9. Les chaînes de caractères

9.1. Propriétés et opérations

Le type `str`, abréviation de *string* (en français *chaîne de caractères*), est un type permettant de stocker des données textuelles. Pour créer une expression de type `str`, on place simplement le texte que l'on veut représenter :

— soit entre guillemets simples, par exemple `'Bonjour à tous.'`,

— soit entre guillemets doubles, par exemple `"Bonjour à tous."`.

Si on utilise les guillemets simples comme délimiteurs, alors on pourra utiliser le caractère « " » sans précaution particulière, en revanche pour utiliser le caractère « ' » il faudra l'*échapper* en le faisant précéder d'un antislash « \ ». Si on utilise les guillemets doubles comme délimiteurs, alors c'est l'inverse. Par exemple :

— la chaîne de caractères « C'est bien ! » peut s'écrire soit `'C\'est bien !'`, soit `"C'est bien !"`,

— la chaîne de caractères « Je lui ai dit : "Zut !". » peut s'écrire soit `'Je lui ai dit : "Zut !".'`, soit `"Je lui ai dit \"Zut !\"."`.

Exercice 9.1. Saisir la chaîne de caractères : « Il m'a dit "Non" et c'est tout. ».

De manière plus générale, la table 9.1 recense quelques *séquences d'échappement* courantes.

Séquence	Caractère obtenu
`\n`	Retour à la ligne
`\t`	Tabulation horizontale
`\'`	Guillemet simple '
`\"`	Guillemet double "
`\\`	Antislash \

FIGURE 9.1. – Quelques caractères obtenus grâce à une séquence d'échappement

Attention. Une séquence d'échappement complète ne compte que pour un seul caractère.

À tester. Observez le résultat affiché par le code ci-dessous.

```
msg = "X\tY\n10\t8\n5\t11\n3\t5"
print(msg)
```

À votre avis, que va donner l'instruction `len(msg)` ? Vérifiez votre résultat.

Remarque. Il est aussi possible d'utiliser des triples guillemets (simples ou doubles) pour saisir une chaîne de caractères sur plusieurs lignes sans utiliser de séquence d'échappement. Ainsi, l'instruction

```
S = """Bonjour

Au
Revoir !"""
```

est équivalente à `S = "Bonjour\n\nAu\nRevoir"` .

Rappelons maintenant quelques propriétés déjà rencontrées au premier semestre.

— Une chaîne de caractères est **indiçable**. Ceci permet d'accéder à un caractère pour le lire, mais ne permet pas de le modifier.

À tester. Observez le résultat des instructions ci-dessous.

```
S = "Hello world!"
a = S[4]    # Valide
S[4] = "u"  # Invalide
```

Plus généralement, comme pour les listes la syntaxe `S[a:b]` renvoie la sous-chaîne obtenue en ne sélectionnant que les caractères de `S` d'indice a (inclus) jusqu'à b (exclus). On peut omettre a (resp. b) pour partir du début (resp. aller jusqu'à la fin) de la chaîne.

À tester. Observez le résultat des instructions ci-dessous.

```
S = "Hello world!"
print(S[2:7])
print(S[:8])
print(S[3:])
```

— L'opérateur `in` permet de tester la présence d'une chaîne dans une autre : `S1 in S2` renvoie `True` si la chaîne `S1` apparaît dans `S2`, et `False` sinon.

À tester. Observez les variables créées par le programme ci-dessous.

```
b1 = ("CHAT" in "UN CHATEAU")
b2 = ("CHIEN" in "UN CHATEAU")
```

— Une chaîne de caractères est **itérable** : l'opérateur `for` permet de parcourir ses caractères un à un.

À tester. Observez le résultat des instructions ci-dessous.

```
word = "AVION"
for x in word:
    print(x)
```

— L'opérateur `+` sur les chaînes correspond à la *concaténation* (c'est-à-dire qu'on « colle » les chaînes bout à bout). On peut alors utiliser l'opérateur `*` entre une chaîne `S` et un entier `n` pour représenter l'opération `S + S + ··· + S`.

À tester. Observez le résultat des instructions ci-dessous.

```
A = "TIC"
B = "TAC"
print(A + B)
print(B + A)
print(2 * A)
print(B * 3)
```

De la même manière qu'on peut créer une liste de manière itérative en ajoutant des éléments un par un à la liste vide, un algorithme commun pour créer une chaîne de caractères est de partir de la chaîne vide et la compléter au fur et à mesure.

À tester. Observez la chaîne contenue dans la variable `S2` à la fin du programme ci-dessous.

```
S1 = "KMD"
S2 = ""
for c in S1:
    S2 = S2 + c + "O"
```

Exercice 9.2. Écrire une fonction `add_dashes(S)` qui prend en argument une chaîne de caractères `S`, et qui renvoie une nouvelle chaîne obtenue en intercalant le caractère « - » entre les caractères de `S`.

Par exemple, `add_dashes("AVION")` *devra renvoyer la chaîne* `"A-V-I-O-N"`.

Exercice 9.3. Écrire une fonction `T_to_X(S)` qui prend en argument une chaîne de caractères `S`, et qui renvoie une nouvelle chaîne obtenue en remplaçant dans `S` la lettre « `T` » (majuscule ou minuscule) par la lettre « `X` » (avec la même capitalisation).

Par exemple, `T_to_X("Tintin au Tibet")` *devra renvoyer la chaîne* `"Xinxin au Xibex"`.

9.2. Méthodes courantes

La table 9.1 (page 105) recense les principales méthodes à connaître du type `str`. Celles-ci sont ici présentées de manière minimaliste. La plupart admettent en réalité un ou plusieurs arguments facultatifs supplémentaires qu'il n'est pas nécessaire de connaître cette année. Le lecteur curieux pourra trouver ces arguments facultatifs et d'autres méthodes non listées ici dans la documentation en ligne [1].

Attention. Ces méthodes ne sont que des raccourcis évitant aux programmeurs de ré-implémenter systématiquement certains algorithmes *simples*. Il faut certes connaître ces méthodes, mais aussi (et surtout!) être capable de les coder soi-même.

1. https://docs.python.org/3/library/stdtypes.html#string-methods

Exercice 9.4.

1. **Sans utiliser de méthode de la table 9.1**, écrire une fonction qui prend en argument une chaîne de caractères `S` et qui renvoie le premier indice où apparaît la lettre `"a"` dans `S` (ou -1 si `S` ne contient pas la lettre `"a"`).
2. Réaliser la même tâche en utilisant la méthode adaptée dans la table 9.1.
3. Si maintenant on cherche le premier indice où apparaît une voyelle minuscule quelconque dans la chaîne `S`, quel code est-il plus simple de modifier parmi les deux précédents ?

9.3. Formatage de chaînes

Le *formatage de chaîne* est une opération consistant à intégrer la valeur d'une ou plusieurs variables dans une chaîne de caractères, en choisissant la façon dont celles-ci seront représentées (par exemple, pour un flottant : le nombre de chiffres après la virgule, la présence d'un signe « + » devant le nombre s'il est positif, etc.). L'exemple ci-dessous présente deux façons équivalentes de procéder sur un cas simple.

À tester. Observer le type et la valeur des variables `S1` et `S2` à l'issue du programme suivant.

```
1 a = 3
2 b = 5
3 S1 = "La somme de {} et {} vaut {}".format(a, b, a + b)
4 S2 = f"La somme de {a} et {b} vaut {a + b}"
```

— Ligne 3, on utilise la méthode `format` du type `str`.

— Ligne 4, on utilise directement la syntaxe des *f-strings*.

Dans les deux cas, on part d'une chaîne de caractères qui joue le rôle de « modèle », dans laquelle chaque paire d'accolades sera remplacée ensuite par la valeur d'une variable.

Attention. Si l'on veut réellement insérer caractère « { » ou « } » dans le modèle, il faut l'écrire en double afin qu'il ne soit pas interprété comme un endroit où l'on souhaite afficher une valeur.

```
x = 12
print("Voici {{}} et voilà {}".format(x))
print(f"Voici {{x}} et voilà {x}")
```

Remarque. Il existe encore une autre syntaxe de formatage qui prend la forme suivante :

```
S = "La somme de %s et %s vaut %s" % (a, b, a + b)
```

Il s'agit de la première syntaxe qui existait avant l'apparition de `format` (Python 2.6) et des *f-strings* (Python 3.6). Celle-ci est donc moins complète et ne sera peut-être pas maintenue dans des futures versions de Python, c'est pourquoi nous ne la présentons pas dans ce cours et nous déconseillons son utilisation.

9.3.1. La méthode `format`

La méthode `format` s'appelle depuis la chaîne de caractères qui servira de modèle, et prend comme arguments les valeurs que l'on souhaite intégrer dans le modèle. Au sein du modèle, on peut :

— soit laisser les paires d'accolades vides, auquel cas les valeurs des variables seront insérées dans la chaîne dans l'ordre où elle ont été passées en argument,

— soit indiquer entre accolades la position de l'argument dont on prendra la valeur à cet endroit-là.

À tester. Analysez les chaînes de caractères créées par le programme suivant.

```
a = 3
b = 9
c = 5
S1 = "Voici trois nombres : {}, {} et {}".format(a, b, c)
S2 = "Voici trois nombres : {0}, {1} et {2}".format(a, b, c)
S3 = "Voici trois nombres : {1}, {0} et {2}".format(a, b, c)
```

Remarque. Un intérêt de la méthode `format` (par rapport aux *f-strings* expliquées ci-après) est que le motif peut être stocké dans une variable avant d'être utilisé.

```
S = "Deux fois {} font {}"
for i in range(1, 10):
    print(S.format(i, 2 * i))
```

9.3.2. Les f-strings

Les *f-strings* ont été introduites afin de proposer une syntaxe plus succincte pour le formatage de chaînes. Il suffit de faire précéder la chaîne du préfixe `f` (**avant** le délimiteur) et de placer les expressions que l'on veut insérer dans la chaîne directement dans les paires d'accolades correspondantes.

À tester.

```
a = 8
b = 3
print(f"La moyenne de {a} et {b} est {(a + b) / 2}.")
```

9.3.3. Options de formatage

Quelle que soit la technique utilisée (`format` ou f-string), on peut préciser davantage la façon dont on veut afficher la variable au sein de la chaîne de caractères de la façon suivante. Au sein de la paire d'accolades, on fait suivre l'éventuelle expression du caractère « : » puis des options de formatage que l'on souhaite utiliser, décrites ci-après.

Commençons par quelques options de formatage communes à tous les types de données (ici n désigne un entier positif).

Option	Description
:n	La variable est affichée de sorte à occuper (au moins) n caractères (en complétant par des espaces si besoin).
:<n	La variable est affichée de sorte à occuper (au moins) n caractères, en étant alignée à gauche.
:>n	La variable est affichée de sorte à occuper (au moins) n caractères, en étant alignée à droite.
:^n	La variable est affichée de sorte à occuper (au moins) n caractères, en étant centrée.

Dans le premier cas, lorsqu'on ne précise pas, l'alignement utilisé dépend du type de données représenté. Par exemple, par défaut les chaînes de caractères sont alignées à gauche tandis que les nombres (entiers ou flottants) sont alignés à droite.

À tester. Observez l'affichage produit par le code ci-dessous.

```
1 words = ["avion", "bateau", "vélo", "voiture"]
2 for i in range(len(words)):
3     S = "| {} | {:7} |".format(i, words[i])
4     print(S)
```

Observez ensuite les (éventuels) changements obtenus pour les variations suivantes de la ligne 3 :

1. `S = " {} | {:>7} |".format(i, words[i])| ,`
2. `S = " {} | {:<7} |".format(i, words[i])| ,`
3. `S = " {} | {:^7} |".format(i, words[i])| .`

Dans le cas où la valeur à formater est un entier, on peut utiliser les syntaxes plus précises suivantes (notez l'apparition du suffixe d), cumulables entre elles lorsque ça a du sens.

Option	Description
:nd	La variable est affichée de sorte à occuper (au moins) n caractères (en complétant par des espaces si besoin).
:0nd	La variable est affichée de sorte à occuper (au moins) n caractères (en complétant par des zéros si besoin).
:+d	La valeur est précédée d'un signe + lorsqu'elle est positive (et d'un signe – sinon, évidemment).

À tester. Observez le résultat affiché par le code suivant.

```
g = 25
print(f"Gain : {g:4d} euros")
print(f"Gain : {g:04d} euros")
print(f"Gain : {g:+4d} euros")
print(f"Gain : {g:+04d} euros")
```

Enfin, on retrouve pour les flottants des options similaires, ainsi que la possibilité d'ajuster le nombre de décimales affichées. Là encore les options sont cumulables entre elles.

Option	Description
`:`n`f`	La variable est affichée de sorte à occuper (au moins) n caractères (en complétant par des espaces si besoin).
`:0`n`f`	La variable est affichée de sorte à occuper (au moins) n caractères (en complétant par des zéros si besoin).
`:+f`	La valeur est précédée d'un signe + lorsqu'elle est positive (et d'un signe – sinon, évidemment).
`:.`p`f`	La valeur est affichée avec p chiffres après la virgule.
`:+0`n`.`p`f`	Toutes les options précédentes.

À tester. Observez l'affichage produit par le code ci-dessous.

```
from numpy import pi
print(f"pi={pi:10f}")
print(f"pi={pi:10.6f}")
print(f"pi={pi:10.4f}")
print(f"pi={pi:+10.4f}")
print(f"pi={pi:010.4f}")
```

Exercice 9.5. Écrire un programme qui affiche les carrés des nombres de 1 à 12 alignés de manière élégante comme suit :

```
 1 x  1 =   1
 2 x  2 =   4
  ...
11 x 11 = 121
12 x 12 = 144
```

Exercices supplémentaires

Exercice 9.6. Écrire une fonction `remove(L, S)` qui prend en argument `L` une liste contenant des caractères et `S` une chaîne de caractères, et qui renvoie une nouvelle chaîne obtenue en retirant de la chaîne `S` tous les caractères contenus dans `L`.

Par exemple, `remove(["v", "a", "l"], "Vive les vacances")` *devra renvoyer la chaîne* `"Vie es cnces"`.

Exercice 9.7. Écrire des fonctions effectuant les mêmes tâches que les méthodes de la table 9.1, mais sans utiliser lesdites méthodes.

Exercice 9.8. La suite de CONWAY (difficile). La suite de CONWAY, ou suite *audioactive*, est définie comme suit. Son premier terme u_0 vaut 1, et chaque terme u_{n+1} est la suite de chiffre formée lorsqu'on lit à voit haute le terme u_n en groupant ensemble les paquets de chiffres consécutifs identiques. Un exemple étant plus clair :

— $u_0 = 1$ se lit « un 1 », donc on pose $u_1 = 11$,

— $u_1 = 11$ se lit « deux 1 », donc on pose $u_2 = 21$,

— $u_2 = 21$ se lit « un 2, un 1 » donc on pose $u_3 = 1211$,

— $u_3 = 1211$ se lit « un 1, un 2, deux 1 » donc on pose $u_4 = 111221$,

et ainsi de suite.

1. Écrire une fonction `first_group(S)` qui prend en argument une chaîne de caractère `S` représentant un terme u_n de la suite de CONWAY, et qui renvoie un entier k indiquant combien de fois le premier chiffre de u_n est répété au début de u_n.
   Par exemple, `first_groupe("1211")` *doit renvoyer* 1 *et* `first_group("111221")` *doit renvoyer* 3.
2. Écrire une fonction `Conway(S)` qui prend en argument une chaîne de caractère `S` représentant un terme u_n de la suite de CONWAY, et qui renvoie une chaîne de caractère représentant le terme suivant u_{n+1}.
   Par exemple, `Conway("1211")` *doit renvoyer* `"111221"` *et* `Conway("111221")` *doit renvoyer* `"312211"`.
3. Afficher les 20 premiers terme de la suite de CONWAY.
4. Sur un graphique, tracer la longueur de u_n en fonction de n, pour n compris entre 1 et 40.

À retenir

— Savoir saisir une chaîne de caractères en échappant les caractères « ' », « " » ou « \ » si nécessaire.
— Connaître le sens des séquences « \n » et « \t ».
— Savoir itérer sur une chaîne de caractères.
— Connaître les effets des opérateurs `+`, `*` et `in` sur les chaînes de caractères.
— Savoir mettre en place un algorithme pour construire pas à pas une chaîne de caractère donnée.
— Connaître les méthodes de la table 9.1 et savoir les ré-implémenter soi-même.
— Savoir utiliser la méthode `format` et les *f-strings*.
— Connaître les options de formatage élémentaires présentées dans ce chapitre.

Méthode	**Description**	**Exemple**
`S.index(s)`	Renvoie le plus petit indice dans `S` où apparaît la chaîne `s`. Provoque une erreur si `S` ne contient pas `s`.	`"Coucou".index("cou")` ↳ `3`
`S.find(s)`	Renvoie le plus petit indice dans `S` où apparaît la chaîne `s`. Renvoie -1 si `S` ne contient pas `s`.	`"Coucou".find("cou")` ↳ `3`
`S.count(s)`	Renvoie le nombre d'occurrences de la chaîne `s` au sein de la chaîne `S`.	`"Didier Dupond".count("d")` ↳ `2`
`S.replace(old, new)`	Renvoie une nouvelle chaîne obtenue en remplaçant dans `S` toutes les occurrences de la chaîne `old` par la chaîne `new`.	`"Coucou".replace("ou", "hien")` ↳ `"Chienchien"`
`S.startswith(s)`	Renvoie un booléen indiquant si la chaîne `S` commence par la sous-chaîne `s`.	`"avion".startswith("av")` ↳ `True`
`S.endswith(s)`	Renvoie un booléen indiquant si la chaîne `S` se termine par la sous-chaîne `s`.	`"avion".endswith("ion")` ↳ `True`
`S.lower()`	Renvoie une nouvelle chaîne obtenue en passant `S` en lettres minuscules (*lower case*).	`"Coucou Martin !".lower()` ↳ `"coucou martin !"`
`S.upper()`	Renvoie une nouvelle chaîne obtenue en passant `S` en lettres majuscules (*upper case*).	`"Coucou Martin !".upper()` ↳ `"COUCOU MARTIN !"`
`S.split(d)`	Renvoie une liste contenant les chaînes obtenues en découpant `S` à chaque occurrence du *délimiteur* `d`.	`"Un, deux, trois !".split(",")` ↳ `['Un', ' deux', ' trois !']`
`S.join(L)`	Concatène les chaînes contenues dans la liste `L`, en reliant deux morceaux consécutifs par la chaîne `S`.	`", ".join(["A", "B", "C"])` ↳ `"A, B, C"`
`S.lstrip()`	Renvoie la nouvelle chaîne obtenue en supprimant tous les caractères d'espacement (espaces, tabulations, retours à la ligne, …) à gauche de `S`.	`"  Coucou  ".lstrip()` ↳ `"Coucou  "`
`S.rstrip()`	Comme précédemment mais en supprimant les caractères d'espacement à droite de `S`.	`"  Coucou  ".rstrip()` ↳ `"  Coucou"`
`S.strip()`	Comme précédemment mais en supprimant les caractères d'espacement des deux côtés de `S`.	`"  Coucou  ".strip()` ↳ `"Coucou"`

TABLE 9.1. – Méthodes appelables à partir d'une chaîne de caractères `S`.