : print, range, for, from, etc.). Enfin, Python est sensible à la casse, ce qui signifie que les variables Test, test ou TEST sont différentes.

Écriture scientifique

On peut écrire des nombres très grands ou très petits avec des puissances de 10 en utilisant le symbole e :

```
>>> 1e6
```

```
1000000.0
```

```
>>> 3.12e-3
```

```
0.00312
```

On appelle cela écriture ou notation scientifique. On pourra noter deux choses importantes :

- $1e^6$ ou $3.12e^{-3}$ n'implique pas l'utilisation du nombre exponentiel e mais signifie 1×10^6 ou 3.12×10^{-3} respectivement ;
- Même si on ne met que des entiers à gauche et à droite du symbole e (comme dans $1e^6$), Python génère systématiquement un float.

Enfin, vous avez sans doute constaté qu'il est parfois pénible d'écrire des nombres composés de beaucoup de chiffres, par exemple le nombre d'Avogadro $6.02214076 \times 10^{23}$ ou le nombre d'humains sur Terre (au 26 août 2020) 7807568245. Pour s'y retrouver, Python autorise l'utilisation du caractère « souligné » (ou underscore) _ pour séparer des groupes de chiffres.

Par exemple :

```
>>> avogadro_number = 6.022_140_76e23
>>> print(avogadro_number)
6.02214076e+23
```

Le caractère _ est utilisé pour séparer des groupes de 3 chiffres mais on peut faire ce qu'on veut :

```
>>> print(7_80_7568_24_5)
7807568245
```


Opérations sur les types numériques

Les quatre opérations arithmétiques de base se font de manière simple sur les types numériques (nombres entiers et floats) :

```
>>> x = 45
```

```
>>> x + 2
```

```
47
```

```
>>> x - 2
```

```
43
```

```
>>> x * 3
```

```
135
```

```
>>> y = 2.5
```

```
>>> x - y
```

```
42.5
```

```
>>> (x * 10) + y
```

```
452.5
```

Remarquez toutefois que si vous mélangez les types entiers et floats, le résultat est renvoyé comme un float (car ce type est plus général). Par ailleurs, l'utilisation de parenthèses permet de gérer les priorités.

L'opérateur / effectue une division. Contrairement aux opérateurs +, - et *, celui-ci renvoie systématiquement un float :

```
>>> 3/4
```

```
0.75
```

```
>>> 2.5/2
```

```
1.25
```

L'opérateur puissance utilise les symboles ** :

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

```
8
```

Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

```
>>> chaine = "Salut"
```

```
>>> chaine
```

```
' Salut '
```

```
>>> chaine + " Python"
```

```
' SalutPython '
```

```
>>> chaine * 3
```

```
' SalutSalutSalut '
```

L'opérateur d'addition + concatène (assemble) deux chaînes de caractères.

L'opérateur de multiplication * entre un nombre entier et une chaîne de caractères duplique (répète) plusieurs fois une chaîne de caractères

Attention

Vous observez que les opérateurs `+` et `*` se comportent différemment s'il s'agit d'entiers ou de chaînes de caractères : `2 + 2` est une addition alors que `"2" + "2"` est une concaténation. On appelle ce comportement redéfinition des opérateurs.

Opérations illicites

Attention à ne pas faire d'opération illicite car vous obtiendriez un message d'erreur :

```
>>> "toto" * 1.3
```

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

```
File "<stdin>", line 1 , in <module>
```

```
TypeError : can ' t multiply sequence by non - int of type ' float '
```

```
>>> "toto" + 2
```

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

```
File "<stdin>", line 1 , in <module>
```

```
TypeError : can only concatenate str ( not " int " ) to str
```

Notez que Python vous donne des informations dans son message d'erreur. Dans le second exemple, il indique que vous devez utiliser une variable de type str c'est-à-dire une chaîne de caractères et pas un int, c'est-à-dire un entier.

La fonction type()

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type()` qui vous le rappellera.

```
>>> x = 2
>>> type(x)
< class 'int' >
>>> y = 2.0
>>> type(y)
< class 'float' >
>>> z = ' 2'
>>> type(z)
< class 'str' >
```

Nous verrons plus tard ce que signifie le mot `class`.

Conversion de types

En programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
>>> i = '456'
```

```
>>> int(i)
```

```
456
```

```
>>> float(i)
```

```
456.0
```

On verra au chapitre 7 Fichiers que ces conversions sont essentielles. En effet, lorsqu'on lit ou écrit des nombres dans un fichier, ils sont considérés comme du texte, donc des chaînes de caractères. Toute conversion d'une variable d'un type en un autre est appelé casting en anglais, il se peut que vous croisie ce terme si vous consultez d'autres ressources.

Division de deux nombres entiers

En Python 3, la division de deux nombres entiers renvoie par défaut un float.

```
>>> x = 3/4
```

```
>>> x
```

```
0.75
```

```
>>> type(x)
```

```
< class 'float' >
```


La fonction print()

Dans la section 1, nous avons rencontré la fonction `print()` qui affiche une chaîne de caractères (le fameux "Hello world!"). En fait, la fonction `print()` affiche l'argument qu'on lui passe entre parenthèses et un retour à ligne. Ce retour à ligne supplémentaire est ajouté par défaut. Si toutefois, on ne veut pas afficher ce retour à la ligne, on peut utiliser l'argument par « mot-clé » `end` :

1. `>>> print(" Hello world !")`
2. Hello world !
3. `>>> print(" Hello world !" , end = "")`
4. Hello world ! `>>>`

Ligne 1. On a utilisé l'instruction `print()` classiquement en passant la chaîne de caractères "Hello world!" en argument. Ligne 3. On a ajouté un second argument `end=""`, en précisant le mot-clé `end`. Ligne 4. L'effet de l'argument `end=""` est que les trois chevrons `>>>` se retrouvent collés après la chaîne de caractères "Hello world!".

Une autre manière de s'en rendre compte est d'utiliser deux fonctions `print()` à la suite. Dans la portion de code suivante, le caractère « ; » sert à séparer plusieurs instructions Python sur une même ligne :

```
>>> print(" Hello"); print(" Awa")
```

Hello

Awa

```
>>> print(" Hello", end = ""); print(" Awa")
```

Hello Awa

La fonction `print()` peut également afficher le contenu d'une variable quel que soit son type. Par exemple, pour un entier :

```
>>> var = 3
```

```
>>> print(var)
```

3

Il est également possible d'afficher le contenu de plusieurs variables (quel que soit leur type) en les séparant par des virgules :

```
>>> x = 32
>>> nom = "Elinam"
>>> print(nom, " a", x, " ans")
Elinam a 32 ans
```

Python a écrit une phrase complète en remplaçant les variables `x` et `nom` par leur contenu. Vous remarquerez que pour afficher plusieurs éléments de texte sur une seule ligne, nous avons utilisé le séparateur `<< , >>` entre les différents éléments. Python a également ajouté un espace à chaque fois que l'on utilisait le séparateur `<< , >>`. On peut modifier ce comportement en passant à la fonction `print()` l'argument par mot-clé `sep` :

```
>>> x = 32
>>> nom = "Elinam"
>>> print(nom, " a", x, " ans", sep = "")
Elinam a 32 ans
>>> print(nom, " a", x, " ans", sep = " - ")
Elinam -a -32 - ans
```

Écriture formatée

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un certain format, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les floats. L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés en « belles colonnes ».

Prise en main des f-strings

f-string est le diminutif de formatted string literals. Les f-strings permettent une meilleure organisation de l'affichage des variables, mise en place depuis python 3.6. Dans le chapitre précédent, nous avons vu les chaînes de caractères ou encore strings qui étaient représentées par un texte entouré de guillemets simples ou doubles. Par exemple : " Ceci est une chaîne de caractères "

L'équivalent en f-string est tout simplement la même chaîne de caractères précédée du caractère f sans espace entre les deux :

f" Ceci est une chaîne de caract è res "

Ce caractère f avant les guillemets va indiquer à Python qu'il s'agit d'une f-string permettant de mettre en place le mécanisme de l'écriture formatée, contrairement à une string normale.

```
>>> x = 26
>>> Prenom = "Fo - Komi"
>>> print(f"{Prenom} a {x} ans")
Fo-Komi a 26 ans
```

Remarque

Une variable est utilisable plus d'une fois pour une f-string donnée :

```
>>> var = "to"
>>> print(f"{var} et {var} font {var}{var}")
```

On peut également spécifier le format de leur affichage.

```
>>> x = 0.4780405405405405
>>> print("x est égale à :", x)
x est égale à : 0.4780405405405405
>>> print(f"x est égale à {x : .2f}")
x est égale à : 0.48
>>> print(f"x est égale à {x : .3f}")
x est égale à : 0.478
```

La méthode .format()

La méthode .format() permet une meilleure organisation de l'affichage des variables (nous expliquerons à la fin de cette section ce que le terme « méthode » signifie en Python). Si on reprend l'exemple précédent :

```
>>> x = 32
>>> nom = "Elinam"
>>> print("{} a {} ans ".format(nom, x))
Elinam a 32 ans
```

— Dans la chaîne de caractères, les accolades vides {} précisent l'endroit où le contenu de la variable doit être inséré.

— Juste après la chaîne de caractères, l'instruction .format(nom, x) fournie la liste des variables à insérer, d'abord la variable nom puis la variable x. La méthode .format() agit sur la chaîne de caractères à laquelle elle est attachée par le point.

Remarque

Il est possible d'indiquer entre les accolades dans quel ordre afficher les variables, avec 0 pour la variable à afficher en premier, 1 pour la variable à afficher en second, etc. (attention, Python commence à compter à 0). Cela permet de modifier l'ordre dans lequel sont affichées les variables.

```
>>> x = 32
```

```
>>> nom = "Elinam"
```

```
>>> print("{0} a {1} ans ".format(nom, x))
```

Elinam a 32 ans

```
>>> print("{1} a {0} ans ".format(nom, x))
```

32 a Elinam ans

Imaginez maintenant que vous vouliez calculer, puis afficher, la proportion de GC d'un génome. La proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G) du génome considéré. Si on a, par exemple, 4500 bases G et 2575 bases C, pour un total de 14800 bases, vous pourriez procéder comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> prop_GC = (4500 + 2575)/14800
```

```
>>> print(" La proportion de GC est " , prop_GC )
```

La proportion de GC est 0.4780405405405405

Le résultat obtenu présente trop de décimales (seize dans le cas présent).

Pour écrire le résultat plus lisiblement, vous pouvez spécifier dans les accolades {} le format qui vous intéresse. Dans le cas présent, vous voulez formater un float pour l'afficher avec deux puis trois décimales :

```
>>> print (" La proportion de GC est {:.2 f }". format(prop_GC))
```

La proportion de GC est 0.48

```
>>> print (" La proportion de GC est {:.3 f }". format (prop_GC ))
```

La proportion de GC est 0.478

Détaillons le contenu des accolades de la première ligne (`{:.2f}`) :

- Les deux points : indiquent qu'on veut préciser le format.
- La lettre `f` indique qu'on souhaite afficher la variable sous forme d'un float.
- Les caractères `.2` indiquent la précision voulue, soit ici deux chiffres après la virgule.

Notez enfin que le formatage avec `.xf` (`x` étant un entier positif) renvoie un résultat arrondi. Il est par ailleurs possible de combiner le formatage (à droite des 2 points) ainsi que l'emplacement des variables à substituer (à gauche des 2 points), par exemple :

```
>>> print(" prop_GC (2 déci.) = {0:.2 f} , prop_GC (3 déci.) = {0:.3 f}  
      }".format( prop_GC ))
```

```
prop_GC (2 déci.) = 0.48 , prop_GC (3 déci.) = 0.478
```

Remarque

Le signe en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

```
>>> print(" Ce génome contient { : d } G et { : d } C, soit un % GC de  
{ : .2f } %" .format ( nb_G , nb_C , perc_GC ))
```

Ce génome contient 4500 G et 2575 C , soit un % GC de 47.80 %

Définition

Une liste est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. En voici quelques exemples :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux
```

```
['girafe', 'tigre', 'singe', 'souris']
```

```
>>> mixte = ['girafe', 5, 'souris', 0.15]
```

```
>>> mixte
```

```
['girafe', 5, 'souris', 0.15]
```

```
>>> tailles = [5, 2.5, 1.75, 0.15]
```

```
>>> tailles
```

```
[5, 2.5, 1.75, 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie. Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé indice (ou index) de la liste.

```
liste : ['girafe', 'tigre', 'singe', 'souris']
```

```
indice : [0, 1, 2, 3]
```

```
>>> animaux[0]
```

```
' girafe '
```

```
>>> animaux[3]
```

```
' souris '
```

Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication :

```
>>> ani1 = ['girafe', 'tigre']
```

```
>>> ani2 = ['singe', 'souris']
```

Opération sur les listes

```
>>> ani1 + ani2
```

```
['girafe', 'tigre', 'singe', 'souris']
```

```
>>> ani1 * 3
```

```
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'opérateur `+` est très pratique pour concaténer deux listes. Vous pouvez aussi utiliser la méthode `.append()` lorsque vous souhaitez ajouter un seul élément à la fin d'une liste. Dans l'exemple suivant nous allons créer une liste vide :

```
>>> a = []
```

```
>>> a
```

```
[]
```

puis lui ajouter deux éléments, l'un après l'autre, d'abord avec la concaténation :

```
>>> a = a + [15]
```

```
>>> a
```

```
[15]
```

```
>>> a = a + [-5]
```

```
>>> a
```

```
[15, -5]
```

puis avec la méthode `.append()` :

```
>>> a.append(13)
```

```
>>> a
```

```
[15, -5, 13]
```

```
>>> a.append(-3)
```

```
>>> a
```

```
[15, -5, 13, -3]
```

Dans l'exemple ci-dessus, nous ajoutons des éléments à une liste en utilisant l'opérateur de concaténation `+` ou la méthode `.append()`. Nous vous conseillons dans ce cas précis d'utiliser la méthode `.append()` dont la syntaxe est plus élégante.

Indiçage négatif

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

Liste : ['girafe', 'tigre', 'singe', 'souris']

indice positif : 0 1 2 3

Indice négatif : -4 -3 -2 -1

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice -2, l'avant-avant dernier l'indice -3, etc.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux[-1]
```

```
' souris '
```

```
>>> animaux[-2]
```

```
' singe '
```


Pour accéder au premier élément de la liste avec un indice négatif, il faut par contre connaître le bon indice :

```
>>> animaux[-4]
```

```
' girafe '
```

Dans ce cas, on utilise plutôt `animaux[0]`.

Tranches

Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant un indilage construit sur le modèle $[m:n+1]$ pour récupérer tous les éléments, du m ème au n ème (de l'élément m inclus à l'élément $n+1$ exclu). On dit alors qu'on récupère une tranche de la liste, par exemple :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux[0 : 2]
```

```
['girafe', 'tigre']
```

```
>>> animaux[0 : 3]
```

```
['girafe', 'tigre', 'singe']
```

```
>>> animaux[0 :]
```

```
['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux[:]
```

```
['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux[1 :]
```

```
['tigre', 'singe', 'souris']
```

Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement. On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> animaux[0 : 3 : 2]
```

```
['girafe', 'singe']
```

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> x[:: 1]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> x[:: 2]
```

```
[0, 2, 4, 6, 8]
```

```
>>> x[1 : 6 : 3]
```

```
[1, 4]
```

Finalement, on se rend compte que l'accès au contenu d'une liste fonctionne sur le modèle `liste[début:fin:pas]`.

Listes de listes

Sachez qu'il est tout à fait possible de construire des listes de listes. Cette fonctionnalité peut parfois être très pratique. Par exemple :

```
>>> enclos1 = ['girafe', 4]
>>> enclos2 = ['tigre', 2]
>>> enclos3 = ['singé', 5]
>>> zoo = [enclos1, enclos2, enclos3]
>>> zoo
[['girafe', 4], ['tigre', 2], ['singé', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie. Pour accéder à un élément de la liste, on utilise l'indiciage habituel :

```
>>> zoo[1]
['tigre', 2]
```

Pour accéder à un élément de la sous-liste, on utilise un double indiçage :

```
>>> zoo[1][0]
```

```
' tigre '
```

```
>>> zoo[1][1]
```

```
2
```

On verra un peu plus loin qu'il existe en Python des dictionnaires qui sont également très pratiques pour stocker de l'information structurée. On verra aussi qu'il existe un module nommé NumPy qui permet de créer des listes ou des tableaux de nombres (vecteurs et matrices) et de les manipuler.

Les fonctions range(), list() et len()

L'instruction `range()` est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle. Lorsqu'elle est utilisée en combinaison avec la fonction `list()`, on obtient une liste d'entiers. Par exemple :

```
>>> list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La commande `list(range(10))` a généré une liste contenant tous les nombres entiers de 0 inclus à 10 exclu. Nous verrons l'utilisation de la fonction `range()` toute seule dans le chapitre 5 Boucles et comparaisons. Dans l'exemple ci-dessus, la fonction `range()` a pris un argument, mais elle peut également prendre deux ou trois arguments, voyez plutôt :

```
>>> list(range(0, 5))
```

```
[0, 1, 2, 3, 4]
```

```
>>> list(range(15, 20))
```

```
[15, 16, 17, 18, 19]
```

```
>>> list( range (0 , 1000 , 200))  
[0, 200, 400, 600, 800]  
>>> list(range(2, -2, -1))  
[2, 1, 0, -1]
```

L'instruction range() fonctionne sur le modèle range([début,] fin[, pas]). Les arguments entre crochets sont optionnels. Pour obtenir une liste de nombres entiers, il faut l'utiliser systématiquement avec la fonction list(). Enfin, prenez garde aux arguments optionnels par défaut (0 pour début et 1 pour pas) :

```
>>> list(range(10, 0))  
[]
```

Ici la liste est vide car Python a pris la valeur du pas par défaut qui est de 1. Ainsi, si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0. Python génère ainsi une liste vide. Pour éviter ça, il faudrait, par exemple, préciser un pas de -1 pour obtenir une liste d'entiers décroissants :

```
>>> list(range(10, 0, -1))
```

L'instruction `len()` vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> len(animaux)
```

```
4
```

```
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
```

```
8
```


Minimum, maximum et somme d'une liste

Les fonctions `min()`, `max()` et `sum()` renvoient respectivement le minimum, le maximum et la somme d'une liste passée en argument.

```
>>> liste = list(range(10))
```

```
>>> liste
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sum(liste)
```

```
45
```

```
>>> min(liste)
```

```
0
```

```
>>> max(liste)
```

```
9
```

Tuples

A partir des types de base (int, float, etc.), il est possible d'en élaborer de nouveaux. On les appelle des types construits.

Un exemple de type construit est le **tuple**. Il permet de créer une **collection ordonnée de plusieurs éléments**. En mathématiques, on parle de **p-uplet**. Par exemple, un quadruplet est constitué de 4 éléments. Les tuples ressemblent aux listes, mais on ne peut pas les modifier une fois qu'ils ont été créés.

On dit qu'un tuple n'est pas mutable.

On le définit avec des **parenthèses**.

```
>>> a = (3, 4, 7)
>>> type(a)
< class 'tuple' >
```

Dictionnaires

Comme on l'a vu avec les listes et les tuples, à partir des types de base (int, float, etc.) il est possible d'élaborer de nouveaux types qu'on appelle des types construits.

Un nouvel exemple de type construit est le dictionnaire. Les éléments d'une liste ou d'un tuple sont ordonnés et on accède à un élément grâce à sa position en utilisant un numéro qu'on appelle l'indice de l'élément. Un dictionnaire en Python va aussi permettre de rassembler des éléments mais ceux-ci seront identifiés par une clé. On peut faire l'analogie avec un dictionnaire de français où on accède à une définition avec un mot.

Contrairement aux listes qui sont délimitées par des crochets, on utilise des accolades pour les dictionnaires.

```
mon_dictionnaire = {"voiture": "véhicule à quatre roues", "vélo":  
"véhicule à deux roues"}
```

Un élément a été défini ci-dessus dans le dictionnaire en précisant une **clé** au moyen d'une chaîne de caractères suivie de **:** puis de la valeur associée :
clé : valeur

On accède à une valeur du dictionnaire en utilisant la clé entourée par des crochets avec la syntaxe suivante :

```
>>> mon_dictionnaire["voiture"]  
'véhicule à quatre roues'
```

Comment créer un dictionnaire ?

Nous avons vu ci-dessous qu'il était possible de créer un dictionnaire avec des accolades qui entourent les éléments. Une autre approche possible consiste à créer un dictionnaire vide et à ajouter les éléments au fur et à mesure.

```
>>> nombre_de_pneus = {}  
>>> nombre_de_pneus["voiture"] = 4  
>>> nombre_de_pneus["vélo"] = 2  
>>> nombre_de_pneus  
{'voiture' : 4, 'vélo' : 2}
```

Comment construire une entrée dans un dictionnaire ?

Il est très facile d'ajouter un élément à un dictionnaire. Il suffit d'affecter une valeur pour la nouvelle clé.

```
>>> nombre_de_pneus["tricycle"] = "véhicule à trois roues"
```

```
>>> nombre_de_pneus
```

```
{'voiture': 'véhicule à quatre roues', 'vélo': 'véhicule à deux roues',  
'tricycle': 'véhicule à trois roues'}
```

Le type d'un dictionnaire est **dict**.

```
>>> type(mon_dictionnaire)
```

```
dict
```

Comment parcourir un dictionnaire ?

On utilise `items()`.

`mon_dictionnaire.item()` renvoie une liste de tuples de la forme : (clé, valeur).

```
>>> nombre_de_roues = {"voiture" : 4, "vélo" : 2, "tricycle" : 3}
```

```
>>> print(nombre_de_roues.item())
```

```
dict_items([('voiture', 4), ('vélo', 2), ('tricycle', 3)])
```

Boucles for

En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche de manière compacte et efficace. Imaginez par exemple que vous souhaitiez afficher les éléments d'une liste les uns après les autres.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> for animal in animaux :
```

```
>>> .... print(animal)
```

```
.....
```

```
girafe
```

```
tigre
```

```
singe
```

```
Souris
```

Commentons en détails ce qu'il s'est passé dans cet exemple :

La variable `animal` est appelée variable d'itération, elle prend successivement les différentes valeurs de la liste `animaux` à chaque itération de la boucle. On verra un peu plus loin dans ce chapitre que l'on peut choisir le nom que l'on veut pour cette variable. Celle-ci est créée par Python la première fois que la ligne contenant le `for` est exécutée (si elle existait déjà son contenu serait écrasé). Une fois la boucle terminée, cette variable d'itération `animal` ne sera pas détruite et contiendra ainsi la dernière valeur de la liste `animaux` (ici la chaîne de caractères `souris`).

Notez bien les types des variables utilisées ici : `animaux` est une liste sur laquelle on itère, et `animal` est une chaîne de caractères car chaque élément de la liste est une chaîne de caractères. Nous verrons plus loin que la variable d'itération peut être de n'importe quel type selon la liste parcourue. En Python, une boucle itère toujours sur un objet dit séquentiel (c'est-à-dire un objet constitué d'autres objets) tel qu'une liste. Nous verrons aussi plus tard d'autres objets séquentiels sur lesquels on peut itérer dans une boucle.

D'ores et déjà, prêtez attention au caractère deux-points « : » à la fin de la ligne débutant par `for`. Cela signifie que la boucle `for` attend un bloc d'instructions, en l'occurrence toutes les instructions que Python répétera à chaque itération de la boucle. On appelle ce bloc d'instructions le corps de la boucle. Comment indique-t-on à Python où ce bloc commence et se termine ? Cela est signalé uniquement par l'indentation, c'est-à-dire le décalage vers la droite de la (ou des) ligne(s) du bloc d'instructions.

Remarque

Les notions de bloc d'instruction et d'indentations avait été abordées rapidement dans la section 1 Introduction.

Dans l'exemple suivant, le corps de la boucle contient deux instructions : `print(animal)` et `print(animal*2)` car elles sont indentées par rapport à la ligne débutant par `for` :

for `animal` in `animaux` :

`print(animal)`

`print(animal * 2)`

`print(" C'est fini ")`

La ligne 4 `print(" C'est fini")` ne fait pas partie du corps de la boucle car elle est au même niveau que le `for` (c'est- à-dire non indentée par rapport au `for`). Notez également que chaque instruction du corps de la boucle doit être indentée de la même manière (ici 4 espaces).

Il se peut qu'au cours d'une boucle vous ayez besoin des indices, auquel cas vous devrez itérer sur les indices :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> for i in range(len(animaux)):
..... print (" L'animal {} est un(e) {}".format (i , animaux [ i ]))
.....
L ' animal 0 est un ( e ) girafe
L ' animal 1 est un ( e ) tigre
L ' animal 2 est un ( e ) singe
L ' animal 3 est un ( e ) souris
```

Python possède toutefois la fonction `enumerate()` qui vous permet d'itérer sur les indices et les éléments eux-mêmes.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
```

```
>>> for i, animal in enumerate(animaux): ..... print("L' animal est un  
( e ) ".format(i, animal))
```

```
.....
```

```
L ' animal 0 est un ( e ) girafe
```

```
L ' animal 1 est un ( e ) tigre
```

```
L ' animal 2 est un ( e ) singe
```

```
L ' animal 3 est un ( e ) souris
```

Comparaisons

Python est capable d'effectuer toute une série de comparaisons entre le contenu de deux variables, telles que :

| Syntaxe Python | Signification |
|--------------------|---------------------|
| <code>==</code> | égal à |
| <code>!=</code> | différent de |
| <code>></code> | supérieur à |
| <code>>=</code> | supérieur ou égal à |
| <code><</code> | inférieur à |
| <code><=</code> | inférieur ou égal à |

Observez les exemples suivants avec des nombres entiers.

```
>>> x == 5
```

```
True
```

```
>>> x > 10
```

```
False
```

```
>>> x < 10
```

```
True
```

Python renvoie la valeur True si la comparaison est vraie et False si elle est fausse. True et False sont des booléens (un nouveau type de variable). Faites bien attention à ne pas confondre l'opérateur d'affectation = qui affecte une valeur à une variable et l'opérateur de comparaison == qui compare les valeurs de deux variables. Vous pouvez également effectuer des comparaisons sur des chaînes de caractères.

```
>>> animal = "tigre"
```

```
>>> animal == "tig"
```

False

```
>>> animal! = "tig"
```

True

```
>>> animal == 'tigre'
```

True

Dans le cas des chaînes de caractères, a priori seuls les tests == et != ont un sens. En fait, on peut aussi utiliser les opérateurs <, >, <= et >=. Dans ce cas, l'ordre alphabétique est pris en compte, par exemple :

```
>>> "a" < "b"
```

```
True
```

"a" est inférieur à "b" car le caractère a est situé avant le caractère b dans l'ordre alphabétique. En fait, c'est l'ordre ASCII 2 des caractères qui est pris en compte (à chaque caractère correspond un code numérique), on peut donc aussi comparer des caractères spéciaux (comme ou) entre eux. Enfin, on peut comparer des chaînes de caractères de plusieurs caractères :

```
>>> "ali" < "alo"
```

```
True
```

```
>>> "abb" < "ada"
```

```
True
```

Dans ce cas, Python compare les deux chaînes de caractères, caractère par caractère, de la gauche vers la droite (le premier caractère avec le premier, le deuxième avec le deuxième, etc). Dès qu'un caractère est différent entre l'une et l'autre des deux chaînes, il considère que la chaîne la plus petite est celle qui présente le caractère ayant le plus petit code ASCII (les caractères suivants de la chaîne de caractères sont ignorés dans la comparaison), comme dans l'exemple "abb" j "ada" ci-dessus.

Boucles while

Une autre alternative à l'instruction for couramment utilisée en informatique est la boucle while. Le principe est simple. Une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i = 1
>>> while i <= 4:
.....     print(i)
.....     i = i + 1
.....
1
2
3
4
```

Remarquez qu'il est encore une fois nécessaire d'indenter le bloc d'instructions correspondant au corps de la boucle (ici, les instructions lignes 3 et 4). Une boucle while nécessite généralement trois éléments pour fonctionner correctement :

1. Initialisation de la variable d'itération avant la boucle (ligne 1).
2. Test de la variable d'itération associée à l'instruction while (ligne 2).
3. Mise à jour de la variable d'itération dans le corps de la boucle (ligne 4).

Faites bien attention aux tests et à l'incrémentation que vous utilisez car une erreur mène souvent à des « boucles infinies » qui ne s'arrêtent jamais. Vous pouvez néanmoins toujours stopper l'exécution d'un script Python à l'aide de la combinaison de touches Ctrl-C (c'est-à-dire en pressant simultanément les touches Ctrl et C). Par exemple :

```
i = 0
while i < 10 :
    print (" Le python c'est cool !")
```

Ici, nous avons omis de mettre à jour la variable `i` dans le corps de la boucle. Par conséquent, la boucle ne s'arrêtera jamais (sauf en pressant Ctrl-C) puisque la condition `i < 10` sera toujours vraie. La boucle while combinée à la fonction `input()` peut s'avérer commode lorsqu'on souhaite demander à l'utilisateur une valeur numérique. Par exemple :

```
>>> i = 0
```

```
>>> while i < 10:
```

```
....     reponse = input("Entrez un entier supérieur à 10 : ")
```

```
....     i = int(reponse)
```

```
.....
```

Entrez un entier supérieur à 10

Entrez un entier supérieur à 10

Entrez un entier supérieur à 10

»»» *i*

15

La fonction `input()` prend en argument un message (sous la forme d'une chaîne de caractères), demande à l'utilisateur d'entrer une valeur et renvoie celle-ci sous forme d'une chaîne de caractères. Il faut ensuite convertir cette dernière en entier (avec la fonction `int()`).

Boucles infinie : while True

Supposons que vous écriviez une boucle while qui théoriquement ne se termine jamais. Cela semble bizarre, non?

Considérez cet exemple :

```
>>> while True :
```

```
....     print('foo')
```

```
foo
```

```
foo
```

```
.
```

```
.
```

```
foo
```

```
foo
```

```
foo
```

KeyboardInterrupt

Traceback (most recent call last):

File "< pyshell2 > ", line 2, in < module >

```
print('foo')
```

Ce code a été terminé par Ctrl+C, qui génère une interruption à partir du clavier. Sinon, cela aurait continué sans fin. De nombreuses lignes de sortie foo ont été supprimées et remplacées par les points de suspension verticaux dans la sortie illustrée.

De toute évidence, True ne sera jamais faux, ou nous avons tous de très gros problèmes. Ainsi, tandis que True : initie une boucle infinie qui s'exécutera théoriquement pour toujours.

Peut-être que cela ne ressemble pas à quelque chose que vous voudriez faire, mais ce modèle est en fait assez courant. Par exemple, vous pouvez écrire du code pour un service qui démarre et s'exécute indéfiniment en acceptant les demandes de service. "Pour toujours" dans ce contexte signifie jusqu'à ce que vous l'arrêtiez, ou jusqu'à la mort thermique de l'univers, selon la première éventualité.

Les boucles pouvantt être rompues avec l'instruction break, il peut être plus simple de terminer une boucle en fonction des conditions reconnues dans le corps de la boucle, plutôt que sur une condition évaluée au sommet.

Voici une autre variante de la boucle illustrée ci-dessus qui supprime successivement les éléments d'une liste à l'aide de .pop() jusqu'à ce qu'elle soit vide :

```
>>> a = ['foo', 'bar', 'baz']
```

```
>>> while True:
```

```
....     if not a:
```

```
....         break
```

```
....     print(a.pop(-1))
```

```
....
```

```
baz
```

```
bar
```

```
foo
```

Tests

Les tests sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions. Pour cela, Python utilise l'instruction `if` ainsi qu'une comparaison que nous avons abordée au chapitre précédent. Voici un premier exemple :

```
>>> x = 2
>>> if x == 2:
....     print("Le test est vrai !")
....
Le test est vrai !
et un second :
>>> x = " souris"
>>> if x == " tigre ":
....     print("Le test est vrai !")
....
```


Il y a plusieurs remarques à faire concernant ces deux exemples :

- Dans le premier exemple, le test étant vrai, l'instruction `print("Le test est vrai !")` est exécutée. Dans le second exemple, le test est faux et rien n'est affiché.

- Les blocs d'instructions dans les tests doivent forcément être indentés comme pour les boucles `for` et `while`. L'indentation indique la portée des instructions à exécuter si le test est vrai.

- Comme avec les boucles `for` et `while`, la ligne qui contient l'instruction `if` se termine par le caractère deux-points `<< : >>`.

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques. Vous connaissez déjà des fonctions internes à Python comme `range()` ou `len()`. Pour l'instant, une fonction est à vos yeux une sorte de « boîte noire » :

1. À laquelle vous passez aucune, une ou plusieurs variable(s) entre parenthèses. Ces variables sont appelées arguments. Il peut s'agir de n'importe quel type d'objet Python.
2. Qui effectue une action.
3. Et qui renvoie un objet Python ou rien du tout.

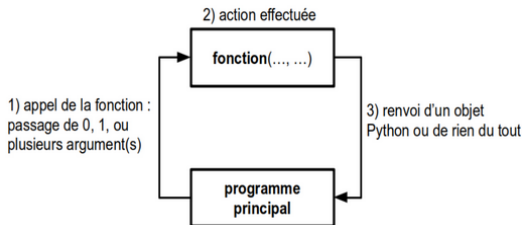
Par exemple, si vous appelez la fonction `len()` de la manière suivante :

```
>>> len ([0 , 1 , 2])
```

```
3
```

- voici ce qui se passe : 1. vous appelez `len()` en lui passant une liste en argument (ici la liste `[0, 1, 2]`) ;
2. la fonction calcule la longueur de cette liste ;
3. elle vous renvoie un entier égal à cette longueur.

Autre exemple, si vous appelez la méthode `ma_liste.append()` (n'oubliez pas, une méthode est une fonction qui agit sur l'objet auquel elle est attachée par un point) :



Aux yeux du programmeur au contraire, une fonction est une portion de code effectuant une suite d'instructions bien particulière. Mais avant de vous présenter la syntaxe et la manière de construire une fonction, revenons une dernière fois sur cette notion de « boîte noire » :

- Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement quelque chose. L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus. On a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur
- Chaque fonction effectue en général une tâche unique et précise. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette modularité améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.

Définition

Pour définir une fonction, Python utilise le mot-clé **def**. Si on souhaite que la fonction renvoie quelque chose, il faut utiliser le mot-clé **return**. Par exemple :

```
>>> def carre(x):  
.....     return x**2  
.....  
>>> print(carre(2))  
4
```

Notez que la syntaxe de définition utilise les deux-points comme les boucles `for` et `while` ainsi que les tests `if`, un bloc d'instructions est donc attendu. De même que pour les boucles et les tests, l'indentation de ce bloc d'instructions (qu'on appelle le corps de la fonction) est obligatoire.

Dans l'exemple précédent, nous avons passé un argument à la fonction `carre()` qui nous a renvoyé (ou retourné) une valeur que nous avons immédiatement affichée à l'écran avec l'instruction `print()`. Que veut dire valeur renvoyée ? Et bien cela signifie que cette dernière est récupérable dans une variable : `>>> res = carre(2)`

```
>>> print(res)
```

```
4
```

Ici, le résultat renvoyé par la fonction est stocké dans la variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
```

```
.....    print(" bonjour ")
```

```
.....
```

```
>>> hello()
```

```
bonjour
```

Passage d'arguments

Le nombre d'arguments que l'on peut passer à une fonction est variable. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui développe une nouvelle fonction. Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au « typage dynamique », c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution. Par exemple :

```
>>> def fois(x, y):  
....     return x*y  
....  
>>> fois(2, 3)  
6  
>>> fois(3.1415, 5.23)  
16.430045000000003
```

```
>>> fois(' to ', 2)
' toto '
>>> fois([1 ,3], 2)
, 3 , 1 , 3
```

L'opérateur * reconnaît plusieurs types (entiers, floats, chaînes de caractères, listes). Notre fonction fois() est donc capable d'effectuer des tâches différentes ! Même si Python autorise cela, méfiez-vous tout de même de cette grande flexibilité qui pourrait conduire à des surprises dans vos futurs programmes. En général, il est plus judicieux que chaque argument ait un type précis (entiers, floats, chaînes de caractères, etc) et pas l'un ou l'autre.

Renvoi de résultats

Un énorme avantage en Python est que les fonctions sont capables de renvoyer plusieurs objets à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):  
....     return x**2, x**3  
....  
>>> carre_cube(2)  
(4 , 8)
```

En réalité Python ne renvoie qu'un seul objet, mais celui-ci peut être séquentiel, c'est-à-dire contenir lui même d'autres objets. Dans notre exemple Python renvoie un objet de type tuple, (à chercher, grosso modo, il s'agit d'une sorte de liste avec des propriétés différentes).

Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):  
....     return [x**2, x**3]  
....  
>>> carre_cube2(3)
```

, 27

Renvoyer un tuple ou une liste de deux éléments (ou plus) est très pratique en conjonction avec l'affectation multiple, par exemple :

```
>>> z1 , z2 = carre_cube2(3)  
>>> z1  
9  
>>> z2  
27
```

Cela permet de récupérer plusieurs valeurs renvoyées par une fonction et de les affecter à la volée à des variables différentes.

- Lorsqu'on définit une fonction `def fct(x, y)`: les arguments `x` et `y` sont appelés arguments positionnels (en anglais `positional arguments`). Il est strictement obligatoire de les préciser lors de l'appel de la fonction. De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction. Dans l'exemple ci-dessus, 2 correspondra à `x` et 3 correspondra à `y`. Finalement, tout dépendra de leur position, d'où leur qualification de positionnel
- Un argument défini avec une syntaxe `def fct(arg=val)`: est appelé argument par mot-clé (en anglais `keyword argument`). Le passage d'un tel argument lors de l'appel de la fonction est facultatif. Ce type d'argument ne doit pas être confondu avec les arguments positionnels présentés ci-dessus, dont la syntaxe est de `fct(arg)`:

Il est bien sûr possible de passer plusieurs arguments par mot-clé :

```
>>> def fct(x =0, y =0, z =0):
```

```
....     return x, y, z
```

```
....
```

```
>>> fct()
```

```
(0 , 0 , 0)
```

```
>>> fct(10) (10 , 0 , 0)
```

```
>>> fct(10 , 8)
```

```
(10 , 8 , 0)
```

```
>>> fct(10 , 8 , 3)
```

```
(10 , 8 , 3)
```

On observe que pour l'instant, les arguments par mot-clé sont pris dans l'ordre dans lesquels on les passe lors de l'appel. Comment pourrions-nous faire si on souhaitait préciser l'argument par mot-clé z et garder les valeurs de x et y par défaut ? Simplement en précisant le nom de l'argument lors de l'appel :

```
>>> fct( z =10)
```

```
(0 , 0 , 10)
```

Python permet même de rentrer les arguments par mot-clé dans un ordre arbitraire :

```
>>> fct(z=10, x=3, y=80)
(3, 80, 10)
>>> fct(z=10, y=80)
(0, 80, 10)
```

Que se passe-t-il lorsque nous avons un mélange d'arguments positionnels et par mot-clé ? Et bien les arguments positionnels doivent toujours être placés avant les arguments par mot-clé :

```
>>> def fct(a, b, x=0, y=0, z=0):
....     return a, b, x, y, z
....
>>> fct(1, 1)
(1, 1, 0, 0, 0)
```

```
>>> fct(1, 1, z =5)
(1, 1, 0, 0, 5)
>>> fct(1, 1, z =5, y =32)
(1, 1, 0, 32, 5)
```

On peut toujours passer les arguments par mot-clé dans un ordre arbitraire à partir du moment où on précise leur nom. Par contre, si les deux arguments positionnels *a* et *b* ne sont pas passés à la fonction, Python renvoie une erreur.

```
>>> fct(z =0)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError : fct() missing 2 required positional arguments : 'a' and 'b'

Variables locales et variables globales

Les variables définies dans une fonction sont appelées variables locales. Elles ne peuvent être utilisées que localement c'est-à-dire qu'à l'intérieur de la fonction qui les a définies. Tenter d'appeler une variable locale depuis l'extérieur de la fonction qui l'a définie provoquera une erreur.

Cela est dû au fait que chaque fois qu'une fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel espace de noms (c'est-à-dire une sorte de dossier virtuel). Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Cet espace de noms est automatiquement détruit dès que la fonction a terminé son travail, ce qui fait que les valeurs des variables sont réinitialisées à chaque nouvel appel de fonction.

```
# définition d'une fonction carre()  
def carre( x ):  
    y = x**2 # y variable locale  
    return y
```

Les variables définies dans l'espace global du script, c'est-à-dire en dehors de toute fonction sont appelées des variables globales. Ces variables sont accessibles (= utilisables) à travers l'ensemble du script et accessible en lecture seulement à l'intérieur des fonctions utilisées dans ce script.

```
# programme principal  
z = 5 # variable globale  
resultat = carre(z)  
print(resultat)
```


Pour le dire très simplement : une fonction va pouvoir utiliser la valeur d'une variable définie globalement mais ne va pas pouvoir modifier sa valeur c'est-à-dire la redéfinir. En effet, toute variable définie dans une fonction est par définition locale ce qui fait que si on essaie de redéfinir une variable globale à l'intérieur d'une fonction on ne fera que créer une autre variable de même nom que la variable globale qu'on souhaite redéfinir mais qui sera locale et bien distincte de cette dernière.

variable x globale

```
>>> x = 10
```

```
>>> def portee():
```

```
....     print(x)
```

```
>>> portee()
```

```
10
```

```
>>> def portee2():
```

```
....     x = 20 # variable x locale
```

```
....     print(x)
```

```
!!! portee2()
```

```
20
```

Modifier une variable globale depuis une fonction

Dans certaines situations, il serait utile de pouvoir modifier la valeur d'une variable globale depuis une fonction, notamment dans le cas où une fonction se sert d'une variable globale et la manipule.

Cela est possible en Python. Pour faire cela, il suffit d'utiliser le mot clef **global** devant le nom d'une variable globale utilisée localement afin d'indiquer à Python qu'on souhaite bien modifier le contenu de la variable globale et non pas créer une variable locale de même nom.

#utilisation du mot clé global

```
>>> x = 10
>>> def portee():
....     global x
....     x = 5
>>> portee()
>>> print(x)
5
```

Méthode `.readlines()`

Créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom `zoo.txt` et le contenu suivant :

girafe

tigre

singe

Souris

Ensuite, testez le code suivant dans l'interpréteur Python :

```
>>> filin = open("zoo.txt ", " r ")
>>> filin
< _io. TextIOWrapper name = ' zoo . txt ' mode = 'r ' encoding = '
UTF-8 '>
>>> filin.readlines()
girafe ' , ' tigre ' , ' singe ' , ' souris '
>>> filin.close ()
>>> filin.readlines ()
Traceback ( most recent call last ):
File " < stdin > " , line 1 , in < module >
ValueError : I / O operation on closed file
```

Il y a plusieurs commentaires à faire sur cet exemple :

Ligne 1. L'instruction `open()` ouvre le fichier `zoo.txt`. Ce fichier est ouvert en lecture seule, comme l'indique le second argument `r` (pour `read`) de la fonction `open()`. Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu). Le curseur de lecture est prêt à lire le premier caractère du fichier. L'instruction `open("zoo.txt", "r")` suppose que le fichier `zoo.txt` est dans le répertoire depuis lequel l'interpréteur Python a été lancé. Si ce n'est pas le cas, il faut préciser le chemin d'accès au fichier. Par exemple, `/home/pierre/zoo.txt` pour Linux ou Mac OS X ou `C:.txt` pour Windows.

Ligne 2. Lorsqu'on affiche le contenu de la variable `filin`, on se rend compte que Python la considère comme un objet de type fichier ouvert (**Ligne 3**).

Ligne 4. Nous utilisons à nouveau la syntaxe `objet.méthode()` (présentée dans le chapitre POO). Ici la méthode `.readlines()` agit sur l'objet `filin` en déplaçant le curseur de lecture du début à la fin du fichier, puis elle renvoie une liste contenant toutes les lignes du fichier (dans notre analogie avec un livre, ceci correspondrait à lire toutes les lignes du livre).

Ligne 6. Enfin, on applique la méthode `.close()` sur l'objet `filin`, ce qui, vous vous en doutez, ferme le fichier (ceci correspondrait à fermer le livre). Vous remarquerez que la méthode `.close()` ne renvoie rien mais modifie l'état de l'objet `filin` en fichier fermé. Ainsi, si on essaie de lire à nouveau les lignes du fichier, Python renvoie une erreur car il ne peut pas lire un fichier fermé (lignes 7 à 10).

Voici maintenant un exemple complet de lecture d'un fichier avec Python.

```
>>> filin = open(" zoo.txt " , " r ")
>>> lignes = filin.readlines()
>>> lignes
girafe ' , ' tigre ' , ' singe ' , ' souris '
>>> for ligne in lignes:
....     print(ligne)
....
girafe
tigre
singe
souris
>>> filin.close()
```

- Chaque élément de la liste lignes est une chaîne de caractères. C'est en effet sous forme de chaînes de caractères que Python lit le contenu d'un fichier
- Chaque élément de la liste lignes se termine par le caractère `\n`. Ce caractère un peu particulier correspond au «saut de ligne 1» qui permet de passer d'une ligne à la suivante. Ceci est codé par un caractère spécial que l'on représente par `\n`. Vous pourrez parfois rencontrer également la notation octale `\123`.
- Par défaut, l'instruction `print()` affiche quelque chose puis revient à la ligne. Ce retour à la ligne dû à `print()` se cumule alors avec celui de la fin de ligne (`\n`) de chaque ligne du fichier et donne l'impression qu'une ligne est sautée à chaque fois.

Il existe en Python le mot-clé **with** qui permet d'ouvrir et de fermer un fichier de manière efficace. Si pour une raison ou une autre l'ouverture ou la lecture du fichier conduit à une erreur, l'utilisation de **with** garantit la bonne fermeture du fichier, ce qui n'est pas le cas dans le code précédent. Voici donc le même exemple avec **with** :

```
>>> with open(" zoo.txt " , 'r ') as filin :
```

```
....     lignes = filin.readlines()
```

```
....     for ligne in lignes:
```

```
....         print(ligne)
```

```
....
```

```
girafe
```

```
tigre
```

```
singe
```

```
Souris
```

Remarque

- L'instruction `with` introduit un bloc d'indentation. C'est à l'intérieur de ce bloc que nous effectuons toutes les opérations sur le fichier.
- Une fois sorti du bloc d'indentation, Python fermera automatiquement le fichier. Vous n'avez donc plus besoin d'utiliser la méthode `.close()`.

Méthode `.read()`

Il existe d'autres méthodes que `.readlines()` pour lire (et manipuler) un fichier. Par exemple, la méthode `.read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

```
>>> with open("zoo.txt " , " r ") as filin:  
....     filin.read()  
....
```

Méthode `.readline()`

La méthode `.readline()` (sans `s` à la fin) lit une ligne d'un fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de `.readline()`, la ligne suivante est renvoyée. Associée à la boucle `while`, cette méthode permet de lire un fichier ligne par ligne.

```
>>> with open ("zoo.txt " , " r ") as filin:
```

```
....     ligne = filin.readline()
....     while ligne != " ":
....         print( ligne )
....         ligne = filin.readline()
....
```

girafe

tigre

singe

souris

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
>>> animaux2 = [" poisson " , " abeille " , " chat " ] >>> with open  
("zoo2.txt " , " w ") as filout:
```

```
....     for animal in animaux2:
```

```
....     filout.write( animal )
```

```
....
```

```
7
```

```
7
```

```
4
```

Quelques commentaires sur cet exemple :

Ligne 1. Création d'une liste de chaînes de caractères animaux2.

Ligne 2. Ouverture du fichier zoo2.txt en mode écriture, avec le caractère w pour write. L'instruction with crée un bloc d'instructions qui doit être indenté.

Ligne 3. Parcours de la liste animaux2 avec une boucle for.

Ligne 4. À chaque itération de la boucle, nous avons écrit chaque élément de la liste dans le fichier. La méthode .write() s'applique sur l'objet filout. Notez qu'à chaque utilisation de la méthode .write(), celle-ci nous affiche le nombre d'octets (équivalent au nombre de caractères) écrits dans le fichier (lignes 6 à 8). Ceci est valable uniquement dans l'interpréteur, si vous créez un programme avec les mêmes lignes de code, ces valeurs ne s'afficheront pas à l'écran. Si nous ouvrons le fichier zoo2.txt avec un éditeur de texte, voici ce que nous obtenons : poissonabeillechat.

Ce n'est pas exactement le résultat attendu car implicitement nous voulions le nom de chaque animal sur une ligne. Nous avons oublié d'ajouter le caractère fin de ligne après chaque nom d'animal. Pour ce faire, nous pouvons utiliser l'écriture formatée :

```
>>> animaux2 = [" poisson " , " abeille " , " chat "]
```

```
>>> with open ("zoo2.txt " , " w ") as filout:
```

```
....     for animal in animaux2:
```

```
....         filout.write("{}\n".format( animal ))
```

```
....
```

```
8
```

```
8
```

```
5
```

Ligne 5. L'écriture formatée vue à la section 3 Affichage, permet d'ajouter un retour à la ligne (`\n`) après le nom de chaque animal. **Ligne 6 à 8.** Le nombre d'octets écrits dans le fichier est augmenté de 1 par rapport à l'exemple précédent car le caractère retour à la ligne compte pour un seul octet. Le contenu du fichier `zoo2.txt` est alors :

```
poisson
abeille
chat
```

Ouvrir deux fichiers avec l'instruction with

On peut avec l'instruction with ouvrir deux fichiers (ou plus) en même temps. Voyez l'exemple suivant :

```
with open("zoo.txt " , " r ") as fichier1 , open ("zoo2.txt " , " w ") as  
fichier2:
```

```
    for ligne in fichier1:  
        fichier2.write ("*" + ligne )
```

Si le fichier zoo.txt contient le texte suivant :

souris
girafe
lion
Singe

alors le contenu de zoo2.txt sera :

- * souris
- * girafe
- * lion
- * singe

Dans cet exemple, with permet une notation très compacte en s'affranchissant de deux méthodes .close(). Si vous souhaitez aller plus loin, sachez que l'instruction with est plus générale et est utilisable dans d'autres contextes.