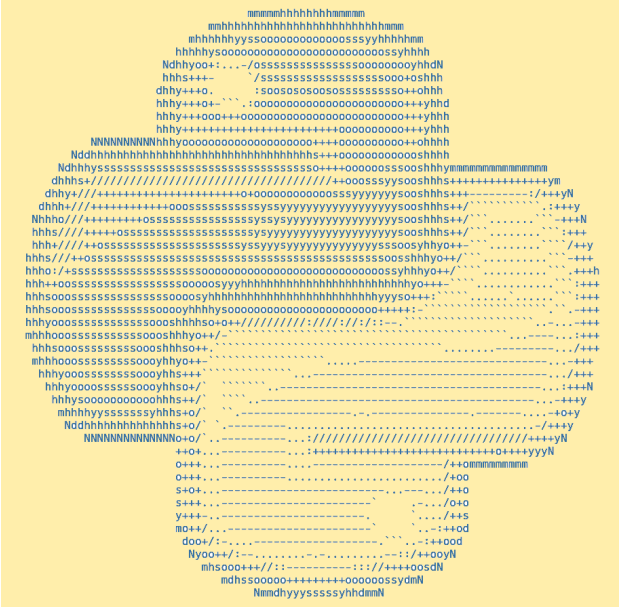
Pascal ORTIZ



Les chaînes de caractères

Version du 11 novembre 2023

Table des matières

Chapitre I

Présentation, opérations, conversions

Les chaînes : premier contact

La notion de chaîne

Les chaînes de caractères permettent de stocker du **texte**. En première approximation, une chaîne est une succession de caractères imprimables. Cela peut être, par exemple, une phrase ou un groupe de mots, comme *Le nombre 42 est magique.*

On dit souvent *chaîne* au lieu de *chaîne de caractères.*

De la même façon qu'un programme peut travailler avec des entiers, il peut travailler avec des chaînes.

Sensibilité à la casse

La *casse* d'un caractère alphabétique est le fait que le caractère soit en majuscule ou en minuscule.

Les chaînes *bonjour* et *bonJour* sont considérées en Python comme distinctes. Si deux chaînes ont, à des positions identiques, des lettres de casses différentes, les chaînes sont considérées comme distinctes.

Chaîne littérale

Le moyen le plus immédiat d'utiliser des chaînes dans un code Python est par le biais d'une *chaîne littérale* :

```
1 s = "orange"
2 print(len(s))
3
4 t = 'rose'
5 print(len(t))
```

```
6 6
7 4
```

À la ligne 1, on lit `"orange"` : c'est une chaîne, dite *littérale.* A la ligne 2, on demande l'affichage du nombre de caractères de la chaîne. On lit à la ligne 6 la longueur de la chaîne.

Une chaîne est *littérale* lorsque

— tous ses caractères sont placés dans le **code-source**,

— la chaîne est encadrée par des caractères particuliers, appelés *quotes*, ici des guillemets anglo-saxons (mais il existe d'autres délimiteurs possibles).

Afficher une chaîne

Pour afficher une chaîne avec une « commande » Python, on utilisera obligatoirement la fonction `print` :

```
1 fruit = "orange"
2 print(fruit)
3 print("bonjour")
```

```
4 orange
5 bonjour
```

— Lignes 2 et 4 : ici, on affiche une chaîne par l'intermédiaire d'une variable.
— Lignes 3 et 5 : affichage direct d'une chaîne littérale.
— On notera que les délimiteurs de chaîne ne sont pas affichés.

Dans certains environnement (mode interactif, cellule Jupyter Notebook), une chaîne peut être affichée sans recours à la fonction `print`.

Erreur courante de débutant

Pour afficher une chaîne littérale, on fera attention de ne pas oublier les quotes autour de la chaîne :

```
print(bonjour)
```

```
NameError: name 'bonjour' is not defined
```

La notion de caractère

L'unité fondamentale d'un texte est le caractère. Un mot, une phrase est composée de caractères. On peut disposer de certains caractères en les plaçant entre guillemets, par exemple la lettre `"Y"` qui est en fait une chaîne d'un seul caractère (alors qu'on en écrit trois dans le code si on compte les deux guillemets).

Parmi les caractères, on trouve les caractères alphabétiques majuscules ou minuscules mais aussi tous les caractères de ponctuation, les chiffres, par exemple le chiffre `"8"`.

On dispose aussi de caractères invisibles comme l'espace `" "` (taper sur la barre d'espace) ou encore le saut de ligne. Le **blanc souligné** _ est aussi un caractère très utilisé. Il existe aussi des caractères non imprimables et qui peuvent se retrouver dans une chaîne.

Python supporte l'encodage UTF-8 et donne donc accès à tous les caractères imaginables.

En pratique, dans un cours de débutant sur les chaînes de caractères, on essaye de ne pas avoir à utiliser des caractères tels que le guillemet ou l'apostrophe car ils ont rôle spécial pour construire des chaînes dites *littérales*.

La notion de chaîne littérale

Une chaîne littérale est un chaîne de caractères **saisie telle quelle dans le code-source** entre une paire de délimiteurs. Ainsi, `"orange"` est une chaîne littérale, le délimiteur étant ici un

guillemet :

```
s = "orange"
print(s)
```

```
orange
```

Les délimiteurs **ne** font **pas** partie du contenu d'une chaîne littérale, comme on le voit à l'affichage.

Insistons : une chaîne littérale n'a de sens que dans un **code-source** puisque *littéral* veut dire qui s'interprète *littéralement* autrement dit, « lettre à lettre », et c'est bien comme cela qu'elle apparaît dans le code-source Python.

Le délimiteur de chaîne littérale le plus familier est le guillemet anglais ". Mais un autre délimiteur usuel est l'apostrophe :

```
1 s = "rose"
2 t = 'rose'
3 print(s)
4 print(t)
```

```
5 rose
6 rose
```

— Ligne 1 : chaîne littérale entourée de guillemets

— Ligne 2 : chaîne littérale entourée d'apostrophes

— Lignes 5-6 : les chaînes `s` et `t` sont égales.

Pour ce qui est du contenu d'une chaîne, par défaut, les caractères d'une chaîne Python (littérale ou pas d'ailleurs) appartiennent au jeu Unicode et donc peuvent comporter des accents ou être des caractères spéciaux. Par exemple, la chaîne littérale `s` suivante est valide :

```
s = "Les pièces de 5€ n'existent pas"
print(s)
```

```
Les pièces de 5€ n'existent pas
```

Opérations sur des chaînes

On peut faire de nombreuses opérations sur les chaînes. Voici juste quelques exemples et qui seront détaillés ailleurs.

Nombre de caractères

Déterminer avec la fonction standard `len` le nombre de caractères d'une chaîne, ce qu'on appelle sa *longueur* :

```
1 print(len("anticonstitutionnellement"))
```

```
2 25
```

— Ligne 1 : la fonction standard `len`, fournie par Python, effectue le calcul de la longueur.

Concaténer deux chaînes

L'opérateur + permet de mettre bout à bout deux chaînes :

```
1 print("bon" + "Jour")
```

```
2 bonJour
```

— Ligne 1 : la *concaténation* (c'est le terme consacré) est effectuée avec l'opérateur +.

— Ligne 2 : noter que les deux chaînes **bon** et **Jour** sont placées bout à bout, sans espace entre les chaînes.

Détail : l'expression `"bon" + "Jour"` est la concaténation de deux chaînes littérales et n'est pas une chaîne littérale.

Chaîne vide

Une chaîne peut être vide autrement dit ne contenir aucun caractère. Une chaîne vide peut être représentée par la chaîne littérale `""` (deux guillemets collés l'un à l'autre et qui n'entourent rien). La chaîne vide est de longueur nulle :

```
vide = ""

print(len(vide))
```

```
0
```

La chaîne vide peut aussi être notée `''` c'est-à-dire deux apostrophes côte-à-côte, sans espace entre les deux apostrophes :

```
vide = ''

# une chaîne vide a une longueur nulle
print(len(vide))
```

```
0
```

La chaîne vide a de nombreux usages (comme le chiffre zéro en a !). Par exemple, une chaîne vide peut être utilisée pour supprimer des caractères à l'intérieur d'une chaîne en remplaçant les caractères par une chaîne vide.

Le caractère *espace*

De même qu'il existe un caractère `A` désignant la lettre `A`, il existe un caractère *espace*, celui obtenu en tapant sur la barre d'espaces. Plus précisément, le contenu de la chaîne `" "` est un espace :

```
1 print("Bug" + "2038")
2 print("Bug" + " " + "2038")
3 print("Bug 2038")
```

```
Bug2038
Bug 2038
Bug 2038
```

— Ligne 1 : Les chaînes `"Bug"` et `"2038"` sont concaténées sans insertion d'espace.
— Lignes 2 et 3 : les chaînes affichées sont identiques : l'espace peut-être considéré comme une chaîne littérale (ligne 2) ou immergé dans la chaîne (ligne 3).

Invisible n'est pas vide

La chaîne `" "` n'est pas vide, elle contient exactement un caractère :

```
espace = " "
print(len(espace))
```

```
1
```

Le caractère *espace* sert aussi à créer une indentation en plaçant des « blancs » en début de lignes. Pour dessiner en mode texte, la figure ci-dessous :

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
*******************
          #
          #
```

il faut placer des caractères espace en début de ligne.

Le caractère *saut de ligne*

Dans beaucoup de langages de programmation, y compris Python, le codage du caractère *saut de ligne* est noté \n. L'usage de ce groupe de **deux** caractères permet effectivement de signifier un saut de ligne dans une chaîne (littérale) :

```
s = "Lundi\nMardi\nMercredi"
print(s)
```

```
Lundi
Mardi
Mercredi
```

On notera que le caractère saut de ligne \n n'a pas de raison d'être d'être suivi d'un espace littéral qui provoquerait un effet peut-être non désiré :

```
s = "Lundi\n Mardi\nMercredi"
print(s)
```

```
3 Lundi
4  Mardi
5 Mercredi
```

— Ligne 1 : noter l'espace dans le code source, dans la chaîne littérale, juste avant le mot *Mardi*.

— Ligne 4 : l'espace est compté dans la chaîne et visible à l'affichage.

Bien que le caractère littéral représentant un saut de ligne s'écrive avec **deux** caractères, ces deux caractères sont insécables et ils comptent pour un caractère dans la longueur de la chaîne vue comme objet Python. Ainsi :

```
s="rose\nbleu"
print(s)
print(len(s))
```

```
rose
bleu
9
```

Plusieurs sauts de ligne

Pour aérer un texte, on peut souhaiter y placer plusieurs sauts de ligne l'un après l'autre. Si on place côte-à-côte deux caractères `\n`, on effectue deux passages à la ligne, ce qui va créer une ligne blanche :

```
texte = "rose\n\nkiwi"
print(texte)
```

```
rose

kiwi
```

Une ligne blanche n'est jamais vide et est constituée d'au moins un caractère, le saut de ligne et éventuellement des espaces en début de ligne.

Concaténation et répétition de chaînes

On peut concaténer des chaînes, c'est-à-dire les mettre bout à bout. La concaténation de chaînes est effectuée avec l'opérateur `+` :

```
1 a = "Rose"
2 b = "Rouge"
3 c = "Rome"
4 s = a + b + c
5 print(s)
6 print(a, b, c)
```

```
7 RoseRougeRome
8 Rose Rouge Rome
```

— Ligne 4 : `s` est la concaténation des chaînes `a`, `b` et `c`.

— Lignes 5 et 8 : la concaténation des chaînes crée une **nouvelle** chaîne et préserve les chaînes initiales.

On peut aussi concaténer n fois une chaîne avec elle-même avec l'opérateur *

```
1 a = "Rose"
2 r = 3 * a
3 print(r)
4 print(a * 3)
5 print(2 * a *2)
```

```
6 RoseRoseRose
7 RoseRoseRose
8 RoseRoseRoseRose
```

— Ligne 3 ou 4 : a est répétée 3 fois

— Lignes 5 : on peut multiplier des deux côtés.

On peut aussi panacher addition et multiplication :

```
a = "Rose"
b = "Rouge"
c = "Rome"
s = 2 * (c + a + ' ') + 2 * (c + b + ' ')
print(s)
```

```
RomeRose RomeRose RomeRouge RomeRouge
```

Concaténation et produit par zéro

L'opération `0 * s` renvoie la chaîne vide et `1 * s`. On peut en déduire une astuce pour mettre un nom au pluriel en ajoutant, selon l'effectif, un `"s"` terminal :

```
for val in (0, 1, 7):
    print(val, "bille" + (val > 1)*"s")
```

```
0 bille
1 bille
7 billes
```

Voici 3 autres façons de réaliser la même opération

```
for val in (0, 1, 7):
    print(val, "bille" + ["s", ""][val < 2])

print("---------------")
for val in (0, 1, 7):
    print(val, "bille" + ("" if val < 2 else 's'))

print("---------------")
for val in (0, 1, 7):
    print(val,  f"bille{'s'[val<2:]}")
```

et qui affiche :

```
0 bille
1 bille
7 billes
---------------
0 bille
1 bille
7 billes
---------------
0 bille
1 bille
7 billes
```

Chaînes égales

L'opérateur == permet de comparer deux chaînes pour savoir si elles ont ou pas exactement les mêmes caractères et dans le même ordre :

```
1 print("ROSE" == "ROSE")
2 print("RoseRose" == "Rose" + "Rose")
3 print("Rose" == "rose")
4 print("ROSE" == 'ROSE')
```

```
5 False
6 True
7 False
8 True
```

— Ligne 1 : Malgré les apparences, les chaînes sont distinctes : à l'indice 1, l'une utilise la lettre `O` majuscule et l'autre un zéro.

— Ligne 2 : `"Rose" + "Rose"` est la concaténation de deux chaînes.

— Ligne 3 : Pour que deux caractères soient égaux, il faut déjà qu'ils aient la même casse, ce qui n'est pas le cas de la lettres initiale.

— Ligne 4 : Des délimiteurs apostrophes ou guillemets n'ont pas d'influence sur l'égalité de chaînes.

Attention que l'égalité de deux chaînes est une opération ayant une complexité cachée. En effet, pour savoir si deux chaînes sont égales ou pas, l'interpréteur Python compare terme à terme les éléments des deux chaînes jusqu'à ce que deux caractères soints distincts ou bien qu'une des chaînes ne possède plus d'élément à examiner.

Donc la comparaison de deux chaînes de longueur chacune 1000, peut nécessiter jusqu'à 1000 comparaisons. Ainsi, l'opérateur == entre deux chaînes peut être coûteux (en réalité, c'est moins simple que ça, cf. implicit Interning).

Accès aux caractères d'une chaîne

On peut accéder en lecture individuellement aux caractères d'une chaîne à l'aide d'un indice entier :

```
p = "Jupiter"

# le premier caractère de la chaîne p
print(p[0])

# le troisième caractère de la chaîne p
print(p[2])

# la longueur de la chaîne
n = len(p)
print(n)

# Dernier caractère de la chaîne p
print(p[n - 1])
```

```
J
p
7
r
```

Les caractères d'une chaînes sont numérotés de la gauche vers la droite. Ces numéros sont appelés des *indices* ; la numérotation commence à 0 (et non pas à 1) et se termine à `n-1` où `n` désigne le nombre total de caractères.

On accède à chaque caractère de la liste avec l'opérateur `[]` d'indexation.

Le fait de pouvoir accéder à tous les caractères d'une chaîne par des entiers consécutifs se traduit en disant que les chaînes de caractères sont du type *séquence*.

Il est possible d'utiliser des indices négatifs, ce qui est parfois à l'origine de bugs pour ceux qui ne connaissent pas leur existence.

Chaîne vs liste des caractères

On recontre parfois dans du code d'apprentis codeurs des « chaînes » définies par la liste de leurs caractères, cf. 1re ligne ci-dessous :

```
1 s = ['C', 'H', 'O', 'C', 'O', 'L', 'A', 'T']
2 print(len(s))
3 print(s[0])
```

```
4 8
5 C
```

Il se trouve que (lignes 4 et 5) la longueur de la chaîne est donnée correctement ainsi que le premier caractère. Une tournure ici équivalente serait d'utiliser une chaîne littérale :

```
s = "CHOCOLAT"
print(len(s))
print(s[0])
```

En pratique, on rencontre souvent la liste des lettres de l'alphabet :

```
alpha = ['A', 'B', 'C', 'D', etc]
```

Bien que la tournure ne soit pas forcément incorrecte (en particulier si on a besoin de modifier une chaîne), la plupart du temps ce n'est pas justifié. La saisie de la chaîne est beaucoup plus laborieuse que sous forme de chaîne littérale, la chaîne est moins facilement lisible, on n'a pas une chaîne mais une liste, donc on ne dispose pas des méthodes de chaînes, l'appartenance n'est pas définie de la même façon, par exemple :

```
s = ['C', 'H', 'O', 'C', 'O', 'L', 'A', 'T']
print("" in s)

s = "CHOCOLAT"
print("" in s)
```

```
False
True
```

Dépassement d'indice dans une chaîne

Comme pour les listes, un indice de chaîne trop grand entraîne la levée d'une exception :

```
1 z = "orange"
2 c = z[10]
3
4 print(c)
```

```
5     c = z[10]
6 IndexError: string index out of range
```

— Ligne 6 : une erreur d'indice est signalée

— Ligne 2 : la variable c désignerait le caractère à l'indice 10 alors que l'indice maximal dans c est 5 puisque la chaîne z est de longueur 6.

— Ligne 2 : L'opération z[10] est en fait interdite, on tente d'accéder en lecture à un caractère qui n'existe pas.

— Lignes 5-6 : Le message d'erreur explique que l'indice 10 est en dehors de la plage d'indices possibles.

Tenter d'accéder en simple lecture à un caractère d'une chaîne avec un indice ne correspondant pas à un élément de la chaîne conduit à une erreur, de type `indexError` et qualifiée de *débordement d'indice*. C'est le même problème que pour les listes.

Boucle for : parcours de chaînes

Une chaîne est une séquence et peut être parcourue, sans indice, par une boucle `for`. Voici un exemple de parcours d'une chaîne avec une boucle for :

```
for c in "ALIBABA":
    print(c*2)
```

```
AA
LL
II
BB
AA
BB
AA
```

C'est le même principe que pour les listes, il est possible de faire un parcours sans recourir à un indice.

Il est aussi possible de parcourir la chaîne par indices :

```
s="ALIBABA"

for i in range(len(s)):
    print(s[i]*2)
```

```
AA
LL
II
BB
AA
BB
AA
```

Cela peut être justifié dans certaines situations (par exemple si on doit comparer un caractère au suivant ou au précédent) où l'usage recommande d'utiliser `enumerate`, voir cette explication.

Comparaison alphabétique de chaînes

Les différents opérateurs de comparaison (par exemple < ou >=) permettent d'effectuer certaines comparaisons alphabétiques de chaînes :

```
print("prune" < "rose")
```

```
True
```

Si les chaînes ont la même casse (soit les deux en minuscule, soit les deux en majuscule) et sont dépourvues de caractères spéciaux ou d'accents, ces comparaisons correspondent à ce qui est attendu. En effet, l'opérateur de comparaison appliqué à des chaînes compare les codes unicodes des chaînes. Ces codes ne correspondent que partiellement à l'ordre alphabétique usuel. Pour des chaînes de casses différentes ou utilisant des caractères spéciaux, les comparaisons sont parfois contraires à l'ordre usuel :

```
print("prune" < "Rose")
print("été"< "printemps")
```

```
False
False
```

Pour comparer deux chaînes en tenant compte des accents, voir le paragraphe intitulé *Comparaison alphabétique de chaînes unicodes.*

La fonction `repr`

La notion exposée dans ce paragraphe n'est pas forcément adaptée à quelqu'un ayant peu d'expérience en Python.

La fonction `repr` est une fonction standard. Voici un exemple commenté d'utilisation de cette fonction :

```
1 X = "rose"
2 code = repr(X)
3 print(code)
4 print(X)
5 print(len(code))
```

```
6 'rose'
7 rose
8 6
```

— Ligne 1 : la fonction `repr` peut prendre en argument n'importe quel objet Python (un entier, une liste, une fonction, etc), ici une chaîne (qui est littérale ici mais ça n'a aucune importance) ;

— Ligne 2 : la variable `code` est ici une chaîne de caractères. Le contenu de cette chaîne est exactement un code Python d'un objet qui vaudrait `X`.

— Ligne 6 : on observera la présence d'apostrophes au début et en fin de chaîne, cela va être expliqué ci-dessous.

— Ligne 7 : par contraste, on observera l'absence de séparateur (apostrophe ou guillemet) dans l'affichage de `X`

— Ligne 8 : `code` n'est pas la chaîne *rose* ; cette dernière est de longueur 4 alors que `code` est une chaîne de longueur 6 : `code` est en fait une chaîne littérale, donc le premier caractère de `code` n'est pas le `r` de `rose` mais une apostrophe (délimiteur de chaîne littérale).

Ainsi, la fonction `repr` appliquée à un objet Python `X` (par exemple un entier, une chaîne, une liste, etc) renvoie une chaîne dont le contenu, s'il était placé dans du code Python, serait évalué en l'objet `X`. Le nom `repr` vient du fait que la fonction renvoie une *représentation* de l'objet sous forme de chaîne.

Pour dire les choses simplement, `repr` transforme des **objets** Python en du **code** Python.

Ne pas s'inquiéter si cette notion n'est pas totalement claire, elle peut nécessiter d'avoir pas mal travaillé avec du code Python pour bien être comprise.

La fonction `repr` ressemble beaucoup à la fonction `print` dans la mesure où, dans la plupart des cas, l'affichage de ce que renvoie `repr` est strictement identique à l'affichage produit par la fonction print :

```
print(repr(42))
print(42)
```

```
42
42
```

Il y a une nuance essentielle pourtant : la fonction `print` appliquée à un objet se contente de fournir une représentation uniquement visuelle d'un objet (la fonction `print` affiche à l'écran et ne retourne rien) tandis que la fonction `repr` renvoie une chaîne représentant l'objet.

La présentation ci-dessus est simplifiée. Il se peut très bien que `repr` ne renvoie pas une chaîne de code Python exécutable. Par exemple,

```
print(repr(print))
```

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

et les cas de ce type sont nombreux. La documentation officielle de la fonction : repr. La définition donnée n'est pas claire et la documentation n'indique pas quand la chaîne retournée est directement exécutable ou pas.

La classe `str`

Toutes les chaînes de caractères sont des instances de la classe `str` qui représente le type *string* en Python. La classe `str` peut être utilisée comme « constructeur » ie un appel de la forme `str(v)` où `v` est un objet Python retourne (construit) une chaîne qui permet d'afficher `v`. Par exemple :

```
1 s = str(6 * 7)
2
3 print(s)
4 print(s[0])
```

```
5 42
6 4
```

— Ligne 1 : le constructeur `str` est appelé sur l'entier 42
— Ligne 4 : `s` est l'interprétation par `str` de l'entier 42. Il s'agit de la **chaîne** `"42"`.
— Lignes 3 et 5 : `s[0]` n'est donc que le premier chiffre de 42 vu comme comme un **caractère**, à savoir le chiffre `"4"`.

C'est anecdotique mais un appel `str()` renvoie une chaîne vide.

La fonction `print` est fortement liée à la fonction `str` : en effet, la fonction `print` appliquée à un objet `X` affiche la chaîne renvoyée par `str(X)`.

Conversion d'un nombre en chaîne avec le type `str`

Dans certaines situations, un nombre `x`, par exemple entier ou flottant, a besoin d'être vu moins comme un nombre que comme la suite des chiffres décimaux qui composent `x`, chaque chiffre (décimal) étant alors vu comme un caractère. Le type natif `str` permet de convertir un nombre en chaîne.

Soient des questions telles que :

— trouver le nombre de chiffres d'un nombre, par exemple 458 a 3 chiffres
— calculer la somme des chiffres d'un nombre, par 458 a pour somme des chiffres 17
— récupérer le premier chiffre d'un nombre, par exemple 4523012 a pour premier chiffre 4
— changer un point par une virgule, par exemple **3.14 -> 3,14**

Pour ce genre de programme, il est préférable de voir les nombres comme chaînes de caractères.

Fonction `str` de conversion

Voici un exemple d'utilisation :

```
1 print(str(42))
2 print(str(3.14))
3 print(str(75 + 30))
```

```
4 42
5 3.14
6 105
```

— Ligne 1 `str` prend ici un entier en argument et renvoie sa conversion en caractères (en chiffres décimaux). En particulier, `str(42)` est la chaîne de caractères **42**

— Ligne 2 : le type natif `str` convertit également des flottants.

— Lignes 3 et 6 : noter que l'argument `75 + 30` est d'abord évalué en l'entier `105` avant d'être converti en la chaîne **105**.

Le type natif `str` s'utilise comme une fonction : on parlera donc de fonction `str` même s'il s'agit plutôt d'un type.

Signalons qu'effectuer la conversion d'un nombre vers sa représentation décimale nécessite un travail qui n'est pas complètement trivial même si l'idée est simple (faire des divisions entières successives par 10). Le code de l'implémentation en C se trouve ici, cf. la fonction `long_to_decimal_string_i`

La Référence du langage indique seulement depuis sa version 3.8 que la représentation se fait en base 10, si possible.

Utilisations de la fonction `str`

Voici quelques exemples d'utilisation de la fonction `str`.

Nombre de chiffres

Pour calculer le nombre de chiffres d'un nombre entier, on peut utiliser la fonction `str` :

```
1 x = 2038 * 2 ** 10
2 ch = str(x)
3 print(len(ch))
4 print(x)
```

```
5 7
6 2086912
```

— Ligne 2 : on convertit l'entier en chaîne et la fonction `len` compte le nombre de caractères de la chaîne, ce qui donne le nombre de chiffres (en base 10, toujours).

— Ligne 6 : en calculant explicitement `x`, on vérifie visuellement que `x` a bien 7 chiffres.

Changer le point décimal en virgule

Pour transformer par exemple le nombre `3.14` en sa représentation à virgule `3,14` on transforme d'abord le flottant `3.14` en chaîne puis on remplacera le point par une virgule dans la nouvelle chaîne :

```
x = 3.14
z = str(x)
zz = z.replace(".", ",")
print(zz)
```

```
3,14
```

— Noter que zz est une chaîne de caractères

Limitation de la conversion en chaîne

Il faut prendre garde qu'une fois un nombre x converti en chaîne s, il n'est plus possible d'effectuer directement des opérations *numériques* sur s car les chiffres sont en fait des caractères et non des entiers.

Par exemple, supposons que l'on cherche à afficher les chiffres pairs d'un entier ; si x = 53421 alors les chiffres pairs sont 4 et 2. Alors, on ne pourra pas tester la parité du chiffre avec l'opération % 2 :

```
x = 53421
s = str(x)
for c in s:
    if c % 2 == 0:
        print(c)
```

```
    if c % 2 == 0:
TypeError: not all arguments converted during string formatting
```

Pour y parvenir, il faudrait reconvertir les chiffres en entiers en utilisant int. On aurait le même type de problème s'il fallait calculer la somme des chiffres.

Conversion d'une chaîne en entier

Il ne faut pas confondre un nombre entier et la chaîne qui le représente. Par exemple, les objets Python "2042" et 2042 s'affichent de la même façon :

```
a = "2042"
print(a)
b = 2042
print(b)
```

```
2042
2042
```

mais ils sont bien distincts : le premier objet est de type chaîne de caractères et le second est de type entier.

C'est quoi le problème ?

Voici un exemple de situation où l'on a besoin de convertir une chaîne en nombre : soit un programme qui, à partir d'une date donnée sous forme de chaîne de caractères et sous un certain format, par exemple 31/12/2042, doit retourner la date du lendemain, ici 01/01/2043 (sous forme de chaîne, toujours). Pour résoudre le problème, il faut extraire de la date les éléments suivants :

— le jour (ici 31),
— le mois (ici 12),
— l'année (ici 2042)

qui sont des **sous-chaînes** et effectuer des opérations sur ces objets vus comme des **entiers**, par exemple une addition, ici $2042 + 1 = 2043$, d'où la nécessité de convertir la chaîne en nombre entier, car la somme des chaînes `"2042" +"1"` ne donne pas le résultat attendu :

```
print("2042" + "1")
```

```
20421
```

La fonction de conversion `int`

Pour convertir une chaîne représentant un entier écrit en base 10 en l'entier correspondant, on utilise le type intégré `int` :

```
1 a = "2042"
2 print(int(a) + 1)
```

```
3 2043
```

— Ligne 2 : `int(a)` vaut l'entier représenté en base 10 par la chaîne "2042" c'est-à-dire le nombre `2042`. Il est donc licite d'ajouter `1` à `int("2042")`.

`int` désigne le type `int` (c'est-à-dire le type entier) mais peut aussi être utilisé comme une fonction. On parlera donc de fonction `int` même s'il s'agit plutôt d'un type.
Remarquer que l'opération suivante :

```
"2042" + 1
```

n'a pas de sens en Python :

```
a = "2042"
print(a + 1)
```

```
    print(a + 1)
TypeError: Can't convert 'int' object to str implicitly
```

Précautions

On prendra garde qu'une fois une chaîne `s` convertie en nombre entier `x`, les opérations de chaînes sur `x` ne sont plus permises (elles n'ont pas de sens). Par exemple, si un nombre `x` est écrit sous forme de chaîne `s` alors `s[0]` représente le premier chiffre de `x`, un chiffre étant ici vu comme un caractère. En revanche, l'opération `x[0]` n'a pas de sens :

```
s = "3000"
x = int(s)
print(s[0])
print(x[0])
```

```
    print(x[0])
TypeError: 'int' object is not subscriptable
```

Le code de l'implémentation en C de la conversion semble se trouver ici, cf. la fonction `PyLong_FromString`. La fonction est particulièrement complexe (mais elle convertit aussi depuis des bases autres que 10).

Chaînes autorisées

Pour qu'un appel `str(s)` convertisse la chaîne `s` en entier encore faut-il que la chaîne fournie représente un entier :

```
x = int("3.14")
print(x)
```

```
    x = int("3.14")
ValueError: invalid literal for int() with base 10: '3.14'
```

La documentation indique que la chaîne doit être un entier littéral, ce qui d'ailleurs, n'est pas tout à fait vrai :

```
x = int("0042")
print(x)
```

```
42
```

alors que `0042` n'est pas un littéral entier valide :

```
x = 0042
print(x)
```

```
    x = 0042
            ^
SyntaxError: invalid token
```

Conversion d'une chaîne en flottant

On peut avoir besoin de convertir une chaîne représentant un nombre décimal « à virgule » et dont on a besoin de connaître la valeur en tant que **nombre**, par exemple pour faire une opération. Typiquement, on a du texte comme ci-dessous

```
Dollar:0.83€
Livre:1.16€
Rouble:0.011€
```

donnant, entre autres, le cours du dollar en euro et qu'on cherche à savoir combien d'euros coûtent 42 dollars. Pour cela, on va capturer, par un moyen ou par un autre et que je ne détaille pas ici, la chaîne `"0.83"`. Une fois ceci fait, voici comment on en récupère la valeur numérique :

```
1 taux="0.83"
2 valeur = float(taux)
3 n=42
4 print(n, "dollars valent", n*valeur, "euros")
```

```
5 42 dollars valent 34.86 euros
```

— Ligne 1 : `taux` est une chaîne

— Ligne 2 : `valeur` est la valeur flottante du nombre que la chaîne `taux` représente.
— Lignes 4 et 5 : dès lors que `valeur` est un nombre, on peut faire avec ce nombre des opérations numériques, par exemple un produit ce qui permet d'effectuer le calcul du change de 42 €.

Valeur d'un entier donné en base autre que 10

On a vu que `int` peut convertir une chaîne de chiffres décimaux en l'entier correspondant :

```
1 x = 10 * int("42") + 1
2 print(x)
```

```
3 421
```

En réalité, il est possible de convertir un entier représenté en n'importe quelle base allant de 2 à 36, il suffit de le préciser dans l'appel à `int` :

```
1 x = int("101010", base=2)
2 print(x)
3
4 x = int("AC", base=13)
5 print(x)
```

```
6 42
7 142
```

Différence entre les fonctions `str` et `repr`

Les deux fonctions standard `str` et `repr` s'appliquent à n'importe quel objet Python et renvoient toutes les deux une chaîne qui est censée représenter l'objet en question. Un appel de `str` sur un objet `x` renvoie une chaîne de caractères qui est censée interpréter `x` de façon *informelle* comme dit la documentation officielle. D'un autre côté, `repr(x)` renvoie une représentation qui est, en principe, une expression Python valide et qui s'évalue en l'objet `x`.

Les objets Fraction montrent bien la différence entre les deux. Python dispose d'un module standard appelé `fractions` qui permet d'utiliser des fractions de nombres entiers. Une fraction est définie par une classe Fraction, par exemple la fraction $\frac{22}{7}$ est définie de la manière suivante :

```
from fractions import Fraction

f = Fraction(22, 7)
```

Pour afficher cette fraction, il serait naturel de lire `22/7` et c'est justement ainsi que cet affichage est codé en interne dans le module fractions :

```
from fractions import Fraction

f = Fraction(22, 7)
print(f)
```

```
22/7
```

Par construction, `print(f)` affiche exactement la chaîne `str(f)` et donc ici, cette chaîne se traduit par la chaîne littérale `"22/7"`.

Maintenant, que vaut `repr(f)` ? Cela ne peut pas être la même chose car l'expression `22/7` n'est pas égale (en Python) à la fraction $\frac{22}{7}$ puisque `22/7` est un nombre flottant que Python ne connaît en plus qu'approximativement :

```
from fractions import Fraction

f = Fraction(22, 7)
print(f == 22/7)
```

```
False
```

Regardons ce que vaut `repr(f)` :

```
from fractions import Fraction

f = Fraction(22, 7)
print(repr(f))
```

```
Fraction(22, 7)
```

On voit que `repr(f) == s` où `s := 'Fraction(22, 7)'`. Et en effet, la valeur de `f` est le résultat de l'évaluation de l'expression entre les apostrophes, à savoir **Fraction(22, 7)**.

Un autre exemple qui peut aider à faire comprendre la nuance est de considérer comment sont affichés les tableaux sous Numpy (qui est une bibliothèque Python massivement utilisée en apprentissage automatique). Observer le code suivant :

```
import numpy as np

z=np.zeros(5)

print(z)
print(repr(z))
```

```
[0. 0. 0. 0. 0.]
array([0., 0., 0., 0., 0.])
```

— Lignes 5 et 7 : un tableau Numpy est affiché en donnant la liste de ses éléments, sans virgule séparatrice.

— Lignes 6 et 8 : `repr(z)` affiche du code Python exécutable et qui renvoie un objet ayant même valeur que `z` :

```
import numpy as np
from numpy import array

z=np.zeros(5)
zz = array([0., 0., 0., 0., 0.])

print(np.array_equal(z, zz))
```

```
True
```

Modifier la terminaison d'affichage par défaut

Par défaut, toute instruction d'affichage avec la fonction `print` se termine par un saut de ligne. La fonction `print` possède une « option » modifiant ce comportement par défaut en faisant en sorte que l'affichage soit terminé par une certaine suite de caractères décidée par le programmeur. Cette option consiste à passer un argument à la fonction `print`, nommé `end` et que le programmeur affecte au type de terminaison choisie.

Par exemple, supposons que l'on souhaite qu'un affichage soit terminé par deux points d'interrogation :

```
1 print("lundi", "mardi", end = "??")
```

```
2 lundi mardi??
```

— Ligne 1 : `print` a reçu trois arguments. Le dernier est un argument dit *nommé* : ici, *c'est comme si* on affectait une variable interne à la fonction `print`, la variable `end`. Le fait d'affecter `end` dans l'appel de `print` modifie la terminaison d'affichage : les chaînes **lundi** et **mardi** sont affichées, séparées par un espace (comme d'habitude) et, **immédiatement** après, le contenu de la chaîne `end` est affiché. On notera l'absence d'espace entre `mardi` et `??`.

— Les deux points d'interrogation ont été placés à la fin de l'affichage grâce à l'option `end` fournie à `print`. Toutefois, (rare) l'affichage dans certaines consoles pourra se comporter légèrement différemment.

Attention au comportement suivant :

```
1 print("lundi", "mardi", end = "??")
2
3 print("mercredi")
4 print("jeudi")
```

```
5 lundi mardi??mercredi
6 jeudi
```

— Ligne 1 : comme `end` n'a plus sa valeur par défaut, l'affichage n'est pas terminé par un saut de ligne

— Ligne 3 : l'affichage est effectué **immédiatement** après l'affichage provoqué par la ligne 1, en particulier sans rajout de saut de ligne au précédent affichage. En revanche, à la fin de l'affichage *mercredi*, `print` rajoute un saut de ligne (c'est le comportement par défaut de `print`).

— Ligne 4 : le mot est placé seul sur la ligne à cause du saut de ligne par défaut placé par le `print` à la ligne 3.

Modifier le séparateur par défaut de `print`

Par défaut, lorsque `print` reçoit plusieurs arguments, les affichages correspondants sont séparés par un espace et un seul. On peut modifier ce comportement par défaut en utilisant un argument nommé `sep` et lui affecter le séparateur que l'on souhaite, par exemple `"..."` si on veut séparer chaque affichage par trois points successifs :

```
1 print("lundi", "mardi", "mercredi", sep = "...")
```

```
2 lundi...mardi...mercredi
```

— Ligne 1 : l'argument nommé sep doit apparaître après la liste des chaînes à afficher.

L'affichage par défaut correspond à l'argument nommé `sep = " "` autrement dit `print("lundi", "mardi")` et `print("lundi", "mardi", sep = " ")` sont équivalents :

```
print("lundi", "mardi")
print("lundi", "mardi", sep = " ")
```

```
lundi mardi
lundi mardi
```

Il est possible de cumuler la modification du séparateurs sep et de la terminaison de ligne end.

```
print("lundi", "mercredi", sep = "...", end = "???????")
m = "mardi"
print("lundi", m, "mercredi", sep = "...")
```

```
lundi...mercredi???????lundi...mardi...mercredi
```

Afficher plusieurs objets sans séparation

On souhaite afficher des objets, sans qu'il n'y ait d'espace entre eux à l'affichage, par exemple :

```
LundiMardiMercredi
```

Il y a deux façons d'y parvenir avec la fonction `print`. Soit on modifie le séparateur par défaut, sep, de la fonction `print` en le plaçant à une chaîne vide, soit on fait de même mais avec le séparateur end de terminaison de ligne :

```
print("Lundi", "Mardi", "Mercredi", sep="")

print("Lundi", end="")
print("Mardi", end="")
print("Mercredi")
```

```
LundiMardiMercredi
LundiMardiMercredi
```

Affichage parfait sur une même ligne et boucle `for`

Ce paragraphe s'intéresse à une question très marginale. À examiner en seconde lecture.

Soit à réaliser avec une boucle `for` l'affichage **exact** suivant :

```
0 1 2 3 4 5 6 7 8 9
```

Un espace doit figurer **uniquement** entre deux chiffres successifs mais pas en fin de ligne.

Le code suivant semble réaliser l'affichage demandé :

```
for i in range(10):
    print(i, end = ' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Même si ce n'est pas visible, l'affichage n'est pas parfaitement conforme puisque lorsque `i` vaut `9` dans la boucle `for`, `9` est affiché **suivi d'un espace** alors qu'un espace est censé séparer un entier de *son suivant* dans la ligne (9 est le dernier entier de la ligne). On peut d'ailleurs le voir si on ajoute un affichage arbitraire :

```
for i in range(10):
    print(i, end = ' ')
print("XXXXX")
```

```
0 1 2 3 4 5 6 7 8 9 XXXXX
```

on voit qu'avant le premier `X` affiché, il y a un espace.

Pour obtenir un affichage parfaitement conforme, il faudrait interrompre la boucle juste avant la dernière étape puis afficher le dernier élément, non suivi d'un espace :

```
for i in range(9):
    print(i, end = ' ')
print(9)
print("XXXXXX")
```

```
0 1 2 3 4 5 6 7 8 9
XXXXXX
```

Une autre solution est possible, voire préférable serait d'utiliser la méthode `join` :

```
print(" ".join(str(i) for i in range(10)))
print("XXXXXX")
```

```
0 1 2 3 4 5 6 7 8 9
XXXXXX
```

Afficher sur la sortie d'erreurs standard

Quand on affiche un message, il apparaît, par défaut, sur ce qu'on appelle la sortie standard, désignée en Python par `stdout`. En pratique, la sortie standard est une zone de texte spécifique : une console par exemple, l'output d'une cellule Jupyter, etc. Mais parfois, on ne veut pas, que certains messages apparaissent (par exemple des diagnostics, des erreurs d'exécution) et dans ce cas, il existe une possibilité de *rediriger* les messages de ce type vers une autre « zone » qu'on appelle l'erreur standard désignée sous le terme de `stderr`.

Pour accéder à l'erreur standard, on utilise le module `sys` :

```
1 import sys
2
3 print(42)
4 print(2038, file=sys.stderr)
```

```
5 42
6 2038
```

On constate que les deux messages sont visibles dans la console alors que le 2e était pourtant dirigé vers l'erreur standard. C'est que, par défaut, les affichages sur la sortie standard et l'erreur

standard, sont confondus.

L'usage de stderr est intéressant en cas de redirection, en permettant de distinguer les messages d'erreurs de la sortie « normale » :

main.py

```
1 import sys
2
3 print(42)
4 print(2038, file=sys.stderr)
```

```
5 $ python3.4 main.py > mon_fichier.txt
6 2038
```

et dans ce cas, le contenu de `mon_fichier.txt` est :

```
1 42
```

Afficher sous Jupyter Notebook ou dans une console Python

Comment est réalisé l'affichage de calculs dans une feuille Jupyter Notebook ou une console Python ? Dans une feuille Jupyter Notebook, soit la cellule de code suivante, suivie de l'affichage produit dans une cellule d'*output* par l'exécution de la cellule :

```
1 1 + 1
2 print(2 + 2)
3 3 + 3
```

```
4 4
5 6
```

— Ligne 1 : la valeur de l'expression n'est pas affichée dans la cellule d'*output*, faute d'appel à la fonction `print`

— Lignes 2 et 4 : puisque la fonction `print` est utilisée, un affichage est visible.

— Ligne 3 et 5 : malgré l'absence de `print`, il y a eu un affichage : en effet la dernière instruction de la cellule est une *expression* (`3 + 3`) et dans ce cas, sa valeur est affichée.

En revanche, la cellule suivante

```
1 + 1
3 + 3
x = 42
```

ne produira aucun affichage. En effet, la dernière instruction n'est pas une expression.

Il reste le cas particulier où l'expression en dernière ligne vaut `None` : dans ce cas, rien n'est affiché :

```
1 L = [40, 41]
2 print("append : ", L.append(42))
3 L.append(43)
```

```
4 append :  None
```

— L'affichage visible provient de la ligne 2 et de l'appel à la fonction `print` : l'appel `L.append(42))` vaut `None` et c'est ce qui est affiché

— Comme indiqué précedemment, l'expression `L.append(43)` vaut toujours `None`. Pourtant, cette expression, bien que dernière dans la cellule, n'affiche pas `None`. C'est normal, c'est l'effet recherché par le fonctionnement d'une cellule Jupyter Notebook.

Par ailleurs, l'affichage que peut produire l'expression, disons `expr`, en dernière ligne de la cellule, affiche en fait `repr(expr)` et non pas l'affichage habituel. On le voit bien avec une chaîne de caractères :

```
1 c = "Orange"
2
3 print(c)
4 print(repr(c))
5
6 "Orange"
```

```
7 Orange
8 'Orange'
9 'Orange'
```

La dernière ligne (ligne 9) provenant de l'expression ligne 6 montre le mot entouré d'apostrophes : c'est exactement l'affichage de `repr(c)`, cf. lignes 4 et 8.

Le comportement de l'affichage lorsque Python est utilisé en mode interactif (une « console avec un prompt Python ») est très semblable :

```
1 >>> print("Orange")
2 Orange
3 >>> "Orange"
4 'Orange'
5 >>>
```

Décompactage des arguments de la fonction `print`

Le décompactage des arguments appliqué à la fonction `print` permet d'obtenir assez simplement des effets d'affichage (le décompactage est le résultat de l'usage de la syntaxe `*` devant un itérable, voir ci-dessous l'exemple).

Par exemple, on dispose d'une liste `L` d'entiers et on veut afficher ces entiers juste séparés par un espace :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]
print(*L)
```

```
65 31 9 32 81 82 46 12
```

En effet, par définition même de la syntaxe de décompactage, l'appel `print(*L)` est équivalent à :

```
print(65, 31, 9, 32, 81, 82, 46, 12)
```

et on sait que, dans ce cas (le cas par défaut), les arguments de la fonctions `print` sont séparés par un espace.

En utilisant les argument nommés de la fonction `print`, l'affichage peut être enrichi :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]
print(*L, sep=', ')
```

```
65, 31, 9, 32, 81, 82, 46, 12
```

La décompactage des arguments s'applique à tout itérable. Par exemple, aux chaînes de caractères, ce qui permet d'afficher facilement les lettres d'un mot :

```
print(*"ORANGE", sep='.')
```

```
O.R.A.N.G.E
```

Il s'applique aussi aux expressions génératrices, par exemple :

```
print(*(k for k in range(20) if k%3==0 and k%5!=0))
```

```
3 6 9 12 18
```

Vider le buffer

La sortie de `print` est bufférisée, c'est-à-dire que le contenu de l'affichage est placé dans une mémoire-tampon et lorsque la mémoire-tampon est pleine ou lorsque les données sont construites, le buffer est vidé et les données sont envoyées dans la zone d'affichage, habituellement la sortie standard (stdout).

La présence de certains caractères vide le buffer, par exemple le caractère saut de ligne (`'\n'`) à pour effet de vider le tampon ce qui provoque l'affichage.

Si une chaîne de caractères doit être affiché et qu'elle ne contient pas de saut de ligne, il peut y avoir un délai d'attente entre l'instruction `print` et l'affichage réel, comme le montre le code ci-dessous :

```
1 import time
2
3 for c in "Python"*3:
4     print(c, end='...')
5     time.sleep(0.5)
```

```
6 P...y...t...h...o...n...P...y...t...h...o...n...P...y...t...h...o...n...
```

Chaque appel à `sleep` temporise l'exécution pendant 1/2 seconde. Dans ce code, l'affichage (ligne 6) n'apparaît qu'au bout de 9 secondes, et il apparaît d'un seul coup. En effet, l'appel à la fonction `print` ne contient pas de saut de ligne (puisque `end` vaut autre chose, cf. ligne 4), donc le buffer est vidé lorsque le programme se termine.

Pour provoquer le vidage du buffer, on peut utiliser l'option `flush=True` de la fonction `print` :

```
1 import time
2
3 for c in "Python"*3:
4     print(c, end='...', flush=True)
5     time.sleep(0.5)
```

```
6 P...y...t...h...o...n...P...y...t...h...o...n...P...y...t...h...o...n...
```

Cette fois l'affichage apparaît progressivement, chaque caractère attend 1/2 seconde avant d'être

affiché.

Chaîne analogue à un fichier

On aimerait parfois qu'un code qui capture des données dans un fichier texte soit encore applicable à un code où les données seraient dans une chaîne au lieu d'être dans un fichier.

Par exemple, soit le fichier `planetes.txt`

```
Mars
Jupiter
Uranus
Neptune
```

et le code suivant qui affiche le contenu du fichier `planetes.txt` :

```
1 fichier = open("planetes.txt")
2 contenu = fichier.read()
3 print(contenu)
```

Le contenu du fichier `planetes.txt` est représenté par la chaîne `"Mars\nJupiter\nUranus\nNeptune"`. On souhaiterait que la partie suivante du code précédent :

```
1 contenu = fichier.read()
2 print(contenu)
```

soit encore applicable si on dispose d'une **chaîne** au lieu du fichier `planetes.txt`. La solution est donnée par la fonction `io.StringIO` du module standard `io` :

```
1 import io
2
3 s = "Mars\nJupiter\nUranus\nNeptune"
4
5 fichier = io.StringIO(s)
6 contenu = fichier.read()
7 print(contenu)
```

```
8 Mars
9 Jupiter
10 Uranus
11 Neptune
```

— Ligne 5 : la fonction `io.StringIO` renvoie un objet de type `io.StringIO` et qui admet les méthodes de fichier.

— Ligne 6 : on peut lire `fichier` (qui **n'est pas** un objet de type *file*) **comme** un objet de type *file*, en particulier avec la méthode **read**.

Le cas de l'entrée standard

De la même façon, on peut émuler l'entrée standard par une chaîne. Par exemple, voici un code qui effectue la somme de deux entiers donnés sur l'entrée standard :

somme_readlines.py

```
1 import sys
2 stdin = sys.stdin
3 L = stdin.readlines()
4 x = L[0]
5 y = L[1]
6 print(int(x)+int(y))
```

```
7 42
8 10
9 52
```

Pour transmettre les nombres :

— soit on appuie sur la touche Entrée après chaque nombre et, à la fin, on transmet via le clavier le caractère de fin de fichier EOF (`Ctrl + D` sous Linux)

— (le plus commode) soit on place les deux nombres dans un fichier `nombres.txt` (un nombre par ligne et sans saut de ligne final) et le code Python (`somme_readlines.py`) capture les deux entiers par redirection en lançant en ligne de commande :

```
$ python3 somme.py < nombres.txt
```

Voici maintenant un code qui conserve l'essentiel de `somme_readlines.py` mais en lisant les entrées dans une chaîne :

```
1 import io
2
3 s = "42\n10\n"
4 stdin = io.StringIO(s)
5
6 L = stdin.readlines()
7 x = L[0]
8 y = L[1]
9 print(int(x)+int(y))
```

```
10 52
```

— Lignes 6-9 : la partie lecture et calcul du code de `somme_readlines.py` est préservée.

Afficher « dans » une chaîne

La sortie standard (là où on affiche communément) est un fichier comme un autre. On y a accède via `sys.stdout` et ce fichier possède une méthode `write` permettant d'afficher (c'est plus ou moins équivalent à `print`). Voici un exemple :

```
import sys
cible = sys.stdout

message1 = "Mars\nJupiter\n"
cible.write(message1)

message2 = "Uranus\nNeptune\n"
```

```
cible.write(message2)
```

qui produit, en principe dans une console ou une zone de sortie textuelle, l'affichage suivant

```
1 Mars
2 Jupiter
3 Uranus
4 Neptune
```

On peut parfois souhaiter que tous les affichages qu'un code effectue soient envoyés dans une **chaîne** au lieu de la sortie standard sans pour autant modifier profondément le code. C'est possible en utilisant le module standard `stringIO`. Voici un exemple équivalent au précédent :

```
1 import io
2 cible = io.StringIO()
3
4 message1 = "Mars\nJupiter\n"
5 cible.write(message1)
6
7 message2 = "Uranus\nNeptune\n"
8 cible.write(message2)
9
10 print(cible.getvalue())
```

```
11 Mars
12 Jupiter
13 Uranus
14 Neptune
```

— Ligne 2 : la chaîne dans laquelle les messages ultérieurs vont être « affichés ».

— Lignes 5 et 8 : envoie des messages dans la chaîne avec la méthode `write`.

— Ligne 10 : pour récupérer le contenu de la chaîne de type `StringIO`, on appelle sa méthode `getvalue` ce qui renvoie une chaîne habituelle et que l'on peut afficher.

— Ligne 11-14 : on récupère bien le contenu envoyé dans la chaîne et comme avec le code précédent.

Utilisation de la fonction `print`

Il est également possible d'envoyer l'affichage dans une chaîne `StringIO` au moyen de la fonction `print`. En effet, la fonction `print` dispose d'un argument nommé `file` qui par défaut pointe vers la sortie standard. On peut passer une autre valeur, par exemple une chaîne `StringIO`, exactement comme à l'exemple précédent. On peut en plus utiliser les autres arguments nommés `sep` ou `end` pour formater l'affichage. Voici un exemple :

```
1 import io
2 cible = io.StringIO()
3
4 print("Mars", "Jupiter", "Uranus", "Neptune", sep='\n', end='', file=cible)
```

Ce code n'affiche rien car l'affichage a été détournée vers la chaîne cible. Dans `cible`, les objets à afficher sont séparés par des sauts de ligne, cf. `sep='\n'`. Noter que pour éviter un saut de ligne non désiré, la fin de ligne par défaut (qui est un saut de ligne) a été écrasée par `end=''`.

Il est bien sûr possible d'afficher le contenu de `cible` en récupérant son contenu via la méthode `getvalue` :

```
import io
cible = io.StringIO()

print("Mars", "Jupiter", "Uranus", "Neptune", sep='\n', end='', file=cible)

print(cible.getvalue())
```

```
Mars
Jupiter
Uranus
Neptune
```

Chapitre II

Sous-chaînes, transformations

Modifier une chaîne

Les caractères d'une chaîne ne sont pas modifiables :

```
1 s = "taTou"
2 s[2] = "B"
3 print(s)
```

```
4 TypeError: 'str' object does not support item assignment
```

— Ligne 2 : on tente de changer le caractère `T` de la chaîne `"taTou"` en `"B"`.

Par construction, les chaînes de caractères sont de type *immuable* : leur contenu ne peut être modifié. Non seulement, on ne peut pas changer un caractère en un autre, mais on ne peut en supprimer ni en rajouter, à quelque endroit que ce soit.

Affectation augmentée par addition

Considérons la réaffection de variable suivante :

```
1 a = "Rose"
2 b = "Orange"
3 a = a + b
4
5 print(a)
```

```
6 RoseOrange
```

La variable `a` a été réaffectée à elle-même après modification. La ligne 3 peut être raccourcie en Python avec une instruction d'affectation augmentée, notée `+=`, ce qui donne le nouveau code suivant :

```
1 a = "Rose"
2 b = "Orange"
3 a += b
4
5 print(a)
```

```
6 RoseOrange
```

La documentation de référence du langage Python n'explique pas précisément comment la somme `a + b` (ligne 3 du premier code) est réalisée : des copies de `a` et de `b` sont-elles réalisées avant que la somme ne soit réalisée ? Cette question est importante car elle va expliquer comment une somme telle `a + b + c` va être réalisée.

Toujours-est-il que l'affectation augmentée n'est pas qu'un raccourci syntaxique, l'opération est « en place » : elle éviterait une copie de `a` et serait plus efficace.

Des exemples seront montrés ultérieurement.

Créer une chaîne à partir de la chaîne vide

Soit à construire une chaîne `z` dont les caractères sont exactement les consonnes d'une chaîne `s` donnée. Par exemple, si `s` est la chaîne `broccoli` alors la chaîne cherchée est `brccl`.

Pour cela, il suffit de parcourir la chaîne `s` et d'ajouter à une chaîne initialement vide (disons `z`) le caractère courant de `s` si le caractère est une consonne (et donc s'il n'est pas une voyelle). D'où le code

```
1 s= "broccoli"
2 VOYELLES ="aeiouy"
3 z = ""
4
5 for c in s:
6     if c not in VOYELLES:
7         z += c
8
9 print(s)
10 print(z)
```

```
11 broccoli
12 brccl
```

— Lignes 3, 10 et 12 : la chaîne à construire est initialement vide.

— Lignes 5-7 : la chaîne `s` est agrandi au fur et à mesure par ajout d'une consonne.

C'est le même type de construction que pour une liste initialement vide à laquelle on applique la méthode append.

Extraire les caractères individuels d'une chaîne

On se donne une chaîne de caractères `s`, par exemple `s = "abricot"` et on cherche à en extraire les caractères successifs pour les placer dans une liste. Pour cela, il suffit d'appliquer le constructeur `list` à `s` :

```
s = "abricot"
L = list(s)
print(L)
```

```
['a', 'b', 'r', 'i', 'c', 'o', 't']
```

La méthode `join`

On dispose d'une liste de chaînes, par exemple les 4 chaînes suivantes :

```
File Open Options Display
```

et on veut rassembler ces chaînes en une seule mais en les séparant avec un séparateur donné, par exemple avec le séparateur

```
 > 
```

(le symbole > entouré d'espaces), ce qui donnerait :

```
File > Open > Options > Display
```

C'est ce que permet de faire la méthode `join` :

```
1 L = ["File", "Open", "Options", "Display"]
2 s = " > ".join(L)
3 print(s)
```

```
4 File > Open > Options > Display
```

— Ligne 1 : les chaînes à rassembler sont placées dans une liste.

— Ligne 2 : noter que la méthode s'utilise comme attribut du séparateur et qu'elle prend en argument la liste.

— Ligne 4 : on observe bien le rassemblement des chaînes de la liste `L`.

L'usage de la méthode `join` n'est pas toujours considéré comme intuitif (voir ici par exemple), mais `join` est une méthode du **séparateur** (d'une chaîne donc), et pas d'une liste.

Séparateurs courants pour la méthode `join`

Quand on applique `sep.join(L)`, tout séparateur `sep` peut être envisagé mais en voici trois très courants :

— la chaîne vide `""`

— le saut de ligne `"\n"`

— l'espace `" "`

Par exemple

```
1 L = ["Mars", "Jupiter", "Uranus", "Neptune"]
2
3 s = "".join(L)
4 print(s)
5 print()
6
7 s = "\n".join(L)
8 print(s)
9 print()
10
```

```
11 s = " ".join(L)
12 print(s)
```

qui affiche

```
1 MarsJupiterUranusNeptune
2
3 Mars
4 Jupiter
5 Uranus
6 Neptune
7
8 Mars Jupiter Uranus Neptune
```

— Ligne 1 : C'est une technique **idiomatique** (ligne 3 du code) de **concaténation** de chaînes où on utilise `''.join(L)` avec un séparateur **vide**.

— Lignes 3-6 : les chaînes sont placées les unes en-dessous des autres avec le séparateur *saut de ligne* (le caractère `\n`).

— Ligne 9 : les chaînes sont placées côte-à-côte, séparées par exactement un espace et uniquement entre les chaînes, pas à la fin ni au début.

Séparer une chaîne suivant des sous-chaînes avec la méthode `split`

On dispose d'un numéro de téléphone, par exemple, `08-42-20-32-17` et on cherche à extraire les composantes de ce numéro de téléphone, à savoir 08, 42, etc. Autrement dit, il s'agit de séparer le numéro suivant le caractère `-`. C'est typiquement ce que peut faire la méthode `split` :

```
numero = "08-42-20-32-17"
L = numero.split("-")
print(L)
print(numero)
```

```
['08', '42', '20', '32', '17']
08-42-20-32-17
```

`split` est une méthode de chaîne. Un appel `numero.split(sep)` a pour fonction de séparer la chaîne `numero` suivant le séparateur `sep`, lui aussi une chaîne. Dans l'exemple ci-dessus, `sep` est la chaîne `"-"`. L'appel à la méthode `split` renvoie la liste des chaînes qui sont séparées par le séparateur, chacune des chaînes étant une sous-chaîne de la chaîne initiale. La chaîne initiale n'est pas modifiée, cf. la dernière ligne de la sortie ci-dessus (de toute façon, une chaîne n'est pas modifiable).

Noter que `s.split(sep)` renvoie une liste de chaînes qui ne contiennent plus `sep` comme sous chaîne.

Le séparateur peut être une chaîne arbitraire (mais non vide), éventuellement contenant plusieurs caractères, comme ci-dessous :

```
s = "marabout -> bout de ficelle -> selle de cheval -> cheval de course"
L = s.split(" -> ")
print(L)
```

```
['marabout', 'bout de ficelle', 'selle de cheval', 'cheval de course']
```

Situation ambiguë

Il se peut que le séparateur empiète sur lui-même, par exemple si `s = vvvABABAvvv` et le séparateur est `ABA` alors l'occurrence du séparateur qui commence en 4e position n'est pas disjointe de l'occurrence qui commence en 6e position. Voici comment `split` réalise le découpage :

```
s="vvvABABAvvv"

print(s.split("ABA"))
```

```
['vvv', 'BAvvv']
```

On voit donc que le découpage se fait *de la gauche vers la droite*, chaque sous-chaîne valant le séparateur est écartée et la recherche du séparateur continue à l'indice qui suit la fin du séparateur.

Règle

L'appel `ma_chaine.split(mon_sep)` élimine `sep` de `s`, en progressant dans `s` de la gauche vers la droite. Les tronçons de `s` restants forment la liste `L` renvoyée par `split` et cette liste `L` de chaînes est telle que si on intercale entre deux éléments consécutif de `L` la chaîne `mon_sep`, on obtient une chaîne identique à `ma_chaine`. Si la chaîne `ma_chaine` contient n occurrences successives et disjointes de `mon_sep` alors la liste `L` contiendra $n + 1$ éléments. Le séparateur doit être une chaîne non vide.

L'option `maxsplit` de la méthode `split`

La méthode `split` dispose d'une option `maxsplit` qui limite le nombre de retraits du séparateur. Voici un exemple :

```
1 numero = "08-42-20-32-17"
2 L = numero.split("-", maxsplit=2)
3 print(L)
```

```
4 ['08', '42', '20-32-17']
```

Le procédé est le même que pour `split` sans cette option : la chaîne est parcourue de la gauche vers la droite et sont rétirées de la chaîne au plus les `maxplit` premières occurrences du séparateur.

Méthode `split` : blancs et saut de ligne

Si on ne donne aucun argument à la méthode `split`, la séparation est effectuée suivant les blancs :

```
s = "L'eau\nLa terre\nLe feu"
L = s.split()
print(L)
```

```
["L'eau", 'La', 'terre', 'Le', 'feu']
```

— La chaîne `s` est découpée suivant les blancs (espace et saut de ligne).

On notera que tout groupement **contigu** de blancs fait alors office de séparateur unique :

```
s = "L’     eau     \n   La \n terre    \t  \n\nLe      feu"
print(s)
L = s.split()
print(L)
```

```
L’     eau
   La
 terre

Le      feu
["L’", ’eau’, ’La’, ’terre’, ’Le’, ’feu’]
```

Ci-dessus, les blancs sont constitués d’espaces, de sauts de ligne et de tabulations horizontales.

Un usage courant de `split` est de découper un texte en une suite de lignes avec le séparateur `"\n"` :

```
s = "L’eau\nLa terre\nLe feu"
L = s.split(’\n’)
print(L)
```

```
["L’eau", ’La terre’, ’Le feu’]
```

Méthode `splitlines` : découper en lignes successives

On dispose d’un texte, avec des sauts de lignes, par exemple

```
--------
L’eau

La terre

Le feu

Le ciel

--------
```

(noter qu’il n’y a pas de saut de ligne final après la 2e ligne de tirets) et on souhaite le découper en une liste `L` de chaînes, chaque élément de `L` étant formé d’une ligne (contenant au plus un saut de ligne). La méthode `split` ne convient pas car elle perd les sauts de lignes et, même si on rajoute le saut de ligne, cela ne convient pas toujours. C’est pour cela qu’existe la méthode `splitlines` :

```
1 s="""--------
2 L’eau
3
4 La terre
5
6 Le feu
```

```
7
8 Le ciel
9
10 --------"""
11 L=s.splitlines(True)
12 print(L)
13
14 L=s.splitlines()
15 print(L)
16
17 M=s.split('\n')
18 print(L == M)
```

```
19 ['--------\n', "L'eau\n", '\n', 'La terre\n', '\n',
20         'Le feu\n', '\n', 'Le ciel\n', '\n', '--------']
21 ['--------', "L'eau", '', 'La terre', '',
22         'Le feu', '', 'Le ciel', '', '--------']
23 True
```

— Lignes 1-10 : pour alléger la saisie, j'ai utilisé une chaîne littérale triple, voir Les différents délimiteurs de chaînes littérales.

— Ligne 11 : la méthode `splitlines` possède un argument nommé `keepends`, placé ici à `True` ce qui entraîne que la ligne complète et son saut de ligne sont capturés, y compris des espaces avant le saut de ligne.

— Ligne 14 : si on ignore l'argument `keepends`, la ligne complète privée de son saut de ligne est capturée, y compris des espaces avant le saut de ligne.

— Lignes 19-23 : Noter la présence ou l'absence de saut de ligne selon les cas (j'ai reformaté les deux lignes pour une meilleure lisibilité).

`join` est l'inverse de `split`

La méthode `join` recompose de ce que la méthode `split` décompose :

```
z="roseXorangeXbegonia"
L=z.split('X')
s='X'.join(L)
print(s==z)
```

```
True
```

autrement dit, on a toujours `sep.join(s.split(sep)) == s`.

En revanche, la méthode `split` n'est pas exactement l'inverse de la méthode `join`, autrement dit, il n'est pas toujours vrai que `sep.join(L).split(sep)` et `L` soient identiques. Par exemple

```
L=['rose', 'oraXge', 'begonia']
sep = 'X'
z=sep.join(L)
M=z.split(sep)

print(M)
```

```
['rose', 'ora', 'ge', 'begonia']
```

Comme on le constate, la liste `M` n'est pas identique à la liste originale `L` ; cela tient au fait que l'une des chaînes de `L` contient le séparateur comme sous-chaîne.

En revanche, si `L` est une liste de chaînes telle qu'aucune des chaînes ne contient une chaîne `sep` donnée comme sous-chaîne alors les chaînes `sep.join(L).split(sep)` et `L` sont identiques.

La méthode `join` vs l'addition augmentée

Voici un code qui sélectionne les consonnes présentes dans une chaîne donnée :

```
s= "broccoli"
VOYELLES ="aeiouy"
z = ""

for c in s:
    if c not in VOYELLES:
        z += c

print(z)
```

```
brccl
```

Toutefois, le procédé est assez laborieux. Dans ce genre de situations où on effectue des concaténations successives, on peut se demander si on ne peut pas appliquer la méthode `join`. Ce genre de code peut nécessiter davantage de connaissances en Python. Ici, l'opération est tout à fait réalisable : il suffit de collecter les consonnes et de les assembler avec `join` appliqué à la chaîne vide :

```
1 s= "broccoli"
2 VOYELLES ="aeiouy"
3
4 z = ''.join(c for c in s if c not in VOYELLES)
5
6 print(z)
```

```
7 brccl
```

Ici, ligne 4, on a utilisé une expression génératrice pour collecter les consonnes. Elles sont ensuite soudées avec `join`.

La méthode `join` appliquée à une liste courte

Si la liste est vide, `join` renverra une chaîne vide :

```
L = []
s = " > ".join(L)
print(s)
```

```

```

Si la liste contient un seul élément, la chaîne renvoyée ignore le séparateur :

```
L = ["File"]
s = " > ".join(L)
print(s)
```

```
File
```

Méthode `split` : chaînes vides ou pleine après séparation

Chaînes vides

Il se peut que des chaînes de la liste renvoyée par `split` soient vides :

```
s = "ABAxyABAABAxxABA"
sep= "ABA"

L=s.split(sep)

print(L)
```

```
['', 'xy', '', 'xx', '']
```

Rappelons en effet que si la sous-chaîne `sep` est présente en `k` séquences disjointes successives dans la chaîne `s` alors `s.split(sep)` est une liste de `k + 1` sous-chaînes délimitées par le retrait des `k` occurrences disjointes de `sep` dans `s`.

Dans le cas de l'exemple, `sep = "ABA"` apparaît, de gauche à droite, suivant 4 sous-chaînes disjointes. Plaçons un délimiteur sous forme de trait vertical à chaque extrémité de la chaîne :

```
|ABAxyABAABAxxABA|
```

Dans la chaîne précédente, plaçons dans un bloc chaque sous-chaîne `ABA` en l'encadrant par un délimiteur :

```
||ABA|xy|ABA||ABA|xx|ABA||
```

Il suffit alors de lire, de la gauche vers la droite, la succession des contenus entre deux séparateurs **qui n'encadrent pas** la sous-chaîne `aba`. On obtient, en écrivant verticalement pour une meilleure lisibilité :

```
||
|xy|
||
|xx|
||
```

ce qui explique pourquoi la valeur de `s.split(sep)` est :

```
['', 'xy', '', 'xx', '']
```

Séparateur absent

Lors d'un appel `s.split(sep)`, si le séparateur `sep` n'est pas présent dans la chaîne, la liste renvoyée par `split` est une liste dont le seul élément est la chaîne elle-même :

```
s = "bonjour"
print(s.split("X"))
```

```
['bonjour']
```

C'est cohérent avec le fait que `s.split(sep)` renvoie une liste de longueur 1 de plus que le nombre d'occurrences disjointes de `sep` dans `s` qui ici vaut 0.

Séparer des caractères, des lignes avec `join`

La méthode `join` s'applique non seulement à une liste (ou a un tuple) mais aussi à une simple chaîne :

```
s="_".join("ORANGE")

print(s)
```

```
O_R_A_N_G_E
```

En réalité, la méthode `join` peut prendre en argument n'importe quel itérable de chaînes. Par exemple, supposons qu'on dispose d'un fichier `fruits.txt`, de contenu

```
pomme
poire
fraise
abricot
```

et qu'on exécute le code suivant :

```
1 fichier = open("fruits.txt")
2 t = "OU BIEN ".join(fichier)
3 print(t)
```

À la ligne 1, on ouvre le fichier `fruits.txt` pour obtenir un objet Python de type `file`. Alors, comme `fichier` est un itérable sur les lignes du fichiers (saut de ligne inclus), la ligne 2 aura pour effet de placer dans une liste `t` les lignes du fichier séparées par la chaîne **OU BIEN**, comme on peut le voir ci-dessous :

```
pomme
OU BIEN poire
OU BIEN fraise
OU BIEN abricot
```

Somme de chaînes coûteuse

Soit `n` un entier positif et soit `c` une chaîne donnée. On cherche à construire une nouvelle chaîne où `c` est répété `n` fois et séparé du suivant par un tiret, par exemple si `c = "AAAAA"` et `n = 3` alors la chaîne à construire est :

```
AAAAA-AAAAA-AAAAA
```

Utilisons une boucle `for` pour créer la chaîne (ce n'est pas la bonne méthode pour faire cela en Python mais la question n'est pas là) :

```
c = "A" * 5
N = 3
s = c
for i in range(N - 1):
    s = s + '-' + c

print(len(s))
```

```
AAAAA-AAAAA-AAAAA
```

Prenons maintenant une assez valeur grande pour `n`, par exemple `n=100_000`. Le temps d'exécution va alors être prohibitif :

```
1 %%timeit -r2
2
3 c = "A" * 5
4 N = 100_000
5 s = c
6 for i in range(N - 1):
7     s = s + '-' + c
8
9 print(len(s))
```

```
10 599999
11 599999
12 599999
13 1 loop, best of 2: 3.41 s per loop
```

On lit le temps d'exécution en dernière ligne : plus de 3 secondes alors que ça devrait être quelques dizaines de ms. Le code a été écrit dans une cellule Jupyter Notebook ce qui permet de mesurer le temps d'exécution moyen (cf. ligne 1, ici 2 répétitions de l'exécution)

Pourquoi ce code est-il si lent ? *Réponse* : à chaque étape de la boucle `for` (ligne 7), l'expression `s + '-' + c` est évaluée. Pour cela, l'interpréteur Python regroupe l'expression de la manière suivante : `(s + '-') + c`. Pour calculer cette somme, il doit d'abord calculer `(s + '-')` ce qui va l'obliger à recopier `s` et à placer au bout le tiret. Comme la longueur de `s` est de plus en plus longue, le calcul va être de plus en plus coûteux. La complexité est quadratique en `n` (si on multiple par 10 le nombre de répétitions, le temps de calcul est multiplié par 100).

Cherchons une alternative, toujours en utilisant une boucle `for`. L'opération à répéter est une réaffection de `s` et qu'on peut aussi écrire :

```
s = s + ('-' + c)
```

On peut donc la remplacer par une **affectation augmentée** `+=` qui est beaucoup plus efficace car elle réalise la concaténation *en place* et la complexité sera quasi linéaire. Et en effet, le code met moins de 20 ms cette fois :

```
%%timeit -n1 -r1

c = "A" * 5
N = 100_000
s = c
```

```
for i in range(N - 1):
    s += '-' + c

print(len(s))
```

```
599999
1 loop, best of 1: 18 ms per loop
```

Noter que l'astuce précédente n'est pas toujours applicable. Par exemple, donnons-nous un entier, disons `n = 20` et supposons qu'on ait à construire la chaîne de caractères formée des entiers (écrits en chiffres) n, n − 1, etc jusqu'à 1 ce qui, avec l'exemple, donnerait :

$$\mathtt{n} = 20191817161514131211109876543 21$$

La façon assez assez naturelle d'écrire le code serait :

```
N = 20
s=''
for i in range(1, N+1):
    s = str(i) + s

print(s)
```

```
20191817161514131211109876543 21
```

et si n est grand, on va rencontrer la même lenteur d'exécution qu'avec le problème précédent mais, cette fois, sans pouvoir utiliser l'affectation augmentée `+=`.

De toute façon, la bonne façon de réaliser les deux tâches en Python était d'utiliser la méthode `join`. Par exemple, pour la première chaîne à construire, le code serait :

```
%%timeit -n1 -r1

N = 100000
c = "A" * 5
s='-'.join(c for i in range(N))

print(len(s))
```

```
599999
1 loop, best of 1: 15.3 ms per loop
```

et qui est légèrement plus rapide et c'est encore meilleur en utilisant une liste en compréhension :

```
%%timeit -n1 -r1

N = 100000
c = "A" * 5
s='-'.join([c for i in range(N)])

print(len(s))
```

```
599999
1 loop, best of 1: 9.32 ms per loop
```

Différentes approches pour concaténer une liste de chaînes

Dans ce paragraphe, nous allons envisager 5 méthodes pour concaténéer une liste de chaînes.

Autant le dire tout de suite, la seule méthode pythonnique et recommandée pour concaténer un itérable de chaînes est d'utiliser la méthode `join` sur une chaîne vide :

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]

s = "".join(L)
print(s)
```

```
MarsJupiterUranusNeptune
```

Une autre méthode envisageable est de construire la chaîne concaténée à partir d'une chaîne vide en ajoutant itérativement chaque chaîne de la liste :

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]

s = ''
for elt in L:
    s += elt
print(s)
```

```
MarsJupiterUranusNeptune
```

On pourrait tenter d'utiliser la fonction intégrée `sum` pour additionner les chaînes mais Python l'interdit :

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]

s = sum(L,[])
print(s)
```

```
    s = sum(L,[])
TypeError: can only concatenate list (not "str") to list
```

Une autre méthode qui peut être envisagée est d'utiliser la méthode `format` comme suggéré ICI à laquelle on transmet tous les éléments de la liste par décompression :

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]

gabarit = "{}"*len(L)
print(gabarit.format(*L))
```

```
MarsJupiterUranusNeptune
```

Une nouvelle possibilité consisterait à exploiter la faculté de la fonction `print` à réaliser une concaténation par le biais de son argument nommé `sep` et qu'on placerait sur une chaîne vide :

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]

print(*L, sep='')
```

```
MarsJupiterUranusNeptune
```

Le problème est de parvenir à capturer la chaîne affichée. Voici comment on peut faire en utilisant le module standard `io` :

```
1 from io import StringIO
2
3 L = ["Mars", "Jupiter", "Uranus", "Neptune"]
4 s = StringIO()
5
6 print(*L, sep='', file=s, end='')
7 print(s.getvalue())
```

```
8 MarsJupiterUranusNeptune
```

— Lignes 1 et 4 : le module standard `io` (qui veut dire input/output) permet de considérer une chaîne de type StringIO comme une « sortie ».

— Ligne 6 : la fonction `print` permet, grâce à un argument nommé `file`, d'envoyer le flux de caractères à afficher dans un objet assimilé à un fichier. Ici, ce fichier sera la chaîne `s` définie précédemment. Par ailleurs, par défaut, `print` termine son affichage en plaçant un saut de ligne mais comme nous voulons afficher des chaînes concaténées, on inhibe cette terminaison en utilisant l'argument nommé `end=''` (terminaison vide).

— Ligne 6 : l'affichage réalisé ne sera pas visible : par défaut, l'argument nommé `file` de `print` est la sortie standard `stdout` (en clair, dans la console de texte) et ici il a été changé en `s`.

— Ligne 7 : pour récupérer le contenu de la chaîne `s`, il faut appeler la méthode `getvalue`.

On peut comparer les performances de ces différentes approches :

```
from time import perf_counter
from random import randrange
from io import StringIO

def maketest(N, M):
    return ['X'*randrange(1, M+1) for _ in range(N)]

def join(A):
    return ''.join(A)

def add(A):
    s=''
    for a in A:
        s=s+a
    return s

def iadd(A):
    s=''
    for a in A:
        s+=a
    return s

def format(A):
```

```
    N=len(A)
    return ("{}"*N).format(*A)

def strIO(A):
    s= StringIO()
    print(*A, sep='', file=s, end='')
    return s.getvalue()

N=3*10**6
M=150
A=maketest(N, M)

F=[join, add, iadd, format, strIO]
check=[]

for f in F:
    begin_perf = perf_counter()

    L=f(A)

    delta = perf_counter() - begin_perf
    msg=f"{f.__name__ :<13}: {delta:.2f}s"
    check.append(len(L))
    print(msg)

print(check==[check[0]]*len(check))
```

qui affiche

```
join          : 0.13s
add           : 0.30s
iadd          : 0.30s
format        : 0.27s
strIO         : 0.55s
True
```

La méthode `join` est la plus rapide, assez largement et c'est d'ailleurs cette méthode qui est recommandée (dans le 2e paragraphe) pour concaténer des chaînes. Curieusement, les fonctions `add` utilisant `s = s + a` et `iadd` utilisant `s += a` ont des performances comparables alors que `s = s + a` est censé être plus coûteux puisque `s + a` recrée une nouvelle chaîne (il semblerait que Python 3 soit capable d'optimiser cette situation dans certains cas, voir ce message).

Remplacer des caractères d'une chaîne

On ne peut pas remplacer un caractère d'une chaîne donnée (puisqu'une chaîne est immuable), on ne peut qu'en donner l'illusion en créant une nouvelle chaîne déduite de la chaîne initiale. La méthode de chaîne `replace` permet de « remplacer » des blocs de caractères consécutifs d'une chaîne.

Par exemple :

```
1 z="abcXYefgXYh"
2 print(z.replace("XY","ZZZ"))
3 print(z)
```

```
4 abcZZZefgZZZh
5 abcXYefgXYh
```

— Ligne 1 : la chaîne originale

— Lignes 2 et 4 : `replace` crée une **nouvelle** chaîne où toute suite `XY` de `z` est remplacée par `ZZZ`.

— Lignes 3 et 5 : La chaîne originale `z` est préservée ie les blocs `XY` n'ont pas été modifiés.

L'exemple ci-dessus montre qu'il s'agit moins d'un *remplacement* dans la chaîne originale que d'une *copie* altérée de la chaîne originale

On pourrait avoir l'illusion d'un vrai remplacement dans la chaîne initiale avec le code suivant :

```
1 z = "abcXYefgXYh"
2 print(z)
3 z = z.replace("XY","ZZZ")
4 print(z)
```

```
5 abcXYefgXYh
6 abcZZZefgZZZh
```

— Ligne 4 : on pourrait penser que la chaîne `z` a eu ses caractères remplacés. En réalité, `z` a été **réaffecté** à la nouvelle chaîne, la chaîne initiale elle étant perdue.

Plusieurs remplacements successifs

Si on veut faire plusieurs remplacements, il faut appliquer plusieurs fois de suite la méthode `replace`. Par exemple soit à changer dans la chaîne `xxaybyxyxaabyxyab` le `x` en `A` et le `y` en `B` :

```
z="xxaybyxyxaabyxyab"
zz=z.replace("x","A")
zzz=zz.replace("y","B")
print(z)
print(zzz)
```

```
xxaybyxyxaabyxyab
AAaBbBABAaabBABab
```

ce qui pouvait être raccourci en

```
z="xxaybyxyxaabyxyab"
Z = z.replace("x","A").replace("y", "B")
print(Z)
```

```
AAaBbBABAaabBABab
```

Bien sûr, ces remplacements ont un coût à l'exécution puisque pour les réaliser :

— un algorithme de recherche doit être employé (Python utilise l'algorithme de Boyer-Moore)

— cet algorithme va parcourir toute la chaîne

— il y aura autant de parcours que d'appels à la méthode `replace`.

Dans notre cas, utiliser la méthode translate était plus approprié.

Option de la méthode `replace`

La méthode `replace` possède une option `count` qui permet de remplacer un nombre déterminé d'occurrences. Voici un exemple d'utilisation :

```
z="xxaybyxyxaabyxyab"
zz=z.replace("x","A", 3)

print(z)
print(zz)
```

```
xxaybyxyxaabyxyab
AAaybyAyxaabyxyab
```

Seules les 3 premières occurrrences du caractère `'x'` dans la chaîne `z` (sur un total de 5) ont été remplacées par le caractère `'A'`.

Supprimer des caractères dans une chaîne

Rappelons qu'une chaîne étant immuable, on ne va pas strictement supprimer des caractères d'une chaîne.

Remplacer des caractères d'une chaîne `s` par la chaîne vide `''` revient à supprimer ces caractères de `s`. Le remplacement est effectué à l'aide de la méthode `replace`.

Par exemple, soit la chaîne `s` suivante

```
Mars Jupiter  Uranus  Neptune     Pluton
```

contenant des mots séparés par des blancs de longueurs variables. On souhaite supprimer tous des espaces de `s`. Pour cela il suffit de « remplacer » chaque caractère espace par une chaîne vide :

```
1 s="Mars Jupiter  Uranus  Neptune     Pluton"
2 espace = " "
3 t= s.replace(espace, "")
4 print(s)
5 print(t)
```

```
6 Mars Jupiter  Uranus  Neptune     Pluton
7 MarsJupiterUranusNeptunePluton
```

— Ligne 7 : les espaces ont bien été supprimés

— Lignes 6 et 7 : La chaîne sans espace est une **nouvelle chaîne** `t`, la chaîne initiale `s` est inchangée.

— La chaîne vide `""` a permis d'écraser les caractères `" "`.

Transformer une chaîne par conversion en liste

On ne peut pas *directement* modifier une chaîne puisqu'une chaîne est de type immuable, par exemple on ne peut pas directement changer dans la chaîne `"taTou"` la lettre `"T"` en le caractère

"B".

Néanmoins, une liste étant mutable, on peut utiliser la stratégie suivante pour simuler une modification de chaîne `s` :

— créer avec le constructeur `list` une liste `L` des caractères de la chaîne initiale `s`
— modifier certains éléments de la liste `L`
— reconstituer avec la methode `join` une nouvelle chaîne `t` à partir de la nouvelle liste de caractères.

Illustration :

```
s = "taTou"
print(s)
print()

L =list(s)
print(L)
print()

L[2]='B'
print(''.join(L))
```

```
taTou

['t', 'a', 'T', 'o', 'u']

taBou
```

Les méthodes `partition` et `rpartition`

La méthode `partition` est une version locale de la méthode `split`. Le plus simple est de voir un exemple :

```
1 s = "abcd!_!xyz!_!XYZ"
2 print(s)
3
4 avant, sep, apres = s.partition("!_!")
5
6 print("avant :", avant)
7 print("sep :", sep)
8 print("apres :", apres)
```

```
9 abcd!_!xyz!_!XYZ
10 avant : abcd
11 sep : !_!
12 apres : xyz!_!XYZ
```

— ligne 4 : l'appel va partager la chaîne `s` en trois tronçons :
  — ce qui est à gauche du séparateur donné (ici `!_!`),
  — ce qui le séparateur lui-même

— ce qui est à droite du séparateur.

La partition se fait suivant la première occurrence du séparateur, autrement dit l'occurrence la plus à gauche.

Ci-dessous, le comportement de `partition` si le séparateur n'est pas dans la chaîne :

```
s = "abcd!_!xyz!_!XYZ"

avant, sep, apres = s.partition("123")

print("avant :", avant)
print("sep :", sep)
print("apres :", apres)
```

```
avant : abcd!_!xyz!_!XYZ
sep :
apres :
```

La méthode `rpartition`

La méthode `rpartition` est analogue à `partition` sauf que la partition s'effectue suivant l'occurrence du séparateur **la plus à droite**. Voici un exemple :

```
s = "abcd!_!xyz!_!XYZ"

avant, sep, apres = s.rpartition("!_!")

print("avant :", avant)
print("sep :", sep)
print("apres :", apres)
```

```
avant : abcd!_!xyz
sep : !_!
apres : XYZ
```

La méthode `rsplit`

La méthode `rsplit` est analogue à la méthode `split` en l'absence d'utilisation de l'option `maxsplit` :

```
numero = "08-42-20-32-17"
L = numero.rsplit("-")
M = numero.split("-")
print(L == M)
```

```
True
```

En revanche, si l'option `maxsplit` est utilisée, les occurrences du séparateur sont retirées en commençant **par la droite** (d'où le *r* pour *right*) au lieu de par la gauche comme c'est le cas pour `split` :

```
numero = "08-42-20-32-17"
L = numero.split("-", maxsplit=2)
print(L)
M = numero.rsplit("-", maxsplit=2)
print(M)
```

```
['08', '42', '20-32-17']
['08-42-20', '32', '17']
```

Recherche de sous-chaînes avec `in`

Soit la chaîne

```
s = "xyxABCyyyxABCxyyxABCxyx"
```

La chaîne `'ABC'` est une sous-chaîne de `s` et elle y apparaît même 3 fois. Rechercher une sous-chaîne dans une chaîne est une opération classique en programmation et appelée *recherche de motif*, en anglais *pattern matching*. Les caractères de la sous-chaîne doivent être contigus et non séparés ; ainsi la chaîne `ABC` n'est pas considérée comme étant une sous-chaine de la chaîne `xxxAxxxxBxxCxxxx`.

Une première possibilité de détection d'une sous-chaîne est d'utiliser l'opérateur `in` qui va renvoyer `True` ou `False` selon la présence ou l'absence de la sous-chaîne :

```
s = "xyxABCyyyxABCxyyxABCxyx"

print("ABC" in s)
print("AbC" in s)
```

```
True
False
```

En première approximation, voici comment se fait la recherche de la sous-chaîne : elle se poursuit jusqu'à obtention d'une occurrence de la sous-chaîne ou alors lorsque la fin de la chaîne est atteinte. Si une occurrence est découverte, la recherche s'interrompt.

Pour entrer dans les détails d'implémentation en CPython, il semble que l'opérateur `in` fasse appel aux algorithmes de Boyer-Moore et Horspool. L'examen du code-source de la fonction `PyUnicode_Contains` montre que des fonctions spécialisées de recherche sont invoquées, par exemple `ucs1lib_find`, qui font appel à la fonction `FASTSEARCH`.

Recherche de sous-chaînes avec `find`, `rfind`, `index` et `rindex`

L'opérateur `in` indique si oui ou non, une sous-chaîne est présente dans une chaîne mais, si elle est présente, il ne fournit aucun information sur sa position. C'est là que les méthodes `find` et `index` sont utiles : elles donnent l'indice de la 1re position d'une sous-chaîne dans une chaîne donnée.

Voici un exemple illustrant l'usage de `find` :

```
1 s = "xyyxxABCyyyxABCxyyxABCxyx"
2
```

```
3 p = s.find("DEF")
4 print(p)
5
6 p = s.find("ABC")
7 print(p)
```

```
8 -1
9 5
```

— Lignes 3 et 8 : si la sous-chaîne est absente, l'entier `-1` est renvoyé.

— Lignes 6 et 9 : si la chaîne est présente, l'indice du *début* de la *première* occurrence de la sous-chaîne est renvoyé. Ici, c'est 5 et, en effet, aux indices 5, 6 et 7 de la chaîne `s`, on lit `A`, `B` et `C`.

La méthode `index` est équivalente à la méthode `find` sauf qu'elle lève une exception `ValueError` lorsque la sous-chaîne est absente, ce qui peut être intimidant si vous ne connaissez pas les exceptions en Python. Voici un exemple :

```
1 s = "xyyxxABCyyyxABCxyyxABCxyx"
2
3 p = s.index("ABC")
4 print(p)
5
6 print()
7
8 p = s.index("DEF")
9 print(p)
```

```
10 5
11
12 Traceback (most recent call last):
13   File "code.py", line 8, in <module>
14     p = s.index("DEF")
15 ValueError: substring not found
```

— Lignes 3-4 et 10 : l'indice du début de la séquence est trouvé.

— Lignes 8 et 12-15 : indice non trouvé donc levée d'une exception et fin du programme.

On pourrait gérer l'exception :

```
s = "xyyxxABCyyyxABCxyyxABCxyx"

for sub in ["ABC", "DEF"]:
    try:
        p = s.index(sub)
        print(p)
        print()
    except ValueError:
        print(f'"{sub}" absent de "{s}"')
```

```
5
```

```
"DEF" absent de "xyyxxABCyyyxABCxyyxABCxyx"
```

L'implémentation des méthodes `find` et `index` est basée sur l'algorithme de Boyer-Moore-Horspool.

Paramètres de la méthode `find`

Les méthodes `find` et `index` disposent de paramètres optionnels permettant de fenêtrer la recherche. Regardons pour `find` :

```
1 s = "xyyxxABCyyyxABCxyyxABCxyx"
2
3 p = s.find("ABC", 10)
4 print(s[p:])
5
6 p = s.find("ABC", 10, 20)
7 print(s[p:])
```

```
8 12
9 ABCxyyxABCxyx
10 12
11 ABCxyyxABCxyx
```

— Ligne 3 : recherche le premier indice de la chaîne `s` à partir de l'indice 10 où pourrait se trouver la chaîne `ABC`

— Ligne 7 : recherche le premier indice de la chaîne `s`, entre les incides 10 (inclus) et 20 (exclu) où pourrait se trouver la chaîne `ABC`

Les méthodes `rfind` et `rindex`

Un appel `s.find(t, i, j)` cherche le plus petit indice de la chaîne `s`, situé entre l'indice `i` et l'indice `j` (non compris) où pourrait commencer la chaîne `t` et donc `find` cherche l'occurrence de `t` dans `s` qui soit la plus à gauche dans la plage à tester.

La méthode `rfind` fait la même chose mais recherche l'occurrence la plus à droite (d'où le `rfind` pour *right*). Voici un exemple :

```
1 s = "xyyxxABCyyyxABCxyyxABCxyx"
2 print(s[10:25])
3 p = s.rfind("ABC", 10, 25)
4 print(p)
5 print(s[p:])
```

```
6 yxABCxyyxABCxyx
7 19
8 ABCxyx
```

— Lignes 2-3 : on cherche la présence de la chaîne `ABC` dans `s` entre les indices 10 et 25, donc dans la sous-chaîne de la ligne 6.

— lignes 3-5 : l'indice le plus à droite (le plus grand donc) où commence `ABC` est 19 et `ABC`se lit au début de la sous-chaîne ligne 8.

La méthode `rindex` est analogue sauf qu'en cas d'échec de recherche, elle lève une exception au lieu de renvoyer -1.

Méthode `count`

La méthode de chaîne `count` permet de compter le nombre d'occurrences d'une sous-chaîne dans une chaîne :

```
s = "xyyxxABCyyyxABCxyyxABCxyx"
sub= "ABC"
N = s.count(sub)
print(N)
```

```
3
```

On notera que la méthode `count` entraîne le parcours caractère par caractère de la chaîne à examiner. L'implémentation est basée sur l'algorithme de Boyer-Moore-Horspool.

Sous-chaînes et chevauchement

Testons le nombre d'occurrences dans la chaîne `vvvABABAvvv` de la sous-chaîne `ABA` :

```
s = "vvvABABAvvv"
sub= "ABA"
N = s.count(sub)
print(N)
```

```
1
```

On pourrait considérer que la sous-chaîne est présente deux fois dans la chaîne puisque dans `ABABA`, la sous-chaîne `ABA` apparaît

— aux trois premières positions

— aux trois dernières.

Néanmoins, les fonctions Python de recherche de toutes les sous-chaînes effectuent toujours une lecture de la chaîne dans le sens gauche-droite et reprennent leur recherche immédiatement après la fin de la sous-chaîne trouvée. Ainsi, la méthode Python `count` ne comptera qu'une seule occurrence de la sous-chaîne `ABA` dans la chaîne `vvvABABAvvv`, et non deux.

Effacer les blancs aux extrémités d'une chaîne

On a parfois besoin de « normaliser » une chaîne de caractères en omettant les blancs qu'elle contient au début ou à la fin. Par exemple, soit la chaîne `z` présente dans le code ci-dessous

```
z = "   oui    "
print("Réponse : ...", z, "...")
```

ce qui affiche

```
Réponse : ...    oui     ...
```

On voudrait « éliminer » de la chaîne `z` les espaces en début et fin de chaîne et récupérer dans une nouvelle chaîne la sous-chaîne de contenu `oui`.

Pour cela, on utilise la méthode `strip` :

```
z = "   oui    "
zz = z.strip()
print("Réponse : ...", z, "...")
print("Réponse : ...", zz, "...")
```

```
Réponse : ...    oui     ...
Réponse : ... oui ...
```

La méthode `strip` appliquée à une chaîne `z` renvoie une nouvelle chaîne obtenue en supprimant tous les blancs initiaux et terminaux présents dans la chaîne `z`. Par blancs, il faut comprendre les espaces, tabulations et/ou sauts de ligne.

Suppression à droite ou à gauche

Pour supprimer uniquement les espaces en début d'une chaîne `z`, on applique à `z` la méthode standard `lstrip` :

```
z = "   oui    "
zz = z.lstrip()
print("Réponse : ...", z, "...")
print("Réponse : ...", zz, "...")
```

```
Réponse : ...    oui     ...
Réponse : ... oui     ...
```

L'initiale `l` dans `lstrip` fait référence à *left strip*.

De même, pour supprimer les espaces en fin de chaîne, on utilise la méthode `rstrip` :

```
z = "   oui    "
zz = z.rstrip()
print("Réponse : ...", z, "...")
print("Réponse : ...", zz, "...")
```

```
Réponse : ...    oui     ...
Réponse : ...    oui ...
```

L'initiale r dans `rstrip` fait référence à *right strip*.

Identification du début et fin de chaîne

On veut savoir si une chaîne donnée se termine, par exemple, par le suffixe *pdf*. On utilise pour cela la méthode `endswith` :

```
nomDeFichier="mon_doc.pdf"
is_pdf = nomDeFichier.endswith("pdf")
print(is_pdf)
```

```
True
```

Noter que le nom de la méthode contient un `s` et que `ends` et `with` ne sont pas séparés par un blanc souligné `_` comme c'est le cas pour certaines méthodes, par exemple `is_integer`.

Il est même possible de proposer plusieurs chaînes placées dans un tuple (et pas une liste) pour tester plusieurs terminaisons à examiner :

```
s = "my_code.py"
is_code = s.endswith(("c", "cpp", "pyx", "py"))
print(is_code)

s = "my_code.txt"
is_code = s.endswith(("c", "cpp", "pyx", "py"))
print(is_code)
```

```
True
False
```

De même, la méthode `startswith` permet de savoir si une chaîne commence par une chaîne donnée :

```
s ="pypy"
is_py = s.startswith("py")
print(is_py)

s ="Python"
is_py = s.startswith(("py", "Py"))
print(is_py)
```

```
True
True
```

Modification du début et fin de chaîne

On peut retirer de la fin d'une chaîne des caractères figurant dans une liste donnée. Pour cela on utilise la méthode `rstrip` (`r` pour *right*). Par exemple, supposons que l'on veuille supprimer de la fin d'une chaîne des signes de pontuation et les espaces :

```
s="Bien sûr ... !????"
print(s)
t=s.rstrip("?!, ;.")
print(t)
```

```
Bien sûr ... !????
Bien sûr
```

La méthode `lstrip` procède de même mais en début de chaîne :

```
s="-00042"
print(s)
t=s.lstrip("+-0")
print(t)
```

```
-00042
42
```

La méthode `strip` agit comme si `lstrip` et `rstrip` étaient appelées successivement :

```
s="+000.42000"
print(s)
t=s.strip("+-.0")
print(t)
```

```
+000.42000
42
```

Il est également possible, à partir de Python 3.9, de supprimer un préfixe donné du début ou de la fin d'une chaîne. Pour cela, on utilise la méthode `removeprefix`. Par exemple, soit à supprimer le préfixe `poly` :

```
L = ["polytechnique", "polygone", "polymorphisme", "Polyvalent"]
for s in L:
    t=s.removeprefix("poly")
    print(f"{s: <14} : {t}")
```

```
polytechnique  : technique
polygone       : gone
polymorphisme  : morphisme
Polyvalent     : Polyvalent
```

De même, on peut supprimer des suffixes avec la méthode `removesuffix`.

Chapitre III

Chaînes littérales, édition de code

Les différents délimiteurs de chaînes littérales

Il existe 4 types de délimiteurs de chaînes littérales :

— *l'apostrophe*
— le *guillemet anglais*, encore appelé *guillemet* ou *guillemet simple*
— les *triples apostrophes*
— les *triples guillemets*

et voici des exemples d'utilisation :

```
1  s = 'La Lune'
2
3  t = "La Terre"
4
5  u = """Mars
6  Jupiter
7  Saturne"""
8
9  v = '''Uranus
10 Pluton'''
11
12 print(s)
13 print(t)
14 print(u)
15 print('--------')
16 print(v)
```

```
17 La Lune
18 La Terre
19 --------
20 Mars
21 Jupiter
22 Saturne
23 --------
24 Uranus
```

```
25 Pluton
```

Un délimiteur quelconque de chaîne littérale sera appelé *quote*. L'apostrophe et le guillemet anglais seront appelés *quotes simples*. L'apostrophe et guillemet triples seront appelés *quotes triples*. Une chaîne littérale encadrée par des quotes simples sera appelée *chaîne littérale simple*. De même, pour une chaîne triple.

Le délimiteur qui ouvre une chaîne littérale est toujours du même type que le délimiteur qui la ferme. Ainsi la suite de caractère `'hello"` ne désigne pas une chaîne littérale valide.

Le choix du type de quotes est, la plupart du temps, **indifférent** et juste déterminé par la commodité de saisie et la lisibilité de la chaîne littérale. Concernant les chaînes triples, en deux mots, et comme on le voit dans les exemples ci-dessus (lignes 5-7 et 9-10), elles autorisent le saut de ligne, à la différence des chaînes simples.

Une chaîne littérale ouverte par un délimiteur simple mais non fermée entraîne une erreur de syntaxe :

```
1 s = "kiwi
2 print(s)
```

```
3 SyntaxError: EOL while scanning string literal
```

— Ligne 3 : `EOL` abrège *End Of Line* : le parseur indique qu'il a rencontré la fin de ligne alors qu'il était en train d'examiner une chaîne littérale et donc qu'il attendait le délimiteur de fin de chaîne.

C'est en fait le même problème que de vouloir passer une ligne à l'intérieur d'une chaîne simple :

```
1 s = 'Uranus
2 Pluton'
3
4 print(s)
```

```
5 SyntaxError: EOL while scanning string literal
```

— Lignes 1-2 : saut de ligne dans une chaîne littérale simple.

Enfin, ne pas confondre un guillemet et une apostrophe doublée :

```
s = ''kiwi''
print(s)
```

```
    s = ''kiwi''
              ^
SyntaxError: invalid syntax
```

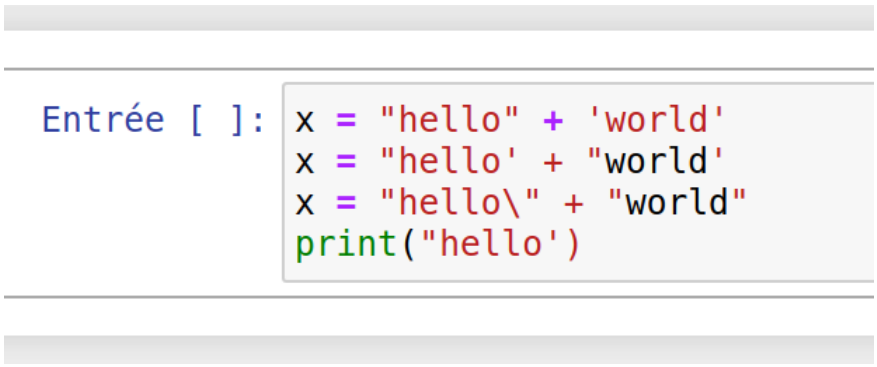
Un guillemet anglais constitue **un seul** caractère bien qu'il soit représenté comme si on utilisait deux apostrophes successives.

Chaîne littérale invalide et coloration syntaxique

Dans un éditeur de code, la coloration syntaxique permet en général de discerner qu'un chaîne littérale est invalide. Par exemple, le code suivant

```
x = "hello" + 'world'
x = "hello' + "world'
x = "hello\" + "world"
print("hello')
```

se présente ainsi dans une cellule Jupyter Notebook :



Les chaînes littérales sont normalement colorées en rouge. Les opérateurs en violet, les identificateurs quelconques et les parenthèses en noir. La ligne 1 est valide et sert de modèle. Les autres sont incorrectes :

— ligne 2 : le signe + devrait être en violet donc l'erreur est avant (apostrophe au lieu de guillemet)
— ligne 3 : même problème, l'apostrophe a été échappée avec la contre-oblique et donc le mot `world` se retrouve en noir.
— ligne 4 : la parenthèse finale est en rouge au lieu d'être en noir car les délimiteurs de chaîne littérale ne correspondent pas.

Quote dans une chaîne littérale

Soit à créer une chaîne littérale `S` de contenu :

```
La fable "Le Corbeau et le Renard" de La Fontaine
```

Noter que cette chaîne contient deux caractères guillemets qui entourent le nom de la fable.

Pour placer cette chaîne en tant que chaîne littérale dans du code Python, il n'est pas possible d'utiliser un délimiteur guillemet. Essayons :

```
1 s = "La fable "Le Corbeau et le Renard" de La Fontaine"
2 print(s)
```

```
3     s = "La fable "Le Corbeau et le Renard" de La Fontaine"
4                     ^
5 SyntaxError: invalid syntax
```

— Ligne 3 : le parseur Python qui lit la chaîne de la gauche vers la droite n'identifie que la chaîne littérale : **La fable** car le 2e guillemet marque, pour lui, la fin de la chaîne. La suite (qui commence avec la séquence de caractères *Le Corbeau*) ne peut être analysée par le parseur car elle ne correspond syntaxiquement à rien.

Pour faire de S une chaîne littérale, il suffit de l'encadrer par **un autre** délimiteur que le guillemet, par exemple l'apostrophe puisque le caractère apostrophe n'est pas présent dans la chaîne :

```
1 s = 'La fable "Le Corbeau et le Renard" de La Fontaine'
2 print(s)
```

```
3 La fable "Le Corbeau et le Renard" de La Fontaine
```

— Ligne 1 : noter que le délimiteur de la chaîne littérale est une apostrophe.

De même, pour créer une chaîne littérale contenant par exemple une apostrophe, comme la chaîne *Le loup et l'agneau*, il suffit de l'encadrer par un autre délimiteur que l'apostrophe, par exemple des guillemets, en supposant que le caractère guillemet ne soit pas présent dans la chaîne :

```
s = "Le loup et l'agneau"
print(s)
```

```
Le loup et l'agneau
```

En particulier, on peut définir des chaînes littérales qui représentent l'apostrophe et le guillemet :

```
APOSTROPHE = "'"
GUILLEMET = '"'
print("apostrophe ->", APOSTROPHE)
print(" guillemet ->", GUILLEMET)
```

```
apostrophe -> '
 guillemet -> "
```

Pour une chaîne contenant à la fois une apostrophe **et** un guillemet, il y a deux solutions qui seront présentées ultérieurement, l'une utilisant des chaînes triples et l'autre utilisant des échappements de quotes.

Triples quotes

Soit le texte brut suivant :

```
Mars
Jupiter
Uranus
Neptune
```

On cherche à construire une chaîne littérale s dont le contenu soit le même que celui affiché ci-dessus, **sauts de ligne inclus** avec la contrainte de ne pas utiliser de saut de ligne sous la forme \n.

Une chaîne littérale avec simple délimiteur (guillemet ou d'apostrophe) ne peut parvenir au résultat : en effet, les caractères d'une chaîne littérale simple, doivent se trouver *sur une même ligne* de code (voir cependant Découper une chaîne littérale sur plusieurs lignes).

Justement, un des rôles des chaînes triples est de permettre d'obtenir un caractère *saut de ligne* en se contentant de taper sur la touche ENTER :

```
1 s = """Mars
2 Jupiter
3 Uranus
4 Neptune"""
5
6 print(s)
```

```
7 Mars
8 Jupiter
9 Uranus
10 Neptune
```

— Lignes 1-4 : l'utilisateur a tapé trois fois sur la touche ENTER pour créer la chaîne littérale `s`. Noter que la chaîne doit commencer et se terminer par des triples quotes, ici des triples guillemets. Les apostrophes de fermeture doivent être écrits sur la même ligne que la fin de la chaîne sinon, vous créez un saut de ligne final dans la chaîne (qui ici n'était pas souhaité).

— Lignes 7-10 : les sauts de lignes apparaissent à l'affichage.

Il aurait été possible d'obtenir le même effet avec une chaîne encadrée par des apostrophes triples :

```
s = '''Mars
Jupiter
Uranus
Neptune'''

print(s)
```

Par ailleurs, il est possible d'utiliser des triples guillemets ou des triples apostrophes en l'absence de saut de ligne :

```
u = """Mars Jupiter Uranus Neptune"""
print(u)
```

```
Mars Jupiter Uranus Neptune
```

On peut, sauf coïncidence malheureuse, placer à l'intérieur d'une chaîne triple des guillemets et des apostrophes simples :

```
s = """La Terre ('planète bleue')
Mars ("planète rouge")"""

print(s)
```

```
La Terre ('planète bleue')
Mars ("planète rouge")
```

Choisir une chaîne triple est donc parfois une **solution** quand l'usage d'une chaîne simple provoque une erreur :

```
s = """La fable "Le loup et l'agneau" de la Fontaine"""
print(s)
```

```
La fable "Le loup et l'agneau" de la Fontaine
```

Attention, cependant : une fois **dans** la chaîne littérale, l'interpréteur cherche une succession de trois quotes pour décider de la fin de la chaîne. Si vous êtes malchanceux, vous aurez une erreur :

```
s = """La Terre ("planète bleue")
Mars  est la "planète rouge""""

print(s)
```

```
    Mars dite  "planète rouge""""
                                ^
SyntaxError: EOL while scanning string literal
```

Ici, il était facile d'y remédier en entourant dès le départ par des triples apostrophes :

```
s = '''La Terre ("planète bleue")
Mars  est la "planète rouge"'''

print(s)
```

```
La Terre ("planète bleue")
Mars  est la "planète rouge"
```

Usage différencié des délimiteurs de chaînes littérales

Une chaîne littérale peut être entourée de différents délimiteurs. En fonction de quel critère choisira-t-on de définir une chaîne littérale avec tel délimiteur plutôt que tel autre ? Un critère est celui de la meilleure lisibilité ; en particulier, on pourra considérer que la présence de séquence d'échappement nuit à la lisibilité

Délimiteurs simples

Les deux délimiteurs simples sont utilisés souvent de façons équivalentes, bien que l'usage chez les programmeurs semble privilégier l'usage de l'apostrophe sur le guillemet lorsqu'il y a équivalent.

Néanmoins, si une chaîne contient une apostrophe, par exemple `l'orange`, l'apostrophe ne peut pas servir de délimiteur et on choisira alors des guillemets :

```
s = "l'orange"
print(s)
```

```
l'orange
```

De même, si une chaîne contient un ou plusieurs guillemets comme `La fable "Le Corbeau et le Renard"` on utilisera des apostrophes comme délimiteur.

Délimiteurs triples

Les délimiteurs triples s'utilisent essentiellement lorsque la chaîne littérale contient des sauts de lignes et que pour des raisons de lisibilité et de commodité d'édition, le programmeur ne veut

pas faire apparaître les sauts de ligne sous forme de séquences d'échappement (\n) mais sous forme de passage « physique » à ligne. Typiquement :

```
s = """Mars
Jupiter
Uranus
Neptune"""

print(s)
```

```
Mars
Jupiter
Uranus
Neptune
```

Un usage fréquent des chaînes triples est le placement d'une docstring bien que des doctrings unilignes puissent être entourées de quotes simples (et cela se rencontre dans du code de référence). Voici un exemple inspiré d'une fonction du code-source de Python :

```
1 def mean(data):
2     """Return the sample arithmetic mean of data.
3
4     >>> mean([1, 2, 3, 4, 4])
5     2.8
6     """
7     return sum(data)/len(data)
8
9
10 print(mean([1, 2, 3, 4, 4]))
```

```
11 2.8
```

— Lignes 2-6 : une docstring sous forme de chaîne triple.

Une docstring est un détournement astucieux de la notion de chaîne littérale. Attention toutefois à ne pas croire qu'un chaîne triple (ou pas, d'ailleurs) peut remplacer un commentaire, une chaîne triple, seule sur une ou plusieurs ligne, étant une (unique) instruction Python.

Une chaîne triple peut être entourée de guillemets triples ou d'apostrophes triples. En cas d'équivalence d'usage, il semble plus habituel de placer des guillemets triples.

Une chaîne triple est appropriée pour encadrer le contenu d'une chaîne littérale contenant à la fois des apostrophes et des guillemets.

Le fait qu'il y ait deux types de délimiteurs triples permet de placer une chaîne triple comme contenu d'une chaîne littérale :

```
s = '''La docstring est :

"""Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

```
Sparse is better than dense.
Readability counts."""

extrait du "Zen of Python".'''

print(s)
```

qui affiche

```
La docstring est :

"""Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts."""

extrait du "Zen of Python".
```

Chaînes triples et indentation

Les chaînes triples peuvent en apparence ne pas respecter d'indentation

```
if True:
    if 42==40+2:
        if 42==40+2:
            if 42==40+2:
                if 42==40+2:
                    if 42==40+2:
                        if 42==40+2 :
                            if 42==40+2:
                                z="""Le Loup et l'agneau

Un agneau se ..."""

print(z)
```

```
Le Loup et l'agneau

Un agneau se ...
```

— Malgré les apparences, bien que l'affectation soit indentée (ligne 9), ce code respecte l'indentation (ligne 11). Le retrait est **obligatoire** si on ne veut pas pas introduire une indentation **erronée** dans la chaîne littérale.

Si on avait conservé l'indentation, on aurait obtenu un affichage sans doute non souhaité :

```
1 if True:
2     if 42==40+2:
3         if 42==40+2:
4             if 42==40+2:
5                 if 42==40+2:
6                     if 42==40+2:
7                         if 42==40+2 :
8                             if 42==40+2:
9                                 z="""Le Loup et l'agneau
10
11                                    Un agneau se ..."""
12
13 print(z)
```

```
14 Le Loup et l'agneau
15
16                                    Un agneau se ...
```

— Ligne 16 : le vers est indenté à tort, à cause de l'indentation incorrecte ligne 11.

C'est une des difficultés que peuvent poser les chaînes triples puisque le code Python est souvent amené à être indenté, par obligation syntaxique

Générer du code Python avec la fonction `repr`

Comme la fonction `repr` génère une chaîne littérale Python dont le contenu est une expression Python valide, on peut utiliser facilement la fonction `repr` pour générer du code Python. Par exemple, on dispose d'une suite de mots dans un fichier :

```
lundi
mardi
mercredi
jeudi
vendredi
samedi
dimanche
```

et on veut créer le code d'une **liste Python** dont tous les éléments soient les chaînes définies par cette liste de mots (par exemple, pour pouvoir replacer la liste dans un code-source Python). Plus précisément, on voudrait générer automatiquement le code Python suivant :

```
["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"]
```

à partir de la chaîne lisible dans le texte ci-dessus (noter que les mots doivent apparaître, non pas entourés d'apostrophes, mais de guillemets même si cela ne change pas la valeur de l'objet). Voici comment la fonction `repr` permet d'accomplir l'objectif :

```
1 mots="""lundi
2 mardi
3 mercredi
4 jeudi
```

```
5 vendredi
6 samedi
7 dimanche"""
8
9 code = repr(mots.split())
10 print(code)
11 print()
12
13 code_final = code.replace("'", '"')
14
15 print(code_final)
```

```
16 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
17
18 ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"]
```

— Ligne 18 : objectif atteint puisqu'on a placé les mots dans une liste Python à partir de la chaîne littérale et que les mots sont entourés de guillemets.
— Lignes 1-7 : on fait un copier-coller du texte depuis le fichier (d'autres méthodes seraient envisageables) dans le fichier-source et on entoure de triples quotes pour en faire une chaîne littérale
— Ligne 9 : à l'aide de la méthode `split`, on crée la liste (objet Python) cherchée ; appliquer `repr` à cet objet transforme l'objet en code littéral Python
— Ligne 10 et 16 : par défaut, Python affiche les chaînes littérales entourées d'apostrophes. On souhaiterait ici plutôt avoir des guillemets.
— Ligne 13 : disposant d'une chaîne (ligne 9), on peut la modifier avec des fonctions Python pour la transformer ; ici, on change les apostrophes en guillemets.

Découper une chaîne littérale sur plusieurs lignes

Soit le programme suivant contenant une longue chaîne littérale :

```
s = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
print(len(s))
```

```
62
```

— Noter que 62 = 26 + 26 + 10 où 26 désigne le nombre de lettres de l'alphabet et 10 le nombre de chiffres décimaux.

Pour des raisons de lisibilité, on peut vouloir faire apparaître la chaîne `s` coupée en différents tronçons, par exemple, on voudrait faire apparaître dans la chaîne littérale les regroupements suivants :

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
```

Solution adéquate

Pour découper `s` en tronçons, il suffit de couper la chaîne sur plusieurs lignes logiques, séparées par le symbole de contre-oblique \ :

```
s = "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ\
0123456789"
print(len(s))
```

```
62
```

En effet, la séquence de caractères \ENTER (c'est-à-dire la contre-oblique suivie d'un appui sur la touche ENTER) réalise une continuation de ligne (cela inhibe le saut de ligne). Cela permet de découper une longue ligne (« logique » comme on dit) sur plusieurs lignes (« physiques »).

Noter cependant que le formatage suivant, qui cherche à aligner les débuts de lignes, modifie la chaîne littérale :

```
1 s = "abcdefghijklmnopqrstuvwxyz\
2       ABCDEFGHIJKLMNOPQRSTUVWXYZ\
3       0123456789"
4 print(len(s))
```

```
5 72
```

— Ligne 5 : la longueur de la chaîne est 72 alors que la chaîne initiale est de longueur 62 ; cela est dû aux espaces ajoutés avant ABCD... (ligne 2) ainsi que 0123... (ligne 3).

Noter qu'il serait possible d'obtenir un effet d'alignement :

```
s = "\
abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ\
0123456789"
print(len(s))
```

```
62
```

On aurait pu essayer d'utiliser une chaîne triple. Mais la représentation suivante n'est cependant pas fidèle :

```
s = """abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789"""
print(len(s))
```

```
64
```

— la longueur de s n'est pas la même que celle de la chaîne initiale (64 au lieu de 62) car des sauts de lignes ont été introduits dans la chaîne triple.

Lien vers la documentation officielle : Explicit line joining

Découper du code Python sur plusieurs lignes

Le placement d'un saut de ligne dans du code Python est très strict. Une instruction uniligne ne peut pas a priori être placée sur deux lignes. Par exemple, soit à évaluer la longue addition ci-dessous :

```
8 + 11 + 17 + 14 + 10 + 11 + 6 + 4 + 1 + 7 + 7 + 8 + 2 + 6 + 5 + 20 + 20 + 2
```

en plaçant la somme sur deux lignes de code Python :

```
s = 8 + 11 + 17 + 14 + 10 + 11 + 6 + 4 + 1 +
7 + 7 + 8 + 2 + 6 + 5 + 20 + 20 + 2

print(s)
```

Ce code poduit une erreur :

```
    s = 8 + 11 + 17 + 14 + 10 + 11 + 6 + 4 + 1 +
                                                 ^
SyntaxError: invalid syntax
```

et cette erreur est due au passage à la ligne.

Il existe deux façons de pallier cette limitation :

— placer une contre-oblique à l'endroit où on veut couper la ligne,
— entourer la liste entre des délimiteurs appariés, comme des parenthèses.

voir les paragraphes dédiés pour le détail.

Enjambement de lignes avec un appariement

On peut contourner l'impossibilité de passer à la ligne à l'intérieur d'une expression en plaçant l'expression à l'intérieur d'une paire de délimiteurs comme des parenthèses, des crochets, des accolades, selon les possibilités qu'offre la situation. Par exemple, pour présenter une longue addition, pourra faire comme suit :

```
1 s = (58 + 20 + 33 + 86 + 87 + 25 +
2       74 + 70 + 54 + 16 + 58 + 40 +
3       49 + 54 + 92 + 13 + 51 + 51)
4 print(s)
```

```
5 931
```

— Lignes 1-3 : l'expression a été placée entre parenthèses et elle peut être ensuite découpée en autant de lignes que l'on souhaite. On peut aussi placer des blancs comme on le souhaite à l'intérieur des parenthèses. Les 3 lignes sont dites lignes *physiques* et elles correspondent à une seule ligne dite *logique*.

Cette possibilité est applicable dans de nombreuses situations et permet d'améliorer la lisibilité du code. Elle permet par exemple, de placer des listes littérales sur plusieurs lignes :

```
1 for L in [
2             [5, 8, 0, 9, 1, 0, 6, 4],
3             [0, 0, 0, 4, 0, 3, 0],
4             [5, 8, 1, 6, 4],
5             [-5, -8],
6             [0, -5, -8],
7             [-3, 0, -5, -8],
8             [0, 5, -8],
```

```
9               [42],
10              [-42]]:
11      print(L, '->', sum(L))
```

— lignes 2-10 : on a placé une grosse liste de listes que l'on peut indenter comme on souhaite.

Le placement d'un code entre délimiteur permet aussi l'insertion de commentaires en fin de ligne physique :

```
L = [
    65, 31, 9, 81,      # impairs
    32,  82, 46, 12     # pairs
    ]
```

Enjambement de lignes avec une contre-oblique

On peut contourner l'impossibilité de passer à la ligne dans une expression en plaçant une contre-oblique juste à l'endroit où on fait le saut de ligne dans l'expression. Exemple appliqué au calcul d'une longue addition :

```
1 s = 58 + 20 + 33 + 86 + 87 + 25 + 74 + 70 + 54 + \
2     16 + 58 + 40 + 49 + 54 + 92 + 13 + 51 + 51
3 print(s)
```

```
4 931
```

— Ligne 1 : on a placé la contre-oblique à la fin de la ligne, juste avant d'effectuer le saut de ligne. Il ne doit y avoir aucun caractère entre la contre-oblique et l'endroit où on appuie sur la touche ENTRÉE pour sauter la ligne.

— Ligne 1-2 : ces deux lignes constituent, pour l'interpréteur Python, une seule ligne (*logique*). Chacune des deux lignes est dite *physique*.

— Ligne 2 : on peut indenter, à des fins de meilleure lisibilité, pour aligner les termes de la somme car cela signifie juste ajouter des espaces entre le signe + de la fin de la ligne 1 et le nombre 16 de la ligne 2.

Il est même possible de placer plusieurs contre-obliques pour la même espression :

```
s = 58 + 20 + 33 + 86 + 87 + 25 +\
    74 + 70 + 54 + 16 + 58 + 40 +\
    49 + 54 + 92 + 13 + 51 + 51
print(s)
```

```
931
```

Puisqu'on place une contre-oblique avant de taper sur la touche ENTRÉE, cette possibilité peut se voir comme un échappement du saut de ligne et j'écrirai parfois \ENTRÉE.

Il n'y a que très peu de limitation à cette règle ; en voici une néanmoins : on ne peut pas couper un nombre littéral, ni couper un identificateur. Exemple :

```
# Code interdit
pi = 3.141592653589\
    7932384626433
```

```
# Code interdit
maLongue\
Variable = 42
```

Il est par ailleurs recommandé de limiter l'usage de ce type de coupure de ligne.

Fusion de chaînes littérales adjacentes

Des chaînes littérales placées côte-à-côte et sur une même ligne (logique) dans du code-source sont traitées comme si elles formaient une seule chaîne littérale :

```
s = "Rose""Kiwi""Prune"
print(s)
```

```
RoseKiwiPrune
```

Les chaînes peuvent même être séparées par des espaces et avoir des séparateurs différents :

```
s = "Rose" 'Kiwi'  """Prune
Fraise
Poire"""

print(s)
```

```
RoseKiwiPrune
Fraise
Poire
```

Lien vers la documentation officielle : String literal concatenation

Édition lisible de chaînes littérales

En utilisant les possibilités de fusion de chaînes littérales adjacentes ainsi que le découpage de lignes logiques, il est possible de formater dans le code-source une chaîne littérale, longue ou structurée, de façon lisible. Soit le programme suivant contenant une longue chaîne littérale :

```
s = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
print(len(s))
```

```
62
```

On peut rendre plus lisible la chaîne littérale ci-dessus par le code suivant :

```
1 s = "abcdefghijklmnopqrstuvwxyz"\
2     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"\
3     "0123456789"
4 print(len(s))
```

```
5 62
```

— Lignes 1-3 : il s'agit d'une seule ligne logique mais répartie sur trois lignes physiques grâce à l'échappement \ENTER. On notera qu'aucun caractère du contenu de la chaîne s n'est le caractère saut de ligne.

— lignes 2 et 3 : les deux chaînes littérales sont précédées d'espaces pour permettre l'alignement vertical des 3 chaînes. Ces espaces ne sont pas incorporés dans la chaîne, ils servent juste à la présentation.

Comme aucun caractère du contenu de la chaîne `s` n'est le caractère saut de ligne, la chaîne ne pouvait être ré-écrites avec des quotes triples.

On peut même encore simplifier le code : le code d'une expression peut être coupé sur plusieurs lignes si l'expression est entourée d'une paire de délimiteurs comme des parenthèses. D'où la ré-écriture suivante :

```
s = ("abcdefghijklmnopqrstuvwxyz" # minuscules
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ" # majuscules
     "0123456789")                # chiffres

print(len(s))
```

```
62
```

Noter qu'il a même été possible de rajouter des commentaires en fin de chaque ligne.

La problématique de l'échappement

Imaginons une chaîne littérale représentant un texte contenant des sauts de lignes, par exemple :

```
Lundi
Mardi
Mercredi
```

Pour qu'une telle chaîne existe en Python, une chaîne de caractères doit pouvoir exprimer avec un marqueur spécifique qu'elle contient, à tel endroit, un saut de ligne. Or, le saut de ligne est un caractère non-imprimable. Il faut donc trouver un mécanisme pour pouvoir représenter le caractère *saut de ligne* à partir d'autres caractères sans que ces caractères soient interprétés à la lettre. Ce mécanisme s'appelle *l'échappement*.

L'*échappement* en Python est effectué avec le caractère \, dit *contre-oblique*. Ainsi, pour représenter un saut de ligne, en anglais *newline*, on utilise le caractère `n` précédé d'une contre-oblique, ce qui donne le caractère `\n` :

```
1 s = "Lundi\nMardi\nMercredi"
2 print(s)
```

```
3 Lundi
4 Mardi
5 Mercredi
```

— Ligne 1 : quand la chaîne est parcourue de la gauche vers la droite, la combinaison du caractère \ suivi du caractère `n` forme un **unique** caractère, interprétée comme un saut de ligne. La chaîne `s` contient trois fois cette combinaison. Noter que la coloration syntaxique du code a placé en gras la séquence d'échappement.

— Lignes 3-5 : les trois mots sont bien séparés par des sauts de ligne.

Caractère unique

Ainsi, la présence d'une contre-oblique devant certains caractères a pour effet de modifier le sens habituel (*littéral*) du caractère. Dans ce contexte, on dit que la contre-oblique est un *caractère d'échappement* et la combinaison contre-oblique + caractère formant le caractère interprété est appelée la *séquence d'échappement*. Par exemple, les deux caractères assemblés `\n` forment une séquence d'échappement.

Une séquence d'échappement s'écrit avec deux caractères (rarement plus) dans le code-source mais représente un seul caractère Python. Pour le vérifier, il suffit d'examiner la longueur d'une chaîne contenant une séquence d'échappement :

```
1 s = "\n"
2 print(len(s))
```

```
3 1
```

— Ligne 1 : `s` est littéralement constitué de deux caractères : la contre-oblique et le caractère `n`

— Ligne 3 : `s` **représente** UN seul caractère (ici, le saut de ligne).

Représenter une quote par une séquence d'échappement

Comme l'apostrophe et le guillemet sont à la fois des caractères et des délimiteurs de chaînes, il est possible d'échapper ces caractères avec une contre-oblique. Ainsi, le caractère `\'` est la séquence d'échappement représentant une apostrophe, le caractère `\"` est la séquence d'échappement représentant un guillemet anglais.

Voici un exemple d'utilisation :

```
1 s = "La fable \"Le loup et l\'agneau\" de la Fontaine"
2 t = 'La fable \"Le loup et l\'agneau\" de la Fontaine'
3 print(s)
4 print(t)
```

```
5 La fable "Le loup et l'agneau" de la Fontaine
6 La fable "Le loup et l'agneau" de la Fontaine
```

— Lignes 1 et 2 : les chaînes `s` et `t` ont même valeur ; elles sont seulement encadrées par des délimiteurs différents.

— Lignes 1 et 2 : les deux guillemets autour du titre *Le loup et l'agneau* sont échappés en `\"`

— Lignes 1 et 2 : l'apostrophe qui précède le mot *agneau* est échappée en `\'`.

Séquences d'échappement usuelles

Voici les séquences d'échappement les plus usuelles :

Nom	Séquence d'échappement
Saut de ligne	`\n`
Tabulation	`\t`
Contre-oblique	`\\`
Ignorance d'un saut de ligne	`\ENTER`
Apostrophe	`\'`
Guillemet	`\"`

Ces séquences ne s'appliquent qu'à des chaînes littérales sauf `\ENTER` qui s'applique aussi à du code Python, cf. Enjambement de lignes avec une contre-oblique.

Échappement du caractère contre-oblique

Le caractère noté \ est appelé *contre-oblique* (en anglais *antislash*). C'est un caractère de type ponctuation. On s'intéresse à ce caractère, non pas en tant que caractère d'échappement dans une chaîne littérale, mais en tant que **vrai caractère** pouvant être utilisé dans une chaîne au même titre qu'une lettre.

Soit à créer une chaîne littérale de contenu :

```
C:\Users\Pascal
```

— La chaîne contient 15 caractères (1+5+6 = 12 lettres, le caractère deux-points et deux caractères contre-oblique)

Si on cherche à afficher en tant que chaîne littérale :

```
path = "C:\Users\Pascal"
print(path)
```

```
  File "code.py", line 1
    path = "C:\users\Pascal"
           ^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode
                    bytes in position 2-3: truncated \uXXXX escape
```

on aurait un message d'erreur, cf. la sortie ci-dessus. *Explication :* la séquence de caractères `\U` (dans `\Users`) au lieu d'être interprétée comme deux caractères (d'abord \ puis `U`) serait interprétée comme une séquence d'échappement.

Le caractère \ ne peut être représenté tel quel dans une chaîne littérale. Pour obtenir un caractère contre-oblique, il suffit d'écrire `\\`, ce qui est assez logique puisque la contre-oblique sert à réaliser un échappement.

Cette fois l'affichage est réalisé correctement :

```
path = "C:\\Users\\Pascal"
print(path)
```

```
C:\Users\Pascal
```

En particulier, la chaîne littérale ayant pour unique caractère une contre-oblique est `"\\"` :

```
print("\\")
```

```
\
```

Ces questions sont particulièrement importantes à comprendre si vous êtes amené à utiliser des expressions régulières.

Contre-oblique littérale : bonne pratique

Ce paragraphe traite d'une question très marginale.

Une contre-oblique dans une chaîne littérale n'est pas forcément le début d'une séquence d'échappement. C'est le cas si le caractère qui suit la contre-oblique ne définit pas une séquence d'échappement répertoriée :

```
s = "C:\Documents and Settings"
print(s)
```

```
C:\Documents and Settings
```

— A l'affichage (ligne 3), on lit bien une contre-oblique suivie de la lettre `D`, cela veut dire que dans la chaîne littérale (ligne 1), les deux caractères `\D` ne forment pas une séquence d'échappement mais deux caractères autonomes.

Si on est malchanceux, et si on n'a pas connaissance de la liste de toutes les séquences d'échappement, la chaîne construite peut ne pas correspondre à ce qu'on souhaite.

Or, il est possible de ne pas produire de séquence d'échappement si on **double la contre-oblique**. Donc on écrira plutôt (ligne 4) :

```
s = "C:\Documents and Settings"
print(s)
s = "C:\\Documents and Settings"
print(s)
```

```
C:\Documents and Settings
C:\Documents and Settings
```

L'intérêt de doubler une contre-oblique littérale est que si, par exemple, dans une version ultérieure de Python, `\D` devient une séquence d'échappement, la version `\D` n'aurait plus la même interprétation tandis que le code utilisant `\\D` resterait valide. Bien qu'il paraît improbable qu'une telle séquence soit introduite, tant cela risquerait de casser de code, le risque n'est pas exclu que dans un future version de Python, cet usage entraîne une erreur de syntaxe.

Le caractère tabulation

La tabulation horizontale, sert à faire des alignements verticaux de texte, comme dans le tableau suivant de nombres en colonnes :

```
45        119       43        54
490        23       72        39
1        23       256        44
```

La question est de savoir comment placer des tabulations horizontales dans une chaîne littérale. Pour cela, on marque la tabulation dans la chaîne à l'aide de la séquence d'échappement `\t` :

```
print("45\t119\t43\t54\n490\t23\t72\t39\n1\t23\t256\t44")
```

```
45      119     43      54
490     23      72      39
1       23      256     44
```

Noter qu'en réalité, il n'est pas nécessaire d'utiliser une séquence d'échappement pour placer des tabulations dans une chaîne littérale :

```
t="""45         119         43          54
490          23          72          39
1          23          256          44"""

print(repr(t))
```

```
'45\t119\t43\t54\n490\t23\t72\t39\n1\t23\t256\t44'
```

et la sortie montre bien que des tabulations ont été saisies au clavier pour séparer les nombres.

Il ne faut pas confondre cette possibilité d'utiliser des « tabulations-clavier » dans une chaîne littérale Python et les restrictions à utiliser ces mêmes tabulations dans du *code* Python hors de chaînes littérales.

Les chaînes littérales ne s'appliquent qu'à du code-source

Les multiples techniques permettant de construire des chaînes littérale ne doivent pas faire croire que le contenu d'une chaîne littérale est « complexe » voire qu'il y aurait un type *chaîne littérale*. Une fois une chaîne référencée dans une variable, à l'exécution du programme, sa provenance éventuelle d'une chaîne littérale n'a aucune particularité.

Par exemple, soit la chaîne de caractères suivante

```
La chaîne de caractères : 'J'ai entendu : " on joue au "plus\moins""'
```

Cette chaîne est assez compliquée à construire *en tant que chaîne littérale*, cela pourrait par exemple être :

```
s = 'La chaîne de caractères : \'J\'ai entendu : " on joue au "plus\\moins""\''
print(s)
```

```
La chaîne de caractères : 'J'ai entendu : " on joue au "plus\moins""'
```

Toutefois, on peut très bien capturer cette chaîne sans aucun effort si elle n'est pas une chaîne littérale, par exemple si cette chaîne est récupérée par la fonction `input` :

```
S = input("Entrez du texte :\n")
print("Voici votre texte :\n")
print(S)
```

```
Entrez du texte :
La chaîne de caractères : 'J'ai entendu : " on joue au "plus\moins""'
Voici votre texte :
```

```
7
8 La chaîne de caractères : 'J'ai entendu : " on joue au "plus\moins""'
```

— Lignes 1 et 4-5 : la chaîne est capturée par la fonction `input`, elle est produite par un utilisateur
— Lignes 3 et 8 : la chaîne est restituée telle quelle.

Identification du contenu d'une chaîne littérale

Une chaîne pré-existe à sa représentation en Python et un écart important peut séparer son contenu de sa représentation en chaîne littérale. Par exemple, soit la chaîne suivante placée sur deux lignes :

```
C:\toto
"OK"
```

Une représentation possible de ce texte sous forme de chaîne littérale Python pourrait être :

```
S = "C:\\toto\n\"OK\""
```

Il est bien clair qu'il n'est pas évident de déchiffrer le contenu de `S` et d'y reconnaître le texte ci-dessus.

Une méthode simple pour identifier le contenu d'une chaîne présentée par une chaîne littérale consiste à :

— afficher la chaîne avec la fonction `print`
— lire la suite de caractères produits.

Illustration :

```
S = "C:\\toto\n\"OK\""
print(S)
```

```
C:\toto
"OK"
```

Il apparaît alors que `S` contient 12 caractères (7 lettres, 1 saut de ligne et 4 caractères de ponctuation). Noter que pourtant, la représentation de `S` en tant que chaîne littérale Python est très différente et nécessite, non pas 12, mais 16 caractères. Cette méthode est presque infaillible (elle pourrait échouer par exemple si la chaîne contenait un caractère `\b` mais c'est anecdotique).

Le contenu d'une chaîne est la succession de caractères qu'on lit dans une console quand on affiche la chaîne avec la fonction `print` (en supposant que la chaîne soit formée de caractères imprimables).

Une autre méthode, très simple, pour discerner les caractères successifs du contenu d'une chaîne (littérale ou pas d'ailleurs), est de transformer la chaîne en liste :

```
1 S = "C:\\toto\n\"OK\""
2 print(S)
3 print()
4
5 L= list(S)
6 print(len(L))
7 print(L)
```

```
8 C:\toto
9 "OK"
10
11 12
12 ['C', ':', '\\', 't', 'o', 't', 'o', '\n', '"', 'O', 'K', '"']
```

— Lignes 2 et 8-9 : on lit bien la chaîne S.

— Ligne 11 : on retrouve que la chaîne représentée en Python par S contient 12 caractères.

— Lignes 5 et 12 : la liste de chaque caractère de S

Contre-oblique en dernier caractère d'une chaîne littérale

Le contenu de ce paragraphe est d'importance secondaire.

Une chaîne littérale dont le contenu se termine par une *unique* contre-oblique est invalide car cette contre-oblique serait vue comme un caractère d'échappement du délimiteur de sortie de chaîne littérale :

```
1 s = "hello\"
2
3 print(s)
```

```
4     s = "hello\"
5                 ^
6 SyntaxError: EOL while scanning string literal
```

— Ligne 1 : s est une chaîne de caractères terminée par une et une seule contre-oblique.

— Ligne 6 : s est forcément une chaîne invalide car le guillemet n'est plus vu comme un délimiteur mais comme un caractère de la chaîne (à cause de l'échappement créé par la contre-oblique) en sorte que s ne possède plus de délimiteur final.

Nombre pair de contre-obliques

Si le contenu d'une chaîne littérale doit se terminer par des contre-obliques, elles doivent figurer en nombre pair dans la chaîne littérale, par exemple :

```
print("hello\\\\\\")
```

```
hello\\\
```

chaque paire représentant en fait une unique contre-oblique en sorte que le délimiteur de chaîne final ne subit aucun échappement.

Lien vers la documentation officielle : raw literal cannot end in a single backslash

L'usage de chaînes brutes peut largement simplifier le code.

Analyse lexicale d'une contre-oblique

Le contenu de cette unité est anecdotique et a surtout pour object de tester sa compréhension des mécanismes d'échappement.

Soit une chaîne littérale contenant une contre-oblique comme les chaînes suivantes :

```
s = "oui\non"
t = "non\oui"
```

Voici par quel principe le caractère contre-oblique est identifié par le parseur Python quand il scanne la chaîne avant l'exécution.

Un point essentiel est le suivant : **le parseur lit la chaîne littérale de la gauche vers la droite**. En particulier, avant de parser la chaîne, il ne sait pas combien de caractères la chaîne représentée contient.

Quand il rencontre un caractère contre-oblique, avant de l'interpréter, il regarde le caractère immédiatement après cette contre-oblique dans la chaîne, disons qu'il s'appelle c. Essentiellement, deux cas se présentent :

— si la contre-oblique et c forment une séquence d'échappement reconnue, elle est interprétée comme une séquence d'échappement, par exemple si c est le caractère n, la séquence \ suivi de n est interprétée comme un caractère *saut de ligne* ;

— sinon, la contre-oblique est interprétée littéralement comme une contre-oblique.

Comparer :

```
s = "oui\non"
t = "non\oui"
u = "oui\\non"
v = "non\\oui"

print(s)
print("-----------")

print(t)
print("-----------")

print(u)
print("-----------")

print(v)
print("-----------")
```

```
oui
on
-----------
non\oui
-----------
oui\non
-----------
non\oui
-----------
```

— Pour s, u et v, la séquence formée de la première contre-oblique et du caractère qui suit immédiatement est une séquence d'échappement : \n pour s et \\ pour u et v.

— Les séquences d'échappement sont interprétées comme telles : un saut de ligne pour s et une vraie contre-oblique pour u et v.

— Pour `t`, la séquence formée de la première contre-oblique et du caractère qui suit immédiatement est `\o` qui n'est pas une séquence d'échappement. Donc, cette séquence est interprétée comme les deux caractères littéraux : une contre-oblique suivie de la lettre **o**.

Caractères `\b` et `\r`

Le combinaison de caractères `\b` effectue un déplacement arrière du curseur d'écriture et peut donc avoir un effet d'effacement, un peu comme appuyer sur la touche du clavier `Retour Arrière` :

```
1 print("Hello worL\bd!")
```

```
2 Hello word!
```

— Ligne 2 : observez que la lettre *L* a été écrasée.

La lettre *b* du caractère fait référence au nom de *backspace* (*retour arrière* en anglais). On rencontre peu en pratique le recours à ce caractère. Il ne s'agit pas véritablement d'un caractère d'effacement, mais plutôt de retour-arrière du curseur, car il n'y a pas forcément d'effacement :

```
1 print("Hello worLd\b\b")
```

```
2 Hello worLd
```

Le caractère `\r` (*carriage return* autrement dit *retour chariot*) à pour effet de reprendre l'affichage au début de la même ligne et a donc un effet d'écrasement :

```
print("Hello\nDear\rWorld")
```

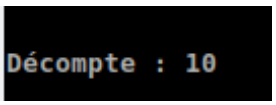
```
Hello
World
```

Comme on peut le voir dans la sortie, la présence du caractère `\r` a fait reculer le curseur d'écriture des caractères jusqu'au début de la ligne où le curseur se trouvait, donc juste après le caractère saut de ligne `\n`, ce qui a eu pour effet d'**écraser** la chaîne `Dear`.

Ce caractère est parfois utilisé pour placer des affichages successifs sur une même ligne :

```
1 import time
2
3 for i in range(10):
4     print(f"Décompte : {i+1}", flush=True, end='\r')
5     time.sleep(1)
```

qui affiche



où on voit que les affichages restent toujours sur la même ligne.

La fonction `repr` appliquée à une chaîne

Si `X` est une chaîne alors `repr(X)` est aussi une chaîne mais *toujours* différente de `X` car `repr(X)` est une chaîne dont le contenu est une chaîne littérale et donc le premier caractère de `repr(X)` est toujours un délimiteur de chaîne littérale :

```
1 X = "bonjour"
2 Y = repr(X)
3
4 print(X[0])
5 print(X)
6 print()
7 print(Y[0])
8 print(Y)
```

```
9 b
10 bonjour
11
12 '
13 'bonjour'
```

— Lignes 4 et 9 : `X` est une chaîne littérale mais le premier caractère de `X` n'est pas un délimiteur de chaîne mais la lettre `b`.

— Lignes 7 et 12 : le premier caractère de `Y` est un délimiteur (ici une apostrophe), le dernier aussi d'ailleurs.

Pour représenter le délimiteur de chaîne litérale, il semble que l'interpréteur Python, s'il a le choix, préfère l'apostrophe au guillement.

Construire une chaîne littérale avec `repr`

La fonction `repr` permet aussi de définir une chaîne littérale de contenu une chaîne donnée `X`.

```
1 APOS = "'"
2 GUIL = '"'
3 SDL = '\n'
4
5 X = APOS *3 + GUIL + SDL+ APOS + SDL + APOS * 3
6 Y= repr(X)
7
8 print("--------- La chaîne X : -----------")
9 print(X)
10 print("-- Chaîne X littérale valant X : --")
11 print(Y)
```

```
12 --------- La chaîne X : -----------
13 '''"
14 '
15 '''
16 -- Chaîne X littérale valant X : --
17 '\'\'\'"\n\'\n\'\'\''
```

— Ligne 5 : `X` est une chaîne bâtie à partir de caractères « spéciaux »
— Ligne 6 : `Y` est une chaîne qui est un équivalent sous forme d'une *chaîne littérale* Python de contenu `X`

Si on copie-colle la chaîne affichée ligne 17, sans rien ajouter (même pas de quotes extérieures) dans du code Python, on obtient une chaîne littérale qui à même valeur que `X` :

```
1 APOS = "'"
2 GUIL = '"'
3 SDL = '\n'
4
5 X = APOS *3 + GUIL + SDL+ APOS + SDL + APOS * 3
6 Y= '\'\'\'"\n\'\n\'\'\''
7 print(X == Y)
```

```
8 True
```

— Ligne 6 : on a copié-collé textuellement l'affichage produit par `print(repr(X))` obtenu au code précédent
— Ligne 8 : Python nous dit que les objets sont égaux : c'est **le point** à comprendre concernant la fonction `repr`.

Transformer une chaîne en une chaîne littérale

Le contenu de ce paragraphe est peu utile en pratique et d'importance secondaire.

Présentons le problème à travers un exemple. Soit la chaîne `s` de *contenu* suivant :

```
"oui"\non'
```

Cette chaîne **ne doit pas** être vue comme une chaîne littérale. Cette chaîne comporte 10 caractères. Le guillemet initial n'est pas à interpréter comme un délimiteur de chaîne littérale Python, ce guillemet est simplement le premier caractère de la chaîne `s`. Aucune séquence de la chaîne `s` n'est à interpréter comme une séquence d'échappement, par exemple la séquence `\n` NE représente PAS le caractère saut de ligne mais bien DEUX caractères consécutifs de `s` : la contre-oblique suivie du caractère `n`.

On cherche à créer une *chaîne littérale* Python `S` dont le contenu soit celui de `s`. Cela pourrait se faire facilement avec des chaînes brutes mais il n'en sera pas question pour l'instant. Ainsi, une fois `S` déterminée, l'extrait de code

```
print(S[0])
print(S[5], S[6])
```

devra afficher

```
"
\ n
```

Il existe au moins deux méthodes pour résoudre ce problème :

— génération de `S`, caractère par caractère, par analyse gauche-droite de `s`
— génération de `S`, après placement de `s` dans un fichier texte et application de la fonction `repr`

Chaîne vers chaîne littérale par analyse gauche-droite

Voyons la première méthode. Le principe de construction de S, assez laborieux, est le suivant :

— on parcourt la chaîne s de la gauche vers la droite, caractère par caractère.
— si le caractère examiné c n'est pas une contre-oblique ni un guillemet, ce caractère produira le même caractère c dans S.
— si le caractère c est une contre-oblique dans s, ce caractère produira les deux caractères \\ dans S
— si le caractère c est un guillemet dans s, ce caractère produira les deux caractères \" dans S
— si le caractère c est une apostrophe dans s, ce caractère produira les deux caractères \' dans S.

Avec cette méthode, on obtient la chaîne littérale S suivante :

```
# "oui"\non'

S = "\"oui\"\\non\'"
print(S)
```

```
"oui"\non'
```

Chaîne vers chaîne littérale en utilisant la fonction repr

Voyons la 2e méthode, elle est un peu laborieuse aussi :

— on construit une chaîne t (non littérale) dont le contenu soit s
— on applique la fonction built-in repr à t ce qui renvoie la chaîne cherchée S

Pour construire t, on place textuellement s (par copier-coller) dans un fichier texte. Ensuite, pour récupérer le texte sous forme de chaîne Python t, on ouvre le fichier texte avec open et on capture le texte avec la méthode read.

repr(t) renvoie une représentation de t sous forme de code-source exécutable, autrement dit, la chaîne littérale cherchée S est repr(t). Pour obtenir la chaîne littérale cherchée, il suffit donc d'afficher avec la fonction print l'objet repr(t) et de faire un copier-coller dans le code-source de ce qui s'affiche à l'écran.

Voici un code complet. On suppose que la chaîne s a été placée dans le fichier chaine.txt :

```
"oui"\non'
```

Le code Python est alors :

```
1 fichier = open("chaine.txt")
2 t = fichier.read()
3 z=repr(t)
4 print(z)
```

```
5 '"oui"\\non\''
```

Vérifions par comparaison visuelle que la chaîne littérale affichée ci-dessus a bien pour contenu la chaîne s initiale :

```
1 s = '"oui"\\non\''
2 print(s)
```

```
3 "oui"\non'
```

— Ligne 1 : `s` est obtenue par copier-coller de l'affichage ligne 5 du précédent code Python.

— La sortie montre que le contenu de `s` est bien la chaîne initiale `S`.

Cette méthode s'appliquerait encore si la chaîne `S` s'étendait sur plusieurs lignes.

Chaînes brutes

Notion de chaîne brute

Une *chaîne brute* (en anglais, *raw string*) est une chaîne **littérale** préfixée par `r` ou par `R` et qui retire à la contre-oblique sa propriété de caractère d'échappement :

```
1 s = "oui\non"
2 print(s)
3 print()
4
5 r = r"oui\non"
6 print(r)
```

```
7 oui
8 on
9
10 oui\non
```

— Ligne 1 : chaîne littérale habituelle ; la séquence `\n` est une séquence d'échappement et est interprétée comme le caractère *saut de ligne*

— Ligne 5 : une chaîne brute reconnaissable à ce que la chaîne littérale est précédée du préfixe `r`.

— Ligne 10 : lorsque la chaîne est brute, la suite de deux caractères `\n` n'est pas interprétée comme une séquence d'échappement mais est interprétée littéralement, comme **deux** caractères : la contre-oblique suivie du caractère `n`, d'où l'absence de saut de ligne.

Pourquoi les chaînes brutes ?

Les chaînes littérales (non brutes) ne sont pas adaptées à la saisie de texte contenant de nombreuses occurrences du caractère d'échappement `\`. Par exemple soit une chaîne littérale `s` dont le contenu `C` est

```
1 une apostrophe : \t\'\nun slash : \t\t\\\nun saut de ligne : \t\\n
```

autrement dit `s` est une chaîne littérale telle que `print(s)` affiche le contenu ci-dessus. Ainsi, il ne faut pas considérer ici la sous-chaîne `\t` de `C` comme une séquence d'échappement mais bien comme **deux** caractères successifs de `C`.

Le code ci-dessous fournit une chaîne littérale `s` dont le contenu est `C` :

```
s = 'une apostrophe : \\t\\\'\\nun slash : \\t\\t\\\\\\nun saut de ligne : \\t\\\\n'
print(s)
```

```
une apostrophe : \t\'\nun slash : \t\t\\\nun saut de ligne : \t\\n
```

On voit que chaque contre-oblique a dû être doublée et que l'apostrophe a dû être remplacée par sa séquence d'échappement. C'est pour éviter ce type de complication qu'existent les *chaînes brutes*.

Usage des chaînes brutes

Une chaîne brute sert essentiellement à faciliter la construction de chaînes littérales représentant des chaînes de caractères contenant une contre-oblique *comme caractère de la chaîne*. Par exemple, on utilise des chaînes brutes pour

— saisir des chaînes littérales à utiliser avec des expressions régulières, c'est un usage **très répandu**, commode et presque obligatoire ;

— produire du code LaTeX dont les commandes commencent par une contre-oblique, par exemple `\begin{document}` ;

— saisir des chemins de fichiers sous Windows, comme `C:\User\monDoc` ;

— sans doute bien d'autres situations.

Les délimiteurs d'une chaîne brute

Une chaîne brute peut avoir pour délimiteur n'importe quel délimiteur de chaîne littérale, y compris des triples quotes :

```
s = r"""oui\non
oui\non
oui\non"""
print(s)
```

```
oui\non
oui\non
oui\non
```

Construire une chaîne littérale complexe en passant par une chaîne brute

Soit la chaîne de caractères `C` suivante :

```
une apostrophe : \t\'\nun slash : \t\t\\\nun saut de ligne : \t\\n
```

Cherchons une chaîne littérale `L` qui représente `C`. Pour cela, il suffit de transformer `C` en chaîne littérale brute :

```
r = r'une apostrophe : \t\'\nun slash : \t\t\\\nun saut de ligne : \t\\n'
print(r)
```

```
une apostrophe : \t\'\nun slash : \t\t\\\nun saut de ligne : \t\\n
```

On observe que la création de la chaîne littérale `r` s'est faite sans effort : il a suffi d'entourer le contenu `C` de la chaîne par une paire d'apostrophes, le premier quote étant précédé du marqueur r de chaîne brute (cette solution n'est pas universelle, il aurait pu y avoir un conflit de quotes par exemple).

Chapitre IV

Type de caractères, Unicode

Accéder à certaines familles de caractères

Le module standard `string` (écrire le nom au singulier) donne accès à certaines listes de caractères. Voici quelques familles remarquables de caractères (les majuscules, les minuscules, etc) et qui sont accessibles avec ce module :

```
import string

# les caractères minuscules
minus = string.ascii_lowercase
print(minus)
print()

# les caractères majuscules
maj = string.ascii_uppercase
print(maj)
print()

# les chiffres décimaux
chiffres = string.digits
print(chiffres)
print()

# la ponctuation
punct = string.punctuation
print(punct)
print()

# les chiffres hexadécimaux
hexa = string.hexdigits
print(hexa)
print()
```

```
abcdefghijklmnopqrstuvwxyz
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

0123456789abcdefABCDEF
```

Les listes précédentes peuvent servir en de nombreuses occasions, d'apprentissage en particulier. L'intérêt que présente ici le module string est de dispenser l'utilisateur de la tâche ingrate de saisir sans erreur les listes ci-dessus.

La méthode `isspace`

On appelle caractères blancs (*blank* en anglais) les caractères invisibles qui peuvent être présents dans du texte ; il s'agit essentiellement des caractères suivants :

— l'espace,
— le saut de ligne
— la tabulation horizontale
— la tabulation verticale (rare)

La méthode `isspace` sert à tester si une chaîne est formée uniquement de caractères « blancs » (noter qu'il y a deux *s* dans le nom de la méthode). Par exemple, c'est le cas de la chaîne littérale ci-dessous :

```
1 s = "\n  \t\n  "
```

Voici un exemple de tests :

```
1 s = "\n  \t\n  "
2 print(s.isspace())
3
4 s = "\n  \t 0 \n  "
5 print(s.isspace())
6
7 s= ''
8 print(s.isspace())
```

```
9 True
10 False
11 False
```

On notera (lignes 11 et 15) qu'une chaîne vide n'est pas considérée comme une chaîne formée de blancs.

Changer la casse d'une chaîne

On dispose d'une chaîne `s` et on veut créer une nouvelle chaîne formée des mêmes caractères mais tous écrits en majuscules, par exemple `orange` deviendrait `ORANGE`. Python dispose de la méthode de chaîne `upper` permettant de changer les caractères en majuscules :

```
1 s = "orange"
2 t = s.upper()
3
4 print(s)
5 print(t)
6 print(s)
```

```
7 orange
8 ORANGE
9 orange
```

— Ligne 2 : une chaîne ayant les mêmes caractères que ceux de `s`, mais en majuscule, est créée.

— Ligne 6 : La chaîne `s` est préservée par cette opération (c'était forcé, les chaînes sont immuables). La chaîne `t` est une deuxième chaîne.

Après application de la méthode `upper` sur une chaîne `s`, les caractères suivants figurant dans `s` restent inchangés :

— un caractère déjà en majuscule

— un caractère dépourvu de majuscule

Ainsi :

```
s = "42 Oranges !"
t = s.upper()
print(s)
print(t)
```

```
42 Oranges !
42 ORANGES !
```

De même, si `s` est une chaîne, `s.lower()` renvoie la chaîne ayant mêmes caractères que ceux de `s` mais en minuscule :

```
s = "to MARS in 2040"
t = s.lower()
print(s)
print(t)
```

```
to MARS in 2040
to mars in 2040
```

La méthode `capitalize` et `title`

Voici un exemple d'action de la méthode `capitalize` :

```
s = """le langage Python
créé par GuiDo"""
print(s)
print("-----------")
print(s.capitalize())
```

```
le langage Python
créé par GuiDo
-----------
Le langage python
créé par guido
```

La méthode `capitalize` appliquée à une chaine « change »

— en lettre majuscule le premier caractère de la chaîne si ce caractère est une lettre minuscule,
— en lettre miniscule toutes les majuscules suivantes.

Le changement en lettre majuscule ne s'applique qu'à la première lettre et non pas à chaque début de ligne.
La méthode `title` est une méthode qui change

— en lettre majuscule le début de chaque **mot** d'une chaîne
— en lettre minuscule toute lettre majuscule qui n'est pas une lettre initiale d'un mot de la chaîne.

Par exemple :

```
s = """le langage Python
créé par GuiDo"""
print(s)
print("-----------")
print(s.title())
```

```
le langage Python
créé par GuiDo
-----------
Le Langage Python
Créé Par Guido
```

Centrer du texte

La méthode `center` appliquée à une chaîne permet de centrer cette chaîne en l'entourant de caractères donnés dans un bloc de longueur donnée. Par exemple

```
s="bonjour".center(20, ".")
print(s)
```

```
......bonjour.......
```

Pour que le texte soit entouré d'espaces, ne pas donner de 2^{e} argument :

```
s="bonjour".center(20)
print(s)
```

```
      bonjour
```

Cette méthode permet de construire facilement le motif suivant :

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
*******************
          #
          #
```

Le code est :

```
n=10
print('\n'.join([("*"*(2*i+1)).center(2*n)for i in range(n)]))
print("#".center(2*n))
print("#".center(2*n))
```

Les méthodes `maketrans` et `translate`

La méthode de chaîne `translate` permet de transformer une chaîne en une autre par la conversion individuelle de certains de ses caractères. Par exemple, soit à transformer un texte en remplaçant

- la lettre `a` par la lettre `X`
- la lettre `b` par la lettre `Y`
- la lettre `c` par la chaîne `XY`

Ainsi, la chaîne `bacdaacbdc` sera convertie en `YXXYdXXXYYdXY`. Voici comment on réalise cette traduction avec la méthode `translate` :

```
1 between = {ord("a"):"X", ord("b"):"Y", ord("c"):"XY"}
2
3 s="bacdaacbdc"
4 print(s)
5 t=s.translate(between)
6 print(t)
7 print(s)
```

```
8 bacdaacbdc
9 YXXYdXXXYYdXY
10 bacdaacbdc
```

- Lignes 3 et 8 : la chaîne à convertir
- Lignes 5 et 9 : la conversion
- Ligne 10 : la chaîne initiale, bien sûr, n'est pas modifiée
- Ligne 1 : la table de conversion qui est un dictionnaire dont chaque clé est la valeur unicode du caractère à transformer ; la valeur d'un caractère unicode `c` s'obtient par `ord(c)`.

— Ligne 5 : la table de conversion est transmise en argument à la méthode `translate`

Attention, écrire le dictionnaire sous la forme suivante :

```
between = {"a":"X", "b":"Y", "c":"XY"}
```

bien que n'entraînant aucune erreur d'exécution, ne donne pas le résultat attendu (`translate` renverra la chaîne `s` telle quelle, sans modification).

Si la valeur associée à une clé `k` du dictionnaire est :

— `None` alors le caractère correspondant à `k` sera écrasé (éliminé)

— un entier `n` alors le caractère correspondant à `k` sera remplacé par le caractère dont le code unicode est `n`, autrement dit le caractère `chr(n)` où `chr` est une fonction standard Python.

Une clé du dictionnaire peut aussi être un code unicode valide, disons `k`. Ce code correspond au caractère `chr(k)`. Ainsi :

— si `k = 42` alors le caractère correpondant est `*`

— si `k = 100` alors le caractère correpondant est `d`

Illustration de ces différentes possibilités :

```
1 between = {ord("a"):"X", ord("b"):"Y", ord("c"):42, 100: None}
2
3 s="bacdaacbdc"
4 print(s)
5 t=s.translate(between)
6 print(t)
```

```
7 bacdaacbdc
8 YX*XX*Y*
```

— Lignes 1 et 8 :

  — le caractère `c` sera remplacé par le symbole `*` dont le code unicode est 42 ;

  — le caractère `d` (qui a pour code unicode 100) sera écrasé puisque la valeur de la clé est `None`.

Plus généralement, si `s` est une chaîne et si `between` est une table de conversion de caractères vers des chaînes, alors `s.translate(between)` crée une chaîne où chaque caractère de `s` présent dans la table est remplacé dans la chaîne par sa valeur dans la table. La table de conversion n'est pas clairement définie par la documentation Python. En pratique, la table semble se présenter sous deux formes :

— un dictionnaire dont les clés sont comme il est expliqué ci-dessus ;

— une table obtenue par la méthode de chaîne `maketrans`, détaillé ci-dessous.

Création d'une table de conversion avec `maketrans`

La fonctionnalité essentielle de `maketrans` est de renvoyer une « table de conversion » utilisable par la méthode `translate`. L'exemple qui suit montre un exemple typique d'utilisation de cette méthode. Imaginons que l'on veuille créer une table de conversion de caractères où

— la lettre `a` est convertie en la lettre `X`

— la lettre `b` est convertie en la lettre `Y`

— la lettre c est convertie en la lettre Z

Voici comment maketrans réalise cette table :

```
1 source = 'abc'
2 cible = 'XYZ'
3
4 trans = str.maketrans(source, cible)
5 print(trans)
6
7 s="baebcdfa"
8 t=s.translate(trans)
9 print(s)
10 print(t)
```

```
11 {97: 88, 98: 89, 99: 90}
12 baebcdfa
13 YXeYZdfX
```

— Lignes 1-2 : on définit deux séquences de caractères de même longueur. La séquence source contient les caractères qui seront remplacés. La séquence cible contient les caractères de remplacement. Les caractères se correspondent positionnellement.

— Lignes 4 et 11 : maketrans renvoie un dictionnaire qui met en correspondance les caractères des deux séquences données en arguments. Les clés du dictionnaire (ligne 11) sont les valeurs unicodes des caractères de la 1re séquence et les valeurs des clés sont les valeurs unicodes des caractères de remplacement.

— Ligne 8 : ce dictionnaire est alors utilisable par la méthode translate.

— Ligne 8 : l'usage est de faire un appel à la méthode maketrans sous la forme str.maketrans puis en donnant en argument les deux séquences de caractères. Pour information, la méthode maketrans est une *méthode statique* : on peut l'appeler comme méthode de n'importe quelle chaîne, le résultat sera toujours le même et elle ne reçoit pas la chaîne d'où on l'applique comme premier argument (ce qui le cas des méthodes d'instance qui reçoivent self).

Dans l'usage précédent, maketrans peut recevoir un 3e argument z qui est une séquence de caractères. Ces caractères deviendront clés du dictionnaire renvoyé trans et auront pour valeur None dans le dictionnaire, ce qui signifie que l'appel s.translate(trans) écrasera tous les caractères de s figurant dans z. Voici un exemple basé sur le précédent où, en plus, les lettres d et e sont écrasées :

```
1 source = 'abc'
2 cible = 'XYZ'
3 perdus = "de"
4
5 trans = str.maketrans(source, cible, perdus)
6 print(trans)
7
8 s="baebcdfa"
9 t=s.translate(trans)
10 print(s)
11 print(t)
```

```
12 {97: 88, 98: 89, 99: 90, 100: None, 101: None}
13 baebcdfa
14 YXYZfX
```

— lignes 3, 5 et 12 : les entiers `100` et `101` sont les valeurs unicodes des caractères `d` et `e` et dans le dictionnaire `trans`, leurs valeurs sont `None`.
— lignes 13 et 14 : observer que les lettres `d` et `e` ont disparu.

Identificateurs unicodes

Par défaut, l'interpréteur Python suppose que le code source est écrit dans l'encodage UFT-8. Un identificateur de variable ou de fonction présent dans le code-source peut utiliser des caractères UTF-8 autres que les caractères alphanumériques ASCII :

```
R = 10
 = 3.14

périmètre = 2 *  * R
print(périmètre)
```

```
62.800000000000004
```

Dans le code ci-dessus, les identificateurs et `périmètre` utilisent des caractères autres que des caractères alphanumériques ASCII. Toutefois, ce sont des caractères Unicode de la catégorie *lettre*. En effet, un caractère d'un identificateur autre qu'un chiffre ou le caractère _ doit être de la catégorie lettre, ce qui n'est pas le cas par exemple du symbole monétaire de l'euro :

```
€ = 1.2
print(€)
```

```
  File "invalid_identif_unicode.py", line 1
    € = 1.2
    ^
SyntaxError: invalid character in identifier
```

Les fonctions `chr` et `ord` de conversion unicode

Deux fonctions built-in, `chr` et `ord`, permettent de passer d'un caractère à son code Unicode :

— un appel `chr(n)` renvoie le caractère ayant l'entier `n` pour cod e Unicode
— un appel `ord(c)` renvoie le code Unicode d'un caractère unicode `c` donné.

Voici quelques exemples restreints au code ASCII :

```
print(chr(81))
print(chr(42))
print(ord('*'))
```

```
Q
*
42
```

et des exemples hors du code ASCII :

```
print(ord("€"))
print(chr(8364))
```

```
8364
€
```

Parcours de tous les caractères unicodes

Les valeurs décimales c des codes unicodes vérifient $0 \leq c < 1114112$. Il est donc possible, à l'aide d'un programme Python, de rechercher exhaustivement des propriétés des différents caractères en parcourant la totalité des caractères possibles. Par exemple, le programme ci-dessous teste le nombre de caractères unicodes reconnus comme des chiffres décimaux :

```
cpt=0

for i in range(1_114_112):
    c=chr(i)
    if c.isdecimal():
        cpt += 1
print(cpt)
```

```
580
```

Comparaison alphabétique de chaînes unicodes

La comparaison alphabétique par défaut est purement informatique, elle ne tient compte que de l'ordre des codes des caractères Unicodes. Mais l'ordre alphabétique linguistique est différent et possède des règles spécifiques à chaque région (du monde) et qui sont gérées en Python, par un module standard de régionalisation appelé `locale`. Voici les règles spécifiques pour le français de France : $\mathrm{fr}_F R$. La fonction `strcoll` du module `locale` permet de comparer des chaînes en tenant compte des particularités linguistiques (majuscule, accent, caractères spéciaux). Voici un exemple d'utilisation :

```
1  import locale
2
3  ma_locale = locale.setlocale(locale.LC_ALL, "")
4  print(ma_locale, end="\n\n")
5
6  def comparer(x, y):
7      c = locale.strcoll(x, y)
8      if c<0:
9          print(x, "<", y)
10     elif c>0:
11         print(y, "<", x)
12     else:
13         print(y, "=", x)
14     print("-"*15)
15
```

```
16 print(f"Hors locale: oranger < Oranges : {'Oranges' < 'oranger'}")
17 print("-"*15)
18 comparer("Oranges", "oranger")
19 comparer("orangés", "Orange")
20 comparer("Orange", "orange")
```

```
21 fr_FR.UTF-8
22
23 Hors locale: oranger < Oranges : True
24 ---------------
25 oranger < Oranges
26 ---------------
27 Orange < orangés
28 ---------------
29 orange < Orange
30 ---------------
```

— Lignes 3-4 et 21 : Tous les usages linguistiques (cf. `LC_ALL`) du domaine `fr_FR` (français de France) seront pris en compte.

— Ligne 16 et 23 : Unicode « pur » considère, par défaut, que *Oranges* est avant *oranger* (les majuscules sont classées avant)

— Ligne 25 : en français, *oranger* est avant *Oranges*.

— Ligne 27 : le module `locale` sait gérer le placement des accents

— Ligne 28 : En français, les minuscules viennent avant les majuscules.

Pour **trier une liste** de chaînes en respectant l'ordre alphabétique défini par la locale, voir Tri de chaînes et particularités linguistiques dont l'exemple est repris ci-dessous :

```
import locale
locale.setlocale(locale.LC_ALL, "")

L = ["Prune", "prune", "levées", "lève-glaces",
     "lèvent", "leçon", "Lecteur"]

print(sorted(L, key=locale.strxfrm))
```

```
['leçon', 'Lecteur', 'levées', 'lève-glaces',
'lèvent', 'prune', 'Prune']
```

Catégorie d'un caractère unicode

L'ensemble des caractères unicodes est partitionné en une trentaine de catégories telles que

lettre, symbole, nombre, ponctuation

puis en sous-catégories comme

lettre majuscule, symbole monétaire, ponctuation fermeture, ponctuation ouverture, etc

Par exemple, le symbole € est dans la catégorie `[Symbol, Currency]`.

La liste des catégories est disponible ICI. Cette liste est uniquement en anglais.

Les catégories ne sont pas liées à une langue spécifique. Par exemple, bien que la lettre ne soit pas dans l'alphabet latin, elle est dans la même catégorie *Letter, Lowercase* que, par exemple, la lettre *a*.

Les catégories sont abrégées avec les initiales ; par exemple, la catégorie *Letter, Uppercase* est abrégée `Lu`.

Caractère unicode dans une chaîne littérale

Il est possible d'insérer un ou plusieurs caractères unicode dans une chaîne littérale d'un code-source Python.

Plusieurs techniques d'insertion sont possibles, visibles dans le code ci-dessous :

```
print("Périmètre = 2 \N{greek small letter pi} R")
print("Périmètre = 2 \u03C0 R")
print("Périmètre = 2  R")
```

```
Périmètre = 2 π R
Périmètre = 2 π R
Périmètre = 2  R
```

Noter que dans les deux premiers exemples, on utilise un échappement pour placer le caractère, pas dans le 3e exemple. L'affichage peut dépendre de la nature de la console, en particulier, si l'encodage de la console n'est pas en UTF-8, il se peut que certains caractères ne soient pas lisibles. Ci-dessous, on donne quelques précisions sur les 3 procédés employés.

Caractère unicode défini par son nom

Un caractère unicode possède un nom standard. Ce nom est en anglais, par exemple *greek small letter pi* pour la lettre grecque (cf. ligne 1 du code ci-dessus). Il est possible de se référer dans une chaîne littérale à un caractère unicode par son nom normalisé en utilisant la séquence d'échappement

`\N{nom_unicode}`

où `nom_unicode` désigne le nom standard. Il doit s'agir du nom exact mais la casse du nom n'a pas d'importance, par exemple `\N{Greek small letter PI}` sera reconnue exactement comme `\N{greek small letter pi}`.

Caractère unicode défini par son code hexadécimal

Chaque caractère unicode possède un unique encodage UTF-8 sous la forme d'un nombre codé avec 4 chiffres hexadécimaux et qui peut commencer par des zéros. Par exemple, la lettre grecque possède le code UTF-8 suivant : `03C0`.

Il est possible de se référer dans une chaîne littérale à un caractère unicode par son code UTF-8 avec la séquence d'échappement

`\u{code_hexa}`

où `code_hexa` est une suite de 4 chiffres hexadécimaux. Par exemple, le caractère est défini par la séquence `\u03C0`, cf. ligne 2 du code ci-dessus.

En particulier, on peut écrire sous cette forme les caractères ASCII, par exemple, l'astérisque `*` a pour code ASCII hexadécimal le nombre `2A`, ce qui donne en unicode littéral :

```
ast = "\u002A"

print(ast * 10)
```

```
**********
```

Caractère unicode inséré littéralement

Si l'éditeur de code supporte l'encodage UTF-8, au lieu d'écrire le caractère à l'aide d'une séquence d'échappement, il est possible d'insérer le caractère directement, via un copier-coller du caractère, soit par un code propre à l'éditeur de code soit par insertion du caractère après sélection à la souris dans un panneau (cf. ligne 3 du code ci-dessus).

Le module `unicodedata`

Les caractères unicodes ont chacun un certain nombres de propriétés comme leur nom ou leur catégorie. Le module `unicodedata` donne accès à ces propriétés.

Ci-dessous, ne sont décrites que les fonctions du module liées aux propriétés de nom et de catégories.

Propriété de nom

Le module `unicodedata` permet d'obtenir le nom d'un caractère unicode à partir du caractère lui-même et inversement :

```
1 import unicodedata
2
3 euro = "euro sign"
4 c=unicodedata.lookup(euro)
5 print(c)
6
7 name = unicodedata.name("€")
8 print(name)
```

```
9 €
10 EURO SIGN
```

— Ligne 3 : la fonction `lookup(my_name)` renvoie une chaîne dont le seul caractère est celui dont le nom standardisé `my_name` est donné à la fonction `lookup`.

— Ligne 7 : la fonction `name(my_char)` renvoie le nom unicode standardisé du caractère `my_char`.

Noter que certains caractères n'ont pas de nom, par exemple, certains caractères de contrôle comme le saut de ligne LINE FEED, de code ASCII valant le nombre hexadécimal `0A` :

```
import unicodedata

print(unicodedata.name("\n"))
```

```
    print(unicodedata.name("\n"))
ValueError: no such name
```

Catégorie d'un caractère

Le module `unicodedata` donne accès à la catégorie avec l'attribut `category`

```
import unicodedata

for s in "Aaé\N{greek small letter alpha}+\N{infinity}€":
    print(s, unicodedata.category(s))
```

```
A Lu
a Ll
é Ll
 Ll
+ Sm
 Sm
€ Sc
```

— Le caractère `A` est catégorisé `Lu` pour *Letter, Uppercase.*

— Le caractère `a` est catégorisé `Ll` pour *Letter, Lowercase*

— Le caractère `+` est catégorisé `Ll` pour *Symbol, Math*

— Le symbole Euro est catégorisé `Sc` pour *Symbol, Currency.*

Caractères littéraux en octal ou en hexadécimal

Une chaîne littérale peut contenir des caractères unicodes définis par leur code octal (en base 8) ou leur code hexadécimal (en base 16). Voici un exemple :

```
1 s = "symétrie : \616 | E"
2 p = "2 \x2A 3 = 6"
3 print(s)
4 print()
5 print(p)
```

```
6 symétrie :  | E
7
8 2 * 3 = 6
```

— Ligne 1 : un caractère donné sous forme de code octal

— Ligne 2 : un caractère donné sous forme de code hexadécimal

Les codes octal ou hexadécimal sont des séquences d'échappement donc précédés d'un contre-oblique.

Le code octal

Le code octal peut contenir un, deux ou trois chiffres entre 0 et 7. Un code octal peut donc coder jusqu'à $8^3 = 512$ caractères. Par défaut, il semble que ce soit le caractère unicode de même valeur qui soit renvoyé, ce que l'on peut tester en utilisant la fonction standard `chr` :

```
print("\616" == chr(0o616))
```

```
True
```

Si une contre-oblique précède une suite de chiffres plus grand que 8, ce n'est pas une erreur et la contre-oblique n'est pas vue comme caractère d'échappement :

```
print("\81")
```

```
\81
```

Le code hexadécimal

Le code hexadécimal commence par la séquence \x et est suivi d'exactement deux chiffres hexadécimaux. Si le code hexadécimal ne possède qu'un seul chiffre, il faut le faire précéder d'un zéro :

```
1 print("kiwi\x0Arose")
```

```
2 kiwi
3 rose
```

— Ligne 1 : la valeur hexadécimale d'un saut de ligne est `A`, il doit être écrit `0A`.

Le code hexadécimal ne peut donc coder que 256 caractères. Les premiers 256 caractères du jeu unicode correspondent à l'encodage latin-1.

Unicode, UTF-8

Ce qui suit n'est qu'une présentation sommaire d'Unicode.

Unicode

Unicode est un standard international permettant d'uniformiser le traitement informatique des caractères de pratiquement toutes les langues du monde. Unicode définit plus de 110000 caractères utilisés dans une centaine de langues naturelles et dans des domaines techniques spécifiques (symboles, caractères phonétiques, hiéroglyphes, etc).

Point de code

Les caractères sont codés par des nombres entiers n dit "point de code" et qui vérifient $0 \leq n < 1114112$. Ce nombre n est souvent donné en hexadécimal et précédé du préfixe `U+`. Par exemple, le symbole euro a pour point de code `U+20AC`.

Chaque caractère peut être identifié par une appellation normalisée (le *nom* du caractère), par exemple

GREEK SMALL LETTER PI

désigne , la lettre grecque *pi* en minuscule.

On constate que la plage de codage des caractères unicodes est de longueur `1 114 112` et donc est beaucoup plus longue que les 110000 caractères. On a la propriété : $1114112 = 2^16 \times (16+1)$ et donc le nombre de caractères unicodes peut s'écrire, en hexadécimal, sous la forme `0x110000`, cf. Unicode range.

L'UTF-8

UTF-8 est une norme internationale définissant un encodage informatique des caractères unicodes. UTF-8 signifie *Universal Transformation Format 8 bits* et indique comment transformer un point de code unicode en une suite de 1 à 4 octets (un octet étant une séquence de 8 bits).

UTF-8 est compatible avec de plus anciens encodages

— l'ASCII 7 bits

— l'encodage latin-1 sur 8 bits (courant dans les pays d'Europe occidentale),

autrement dit le code latin-1 (et donc ASCII) d'un caractère a même encodage en UTF-8.

En 2021, l'UTF-8 est utilisé dans presque 97% des sites web. L'UTF-8 est aussi le jeu de caractères par défaut sur Android. Depuis mai 2019, Microsoft recommande que tous les logiciels utilisés sur sa plateforme emploient l'encodage UTF-8.

Point de code et encodage UTF-8

Soit `c` un caractère unicode. Il ne faut pas confondre

— le point de code de `c` qui est l'entier figurant dans son code unicode

— l'encodage UTF-8 de `c` qui est une suite d'octets

Par exemple, pour le symbole € :

— point de code : U+20AC

— encodage UTF-8 : les trois octets `0xE2`, `0x82`, `0xAC`.

La plupart du temps, l'utilisateur (y compris le programmeur) n'a pas besoin d'interférer avec les octets d'encodage UTF-8 et interfère avec le caractère via son point de code. L'algorithme de conversion entre code unicode et suite d'octets est présenté en 7 points dans cet article What is UTF-8?.

Documentation :

— Le Unicode HOWTO du langage Python

— Wikipedia : Unicode et UTF-8.

— Livre Fluent Python de Luciano Ramalho, éditeur O'Reilly : le chapitre 4 traite essentiellement d'Unicode sous Python et en entrant dans les détails. Il existe une traduction française de l'ouvrage.

Caractères ASCII

Les caractères ASCII forment une famille de caractères utilisée en informatique et en programmation depuis les années 1960. Avec la généralisation du standard Unicode, ils ont aujourd'hui un rôle beaucoup plus réduit, surtout que Unicode étend l'ASCII tout en restant compatible.

Voici 8 caractères ASCII :

```
Z    t    >    .    ^    }    a    $
```

et voici 8 caractères n'appartenant pas à la famille ASCII :

€ é ç ß ↓

Les caractères ASCII représentent :

— les caractères alphanumériques usuels, majuscules ou minuscules et **non accentués**,
— des caractères de ponctuation,
— des caractères symboles comme $ ou %
— des caractères non imprimables comme *le saut de ligne*
— des caractères de contrôle et qui n'ont pas de signification ni de représentation textuelle particulière.

Les caractères ASCII sont d'origine américaine, l'acronyme signifie *American Standard Code for Information Interchange* ; et donc les caractères ASCII ne sont jamais des caractères accentués. On n'y trouve pas non plus le symbole euro.

On dit parfois qu'un texte utilisant uniquement des caractères ASCII est du *texte pur*.

Il existe exactement 128 caractères ASCII. Noter que $128 = 2^7$: les caractères ASCII sont codés sur 7 bits.

En informatique, les caractères (alphabétiques, chiffres, signe de ponctuation, etc), quand ils sont utilisés dans un cadre de programmation, sont identifiés par un code entier.

Par exemple, dans le codage ASCII, le caractère `P` majuscule est codé par l'entier 80. Les caractères ASCII sont codés par les entiers de 0 à 127.

L'*encodage* est la correspondance entre un jeu de caractères plus ou moins riche et le codage de ces caractères par des entiers. L'ASCII est un encodage parmi d'autres.

Au lieu du terme d'*encodage*, on trouve le terme *jeu de caractères* (*charset* en anglais).

L'encodage ASCII a été très utilisé car établi de longue date (1961), minimal et compatible avec de nombreux systèmes informatiques.

Risque de confusion

Il existe des codages ASCII dit « étendus » et qui ont les deux propriétés suivantes :

— ils englobent les caractères ASCII habituels avec le même codage
— ils contiennent des caractères supplémentaires, des caractères accentués par exemple

À cause de la première propriété, on dit que ces codages sont des **extensions** de l'ASCII. L'encodage Latin-1 et plus spécifiquement l'encodage ISO 8859-15 a longtemps été en usage en Europe occidentale avant l'avènement d'Unicode.

Par ailleurs, il n'existe pas une unique extension du codage ASCII.

Quand on fait référence au codage ASCII, sans précision supplémentaire, on ne fait pas référence à des codages ASCII étendus.

Méthodes isdigit, isdecimal, isnumeric

Les méthodes `isdigit` et `isdecimal` permettent de reconnaître respectivement les chiffres et les chiffres décimaux avec des variantes de présentation. Les caractères reconnus par `isdecimal` forment un sous-ensemble des caractères reconnus par `isdigit`.

La méthode `isnumeric` permet de reconnaître des caractères représentant des nombres.

isdecimal

`isdecimal` reconnaît tous les caractères représentant les chiffres décimaux entre 0 et 9, sous différentes formes et dans différentes cultures. Voici quelques caractères représentant le chiffre 5 et reconnus par la méthode isdecimal :

5

Le dernier caractère de la liste représentant le chiffre 5 dans l'alphabet tamoul.

isdigit

C'est une extension de la famille précédente. Elle comprend en plus certaines représentations de chiffres, par exemple sous forme d'exposant ou stylisés. Voici deux exemples :

isnumeric

Les caractères reconnus par `isnumeric` sont un sur-ensemble des caractères reconnus par `isdigit`. Parmi les caractères non reconnus par `isdigit`, on trouve, par exemple, des nombres romains, des fractions, des nombres stylisés ou des nombres représentés dans des civilisations non occidentales ou d'anciennes civilisations. Voici quelques exemples :

Autres méthodes de reconnaissance de caractères

La méthode `isspace` permet de reconnaître les caractères dits blancs comme l'espace (de code ASCII `0x20`) ou le saut de ligne (code ASCII `0x0A`) ainsi que des caractères servant à définir des espaces en typographie.

La méthode `isalpha` permet de reconnaître des caractères alphabétiques, appartenant à des alphabet occidentaux ou non, des alphabets mathématiques, etc. Voici quelques exemple de caractères reconnus par la méthode isalpha :

C c Ç ü

Les méthodes `isupper` permet de distinguer des versions minuscule et majuscule de caractères reconnus par `isalpha`.

La méthode `isalnum` permet de reconnaître les caractères qui sont soit reconnus par `isalpha`, soit reconnus par `isnumeric`.

La méthode `isprintable` reconnaît les caractères unicodes dits *imprimables*; il s'agit de ceux n'appartenant pas aux catégories *Other* ou *Separator* de l'encodage Unicode. Le caractère saut de ligne, de code ASCII `0x10` est considéré comme non-imprimable de même que tous les caractères de code ASCII entre 0 et 31. Le caractère *espace*, de code ASCII 32, est le premier caractère du code ASCII qui soit imprimable.

Chapitre V

Exercices sur les chaînes de caractères

Entier léger suivant

Un entier strictement positif est dit *léger* si tous ses chiffres sauf le premier sont nuls. Par exemple, les entiers 7000, 40 ou 5 sont légers tandis que les entiers 2200, 50050 ou 42 ne le sont pas.

On donne un entier $1 \leq n \leq 10^9$ et on demande de trouver le plus petit entier léger $p > n$. Par exemple si $n = 42$ alors $p = 50$. On commencera par déterminer le nombre de chiffres de n ainsi que son premier chiffre.

L'exercice a été posé sur codeforces et la suite des entiers « légers » est répertoriée sur le site OIES.

Numéronyme

On va décrire ci-dessous un procédé de compression de mot. On donne une chaîne de caractères, par exemple

```
s = "Kubernetes"
```

et on demande de construire une nouvelle chaîne `S`, dans notre exemple, ce sera

```
S = "K8s".
```

La règle de construction de la chaîne `S` est précisée ci-dessous :

- si `s` a au moins trois caractères :
  - le premier caractère de `S` est identique à celui de `s`
  - le dernier caractère de `S` est identique à celui de `s`
  - les caractères de `S` strictement entre le premier et le dernier caractère de `S` sont remplacés par le nombre d'éléments de `s` **strictement** compris entre les extrémités de `s`
- sinon `S = s`.

Voici quelques exemples de comportement :

```
Kubernetes -> K8s
Python -> P4n
Internationalization -> I18n
globalization -> g11n
```

On aura besoin d'utiliser la fonction `str`.

Calculer une moyenne

On donne une chaîne de caractères représentant une addition de nombres flottants séparés par des signes `+` et d'éventuels espaces et on vous demande de calculer la moyenne des nombres qui apparaisent. La virgule décimale n'est pas écrite à l'aide d'un point mais d'une virgule. Par exemple, si la chaîne est

```
"12,25+  1+10+10,75 + 7+1,25+6,25   + 6,75+13,5+8,5"
```

la moyenne à calculer devrait être `7,725`. On utilisera les méthodes de chaînes `split` et `replace` et le constructeur `float`.

Diagonale de 1

On donne un entier `n` $\geq$ `0`, par exemple `n` $=$ `5` et on demande de construire une chaîne `s` formant un motif comme expliqué ci-dessous. Si `n` $=$ `5` alors `s` s'affichera ainsi :

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

Plus généralement, la chaîne `s` à construire est formée de `n` lignes, chacune de `n` caractères parmi `'0'` ou `'1'` et en sorte que le k^e caractère de la k^e ligne soit `'1'`, les autres valent `'0'`. Entre deux caractères d'une même ligne, il aura un espace.

On utilisera la méthode `join` et les opérateurs `+` et `*` sur les chaînes et éventuellement le constructeur `list`. Vous pouvez aussi utiliser des listes en compréhension si elles vous sont connues.

Noter qu'on demande de construire une chaîne et non pas seulement de réaliser l'affichage du motif.

Il est possible d'écrire le code sur une seule ligne.

Chaîne représentant un motif en forme de croix

On souhaite générer une chaîne de caractère représentant un motif en forme de croix décrit ci-après. On part d'un entier impair $n > 1$ et le motif est de forme carrée, rempli de zéros ou de uns, en sorte que les zéros forment une croix. Par exemple, si $n = 9$, le motif est :

```
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1
```

```
1 1 1 1 0 1 1 1 1
0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1
```

Le motif est un carré formé de $n \times n$ chiffres valant 0 ou 1. Deux chiffres seront toujours séparés d'un unique espace. Les axes vertical et horizontal de symétrie du carré sont formés de 0 et les autres parties du carré sont formées de 1.

Il est attendu de créer une chaîne (et pas seulement d'afficher le motif). On utilisera la méthode `join` et la fonction `list` et les opérations d'addition et d'itération de chaînes (du type, `k * s` où `k` est un entier positif et `s` est une chaîne). On pourra aussi utiliser l'itération de liste comme l'exemple ci-dessous l'explique :

```
k = 5
L = k * ["Noix"]

print(L)
```

```
['Noix', 'Noix', 'Noix', 'Noix', 'Noix']
```

Carrés concentriques de lettres

On demande de dessiner un motif formé de lettres majuscules placées dans l'ordre alphabétique et formant des carrés emboîtés. Voici le motif à dessiner pour $n = 10$:

```
A A A A A A A A A A
A B B B B B B B B A
A B C C C C C C B A
A B C D D D D C B A
A B C D E E D C B A
A B C D E E D C B A
A B C D D D D C B A
A B C C C C C C B A
A B B B B B B B B A
A A A A A A A A A A
```

Le motif est constitué d'une succession de carrés concentriques formés de lettres majuscules toutes identiques et progressant de l'extérieur vers l'intérieur dans l'ordre alphabétique. Entre deux caractères d'une même ligne, il y aura un espace. On demande de créer une chaîne de caractères qui une fois affichée aura l'allure indiquée ci-dessus.

Plusieurs méthodes sont envisageables. On pourra par exemple créer une liste 2D dans laquelle les caractères seront placés individuellement. La liste sera ensuite transformée en chaîne en utilisant, à plusieurs reprises, la méthode join. Il est conseillé de calculer le nombre `N` de carrés concentriques en fonction de `n`, bien qu'il soit aussi possible de s'en passer.

Il est possible de réaliser la construction en un one-liner.

Mettre un nom au pluriel

Ecrire une fonction qui prend en entrée un nom commun de la langue française et renvoie le même mot mais écrit au pluriel. Les règles de mise au pluriel sont les suivantes :

— pour mettre au pluriel, on ajoute un `s` à la fin du mot, sauf exceptions décrites ci-dessous ;
— si le mot se termine par `s`, `z` ou `x`, on ajoute pas de `s` final ;
— les noms se terminant en AIL, en AL ou en AU font leur pluriel en AUX, comme *décimal* qui devient *décimaux*, *travail* qui devient *travaux*, etc. ; il y a cependant une liste d'exceptions signalées ci-dessous ;
— les noms se terminant en EU font leur pluriel en EUX ; il y a cependant une liste d'exceptions signalées ci-dessous ;
— certains mots signalés ci-dessous, comme *bijou*, se terminant en OU, font leur pluriel en OUX.

Exceptions :

— AIL : corail, émail, soupirail, travail, vantail, vitrail
— AL : bal, cal, carnaval, cérémonial, chacal, festival, récital, régal
— AU : landau, sarrau
— EU : bleu, pneu, émeu, lieu
— OU : bijou, caillou, chou, genou, hibou, joujou, pou

Si vous ne connaissez pas les slices, vous pourrez avoir besoin d'écrire une fonction `term(mot, k)` qui renvoie la partie d'un mot formée de ses `k` dernières lettres. Si `k` dépasse la longueur du mot, la fonction renverra le mot en entier.

Découpage en syllabes

Dans cet exercice, on demande d'écrire un programme qui découpe un mot écrit en syllabes. Pour simplifier, les exemples soumis au code seront sans accents et les mots ne contiendront ni apostrophe (comme *aujourd'hui*) ni tiret (comme *chou-fleur*). Certaines exceptions ne seront pas implémentées.

Une syllabe contient toujours un groupe de voyelles (par exemple *ba-teau* contient deux syllabes)

Toutes les voyelles contiguës d'un mot comptent comme une seule et donc une syllabe ne peut couper une suite de voyelles contiguës (par exemple, on découpe *théâ-tre*). Par exemple, dans le mot *pointeur*, la séquence *oi* compte comme une seule voyelle et de même, la séquence *eu* comme une seule voyelle

Certains groupements de consonnes sont insécables, tels que *gn*, *ph*, *th*, *lh*, ou encore des groupements du type *br* ou *fl* où la 2e consonne est un *r* ou un *l*.

La césure se fait entre un groupement de voyelles et un groupement de consonnes (par exemple *a-dieu* ou *bra-ve*) ou entre deux groupements de consonnes (comme *ad-mis*).

On ne laisse pas dans une même syllabe deux blocs consécutifs de consonnes. Lorsqu'on a une succession deux consonnes qui ne forment pas un bloc, on découpe entre les deux consonnes. Lorsqu'on a une succession de trois blocs de consonnes, on coupe entre le 2e et le 3e, par exemple *obs-tiné*.

Quelques références :

— Les coupures de mots à l'écrit
— Tester en ligne le découpage en syllabbe
— Coupure de mots
— Division des mots en syllabes et en fin de ligne de Bertrand Boutin (web archivé).

Afficher le développement par la formule du binôme

On donne un entier `n` ≥ 0 et on demande d'afficher le développement de $(x+y)^n$ par la formule du binôme de Newton.
Par exemple, pour `n=7`, le développement est :

$$(x+y)^7 = x^7 + 7x^6y + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7xy^6 + y^7.$$

Dans l'exercice, l'affichage attendu sera :

```
x^7 + 7x^6y + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7xy^6 + y^7
```

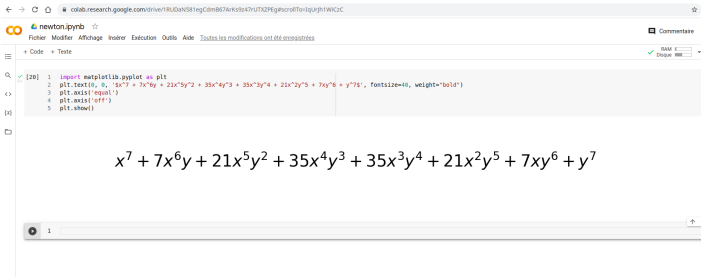
On notera qu'il n'y a pas de facteur 1 et que les puissances d'exposant 0 et 1 sont écrites de manière simplifiée.
On pourra essayer d'obtenir, dans un premier temps, l'affichage approximatif :

```
1x^7y^0 + 7x^6y^1 + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7x^1y^6 + 1x^0y^7
```

On utilisera des f-strings, il n'y a que le minimum à connaître. On utilisera la méthode join pour placer les signes `+`.
Pour traiter le facteur 1 et les puissances d'exposant 0 et 1, on pourra, au choix

— écrire une fonction `monome(c, a, i, b, j)` et qui renverra une chaîne représentant convenablement ca^ib^j ; pour ne pas avoir trop de cas à traiter, on pourra introduire une fonction `puissance(x, k)` représentant convenablement x^k ;
— utiliser la méthode replace mais, ici, c'est plutôt un hack.

L'exercice a été posé sur le forum Python du site OpenClassrooms.
On pourra réaliser un affichage de la formule sous Matplotlib avec la syntaxe demandée plus haut pour obtenir une mise en forme correcte des exposants :



Nombres dont la somme des chiffres vaut 42

Déterminer combien d'entiers existent entre 1 et un million et dont la somme des chiffres de l'écriture en base 10 vaut 42 (on devra en trouver 6062). On utilisera les fonctions natives int et str même si c'est algorithmiquement peu efficace. Si vous les connaissez, vous pouvez utiliser la fonction standard `sum`, une liste en compréhension voire une expression génératrice.

On pourra écrire une fonction `dsum(n)` renvoyant la somme des chiffres de l'écriture en base 10 de l'entier `n`.

Concaténer des entiers consécutifs (méthode de chaînes)

Si on écrit côte-à-côte tous les entiers entre 1 et `n = 42`, on obtient le très grand entier `N` suivant :

```
12345678910111213141516171819202122324252627282930313233343536373839404142
```

On peut vérifier que `N` est un entier composé de $p = 75$ chiffres (il suffit de compter).

Plus généralement, étant donné un entier $n \geq 1$, on place côte-à-côte tous les entiers entre 1 et `n`, écrits en base 10 ce qui construit un très grand entier `N`. On demande de calculer le nombre `p` de chiffres de `N`.

On construira explicitement la représentation de `N` sous forme de chaîne de caractères dont on calculera la longueur.

Mots en lignes, en colonnes (avec la méthode `join`)

On donne une liste de mots, tous de même longueur.

On demande de construire deux chaînes :

- l'affichage de la première chaîne montre les chaînes écrites les une sous-les autres
- l'affichage de la deuxième chaîne montre les chaînes écrites en colonnes séparés de traits verticaux.

Par exemple, si la liste des mots est :

```
mots = ["POIRE", "SAVON", "LAVER", "LABEL"]
```

alors l'affichage horizontal sera :

```
POIRE
SAVON
LAVER
LABEL
```

et l'affichage vertical sera :

```
|P|S|L|L|
|O|A|A|A|
|I|V|V|B|
|R|O|E|E|
|E|N|R|L|
```

Il s'agit de placer les mots dans des colonnes. On notera la présence de traits aux extrémités.

Un trait vertical s'obtient avec la combinaison `Alt Gr + 6`. Noter qu'on demande de produire une chaîne et pas seulement un affichage. On utilisera à plusieurs reprises la méthode `join` afin d'obtenir la chaîne

Nombre de pages à imprimer

Certains utilitaires d'impression permettent à l'utilisateur de définir les pages à imprimer. Par exemple, s'il écrit

3-5, 7, 9, 11-15

les pages à imprimer sont les pages :

3, 4, 5, 7, 9, 11, 12, 13, 14, 15.

ce qui fait un total de 10 pages.

On demande d'écrire un programme Python qui calcule le nombre total de pages à imprimer. Les pages seront données sous la forme d'une chaîne de caractères formée d'un ou plusieurs items séparés par une virgule suivie d'un unique espace. Tout item est de l'une des deux formes suivantes :

— un nombre décimal
— deux entiers décimaux, écrits dans l'ordre croissant et séparés par un simple et unique tiret, sans espaces.

Par exemple, la chaîne littérale Python suivante :

```
"3-5, 7, 9, 11-15"
```

est une présentation valide d'une suite d'items.

Pour séparer suivant les virgules et les tirets, on aura besoin de la méthode `split`. On aura aussi besoin de la fonction `int`.

Afficher une somme de fractions

(1) On donne deux entiers positifs non nuls et on demande d'afficher, en mode texte, la fraction $\frac{a}{b}$. Par exemple, les fractions $\frac{45073}{512}$ et $\frac{45073}{2030}$ devront s'afficher comme suit :

```
45073
-----
 512

45073
-----
2030
```

On centrera au mieux le numérateur et le dénominateur par rapport au trait de fraction. Si l'un deux ne peut être exactement centré, comme c'est le cas de la 2e fraction donnée en exemple, il y aura un blanc de moins à gauche du nombre qu'à droite. Une fraction sera représentée par une liste de trois **chaînes de même longueur** (espaces inclus) :

— le numérateur,
— le trait de fraction écrit avec des tirets
— le dénominateur.

On écrira une fonction `pretty_frac(a, b)`. On ne cherchera pas à simplifier la fraction. Une fraction du type $\frac{a}{1}$ sera représentée par `a` mais la fonction renverra toujours une liste de trois chaînes de même longueur.
Par exemple, `pretty_frac(45073, 42)` renverra la liste suivante :

```
['45073', '-----', ' 42  ']
```

formée de 3 chaînes ayant 5 caractères chacune.

(2) Utiliser la fonction précédente pour afficher la somme de fractions suivante

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

pour un entier `n` > 0 donné. On affichera la fraction simplifiée égale à la somme à calculer.
Par exemple, si `n = 12`, on devra obtenir :

```
    1   1   1   1   1   1   1   1   1    1    1    86021
1 + - + - + - + - + - + - + - + - + -- + -- + -- = -----
    2   3   4   5   6   7   8   9   10   11   12   27720
```

Pour calculer la **valeur** de la somme des fractions, on utilisera le module standard `fractions`. On montre ci-dessous des exemples de ce qu'il faut savoir pour calculer la somme demandée :

```
from fractions import Fraction

# Fraction 2/3
x = Fraction(2, 3)

# Fraction 1/12
y = Fraction(1, 12)
print(x, y)

# on peut calculer une somme de fractions
# avec l'opérateur + comme pour des entiers
s = x + y

# La fraction est fournie simplifiée
print(s)
```

```
2/3 1/12
3/4
```

Afficher une addition

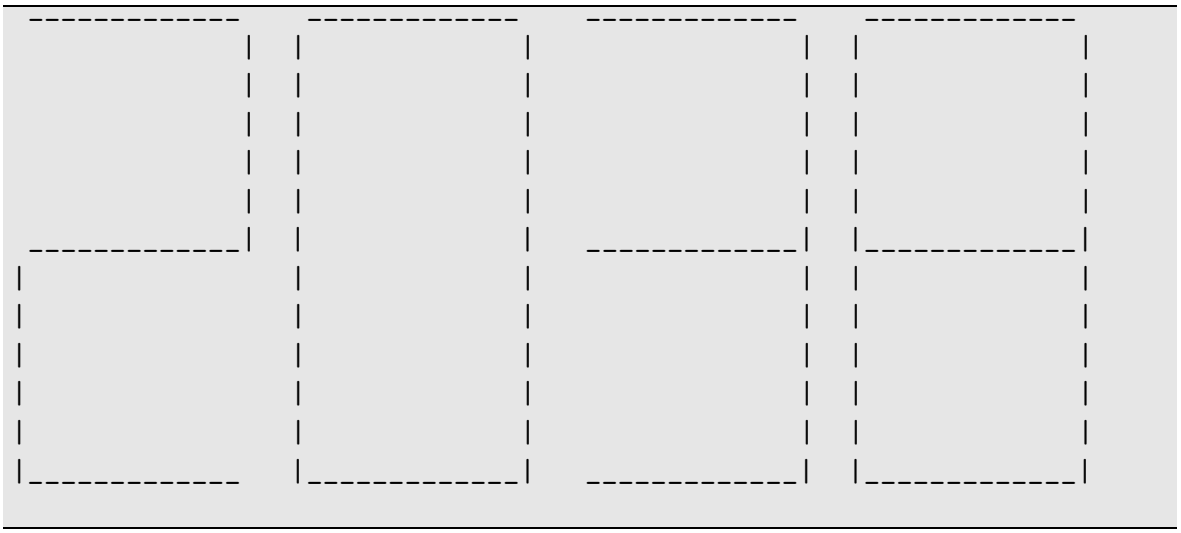
On donne une liste de nombre entiers positifs et on demande d'afficher l'opération d'addition de tous ces entiers et du résultat. Par exemple, si les entiers sont 10, 2038 et 8, l'opération sera affichée comme ci-dessous :

```
   10
+ 2030
+    8
------
  2048
```

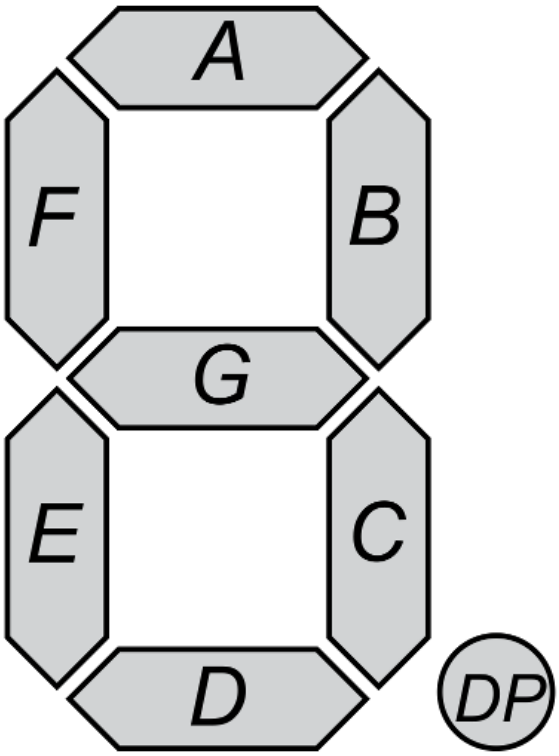
Plutôt que de réaliser un affichage, on produira une chaîne dont l'affichage est l'opération d'addition visible ci-dessus. On utilisera la méthode `join` et la fonction `str`

Affichage à sept segments

L'objectif de l'exercice est d'écrire un ensemble de fonctions permettant de réaliser, en mode texte, un affichage de chiffres avec sept segments. Par exemple, l'entier 2038 sera affiché sous la forme :



Chaque chiffre est représenté en affichant des segments parmi les sept possibles que montre l'image ci-dessous :



Pour la commodité de l'écriture du code, les segments utilisés sont étiquettés par une lettre de A à G (cf. l'image ci-dessus).

Chaque segment horizontal d'un chiffre est construit en utilisant `w` caractères blanc souligné (autrement dit, le caractère _) et chaque segment vertical est construit en utilisant `h` caractères comme ceci | (une barre verticale). Dans l'exemple ci-dessus de l'affichage de 2038, on a choisi `w=13` et `h=6`. Tous les autres caractères (des blancs) sont des caractères *espace*.

Les 7 segments seront étiquettés de A à G suivant la disposition visible ci-dessus. Chaque chiffre reçoit un code formé des lettres entre A et G utilisées pour représenter le chiffre. Par exemple, le chiffre 9 se code avec la chaîne `"GFABCD"` et qui indique les 6 segments affichés.

Pour construire un chiffre, on utilise un liste à deux dimensions formée de caractères, initialement des caractères *espace* Cette liste sera appelée `t`. La taille, lignes x colonnes, de `t` est $(2\texttt{h}+3) \times (\texttt{w}+2)$. Pour coder chacun des chiffres, on utilise son « code » sous la forme d'une chaîne de caractères entre A et G, par exemple le code `"GFABCD"` correpond au chiffre 9 et on change dans la liste `t` les caractères *espace* d'un segment par le caractère de remplissage adapté (_ pour un segment horizontal et | pour un segment vertical). Il est conseillé de faire un schéma indiquant les indices dans la liste `t` de début et de fin de chaque segment. Pour construire un chiffre, on définira une fonction `remplir(code, h, w)` où `code` est la chaîne de caractères entre A et G indiquant les segments utilisés. On écrira aussi une fonction `chiffre(c)` qui renvoie le code d'un chiffre donné `c`.

Enfin, on écrira une fonction `ssd(n)` qui affiche le nombre entier `n` en représentant chaque chiffre avec un affichage à 7 segments. Deux chiffres voisins seront affichés en les séparant d'une colonne de deux espaces.

Mot suivant dans l'ordre alphabétique

On se donne un alphabet formé de lettres distinctes et classées dans l'ordre strictement croissant, par exemple `alpha = "ABIL"`. On donne un mot `m`, composé de lettres uniquement dans l'alphabet donné et on demande de trouver le mot qui suit immédiatement le mot `m` dans l'ordre alphabétique et composé uniquement de lettres parmi l'alphabet donné. Par exemple, avec l'aphabet `"ABIL"`, le mot qui suit `"BAILL"` est `"BALAA"`. De même, le mot qui suit `"LLLLL"` est `"AAAAAA"`. On écrira une fonction `suivant(mot, alpha)`.

Anagrammes par ordre lexicographique

On donne une chaîne `s`. On demande d'écrire une fonction `suivant(s)` qui détermine, le mot obtenu en permutant les lettres de `s` et qui soit immédiatement postérieur à `s` pour l'ordre lexicographique (ie l'ordre du dictionnaire). Par exemple :

— le suivant de "RARBA" est "RBAAR"

— le suivant de "ARRBA" est "BAARR"

— "RRBAA" n'a pas de suivant

En déduire une fonction qui génère tous les anagrammes d'un mot donné. Par exemple, vérifier que la liste des anagrammes de "BARRA" est :

```
['AABRR', 'AARBR', 'AARRB', 'ABARR', 'ABRAR', 'ABRRA', 'ARABR', 'ARARB', 'ARBAR',
'ARBRA', 'ARRAB', 'ARRBA', 'BAARR', 'BARAR', 'BARRA', 'BRAAR', 'BRARA', 'BRRAA',
'RAABR', 'RAARB', 'RABAR', 'RABRA', 'RARAB', 'RARBA', 'RBAAR', 'RBARA', 'RBRAA',
'RRAAB', 'RRABA', 'RRBAA']
```

Vérifier que le mot ABRACADABRA possède 83160 anagrammes.

Ecriture dans une base de l'entier suivant

On donne une base b de numération et une chaîne de caractères `s` représentant un entier n dans la base b. Par exemple, si $n = 1599$ alors

— si `b` $= 10$ alors `s = "1599"`

— si `b` $= 2$ alors `s = "11000111111"`

— si `b` $= 20$ alors `s = "3jj"`

On demande d'écrire une fonction `suivant(n, b)` qui renvoie la chaîne de caractères représentant l'entier $n + 1$ en base b. Par exemple, si $n = 1599$ alors

— si `b` $= 10$ alors `suivant(n, b) = 1600`

— si `b` $= 2$ alors `suivant(n, b) = 11001000000`

— si `b` $= 20$ alors `suivant(n, b) = 400`

On supposera que $2 \leq b \leq 36$ et les chiffres de la base b seront les b premiers caractères de la chaîne formée des chiffres décimaux (de 0 à 9) suivis des lettres alphabétiques minuscules (de `a` à `z`), autrement dit la chaîne :

```
0123456789abcdefghijklmnopqrstuvwxyz
```

Immatriculation suivante

En France, le système d'immatriculation de véhicules (SIV) utilise des immatriculations ayant le motif suivant :

AB-012-CD

autrement dit, de la gauche vers la droite :

— un bloc de deux lettres majuscules,

— un bloc de trois chiffres,

— un bloc de deux lettres majuscules.

Par exemple, CZ-421-UB est une immatriculation valide. Le bloc central 000 n'est jamais utilisé. Pour simplifier, on suppose que toutes les lettres sont utilisées.

Les immatriculations évoluent depuis la 1re immatriculation qui est AA-001-AA jusqu'à la dernière qui est ZZ-999-ZZ. Pour obtenir l'immatriculation qui suit immédiatement une immatriculation donnée, on fait avancer un des trois blocs au bloc suivant avec la priorité définie par :

— le bloc de 3 chiffres,

— puis le bloc de lettres à droite,

— puis le bloc de lettres à gauche.

Le suivant d'un bloc est le suivant dans l'ordre naturel des chiffres ou dans l'ordre alphabétique. Par exemple, le suivant de 421 est 422, le suivant du bloc CE est CF. Le suivant du bloc 999 est 001 et le suivant du bloc ZZ est AA.

Ecrire une fonction qui étant donné une immatriculation renvoie l'immatriculation suivante ou, si l'immatriculation n'a pas de suivante, renvoie la chaîne vide.