chapitre 6

Les chaînes de caractères

Table des matières

Une **chaîne de caractères** est une donnée composite [1] de type **string** `str` qui est une suite d'entités (ou items) plus simples, les caractères.

En Python une chaîne de caractères est délimitée par des simples ou doubles quotes (voir des triples quotes). On peut la considérer comme une séquence ordonnée de variables qui contiennent chacune un caractère.

Il y a énormément de similitudes entre la gestion des chaînes de caractères et celle des listes vues dans le précédent chapitre.

I Similitudes et différences avec les listes

On crée une variable de type `str` avec une affectation du type `nom_chaine = "J'adore les BCPST du Parc"`. Dès lors comme pour les listes :

- ⋆ Les éléments d'une chaîne sont indexés à partir de `0`. (les espaces comptent comme des caractères)
- ⋆ On accède au caractère d'indice `i` par la commande `nom_chaine[i]`
- ⋆ On obtient sa longueur avec `len(nom_chaine)`
- ⋆ Les chaînes sont des objets itérables, cela veut dire en particulier qu'on peut écrire quelque chose du type : `for i in nom_chaine:`
- ⋆ On peut concaténer des chaînes, et les répéter.

Exemple 6.1.

```
>>> phrase1="J'adore les BCPST du Parc"
>>> phrase2=", et toi ?"
>>>phrase1+phrase2
"J'adore les BCPST du Parc, et toi ?"
>>>a,b="Hip "," houra !"
>>>print(3*a+b)
Hip Hip Hip houra !
```

- ⋆ On peut tester l'égalité de deux chaînes avec `==`.
- ⋆ On peut tester si un élément ou un mot est présent ou non dans une chaîne avec la commande `in`.

Exemple 6.2.

```
>>> phrase1="J'adore les BCPST du Parc"
>>> "a" in phrase1
True
>>>"du P" in phrase1
True
>>>"p" in phrase1
False
```

Remarque : Une majeure partie du TP sera consacrée à programmer la recherche d'un élément (caractère, mot, phrase ...) dans une chaine de caractère, bien entendu sans utiliser la commande `in`

- ⋆ Enfin comme pour les listes on peut faire du slicing

Exemple 6.3.

```
>>> phrase1="J'adore les BCPST du Parc"
>>> phrase1[2:6]
'ador'
>>> phrase1[2:15:2]
'aoelsBP'
>>> phrase1[:10]
"J'adore la"
>>> phrase1[-6:]
'u Parc'
```

 La plus grande différence est que le type `str` est un type de variable **non mutable**, c'est pourquoi on ne peut ni modifier ni supprimer (avec `del`) un (des) élément(s) d'une chaîne de caractères. Par exemple quelque chose du type `nom_chaine[0]='T'` est impossible.

Exercice 6.1

1. Une donnée (variable ou expression) de type composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples. Les listes sont des données composites par exemple.

1. Que renvoie le script suivant :

```
a = "10"
b = "11"
c = a + b
print (c)
```

☐ "1011"
☐ 21
☐ 1011

2. Parmi les expressions ci-dessous, lesquelles peuvent permettre d'obtenir le dernier caractère d'une chaîne non vide nommée `maChaine` ?

☐ `maChaine[len(maChaine)]`
☐ `maChaine[len(maChaine) + 1]`
☐ `maChaine[len(maChaine - 1)]`
☐ `maChaine[len(maChaine) - 1]`
☐ `maChaine[-1]`

3. Que renvoie le script suivant :

```
resultat=""
for c in "Bonsoir" :
    resultat = resultat + c
print(resultat)
```

☐ `B Bo Bon Bons Bonso Bonsoi Bonsoir` *(sur plusieurs lignes)*
☐ `riosnoB`
☐ `BonsoirBonsoirBonsoirBonsoirBonsoirBonsoirBonsoir`
☐ `Bonsoir`

4. Que renvoie le script suivant :

```
resultat=""
for c in "Bonsoir" :
    resultat = resultat + c
    print(resultat)
```

☐ `B Bo Bon Bons Bonso Bonsoi Bonsoir` *(sur plusieurs lignes)*
☐ `riosnoB`
☐ `BonsoirBonsoirBonsoirBonsoirBonsoirBonsoirBonsoir`
☐ `Bonsoir`

II Quelques méthodes sur les chaînes

Avant de commencer sur l'énumération de quelques méthodes une petite remarque. La séquence `\n` dans une chaîne provoque un saut à la ligne alors que le `\t` provoque une tabulation. L'exemple suivant nous permettra en outre d'apprécier la différence entre l'affichage et l'appel d'une chaîne de caractère.

Exemple 6.4.

```
>>> phrase="J'adore\t les BCPST\t du Parc\n et\ntoi ?"
>>> phrase
"J'adore\t les BCPST\t du Parc\n et\ntoi ?"
>>> print(phrase)
J'adore     les BCPST     du Parc
 et
toi ?
```

Les méthodes qui vont être données dans les deux encadrés suivants, nous seront très utile lorsqu'on abordera le chapitre sur les fichiers.

Exemple 6.5.

```
>>> ph="Bonjour comment vas tu ?"
>>> liste=ph.split()
>>> liste
['Bonjour', 'comment', 'vas', 'tu', '?']
```

`nom_chaine.split(sep=..., maxsplit=...)` retourne une liste de mots, en utilisant un séparateur (par défaut l'espace) avec un nombre de découpe égale à maxsplit.

Par exemple si `mot=abracadabra`, alors `mot.split('a')` renvoie `["", 'br', 'c', 'd', 'br', ""]`. On peut mettre `\n` ou `\t`, comme séparateur.

Il existe une méthode réciproque à `split`, il s'agit de `join(liste)` qui rassemble une liste de chaînes en une seule.

```
>>> liste=["j'ai","longtemps","habité","sous","de","vastes","portiques"]
```

```
>>> chaine=' '.join(liste)
>>> chaine
"j'ai longtemps habité sous de vastes portiques"
>>> liste2=["j'", 'i longtemps h', 'bité sous de v', 'stes portiques']
>>> print('A'.join(liste2))
j'Ai longtemps hAbité sous de vAstes portiques
```

Exemple 6.6.

```
>>> mot="abracadabra"
>>> mot.strip('a')
'bracadabr'
```

```
>>> mot.lstrip('a')
'bracadabra'
>>> mot.rstrip('a')
'abracadabr'
```

`nom_chaine.strip([chars])` retourne une copie de la chaîne et enlève le caractère (s'il est présent) en début et fin de chaîne. Par défaut si rien n'est donné on enlève les caractères d'échappement (espace tabulation `\t`, retour à la ligne `\n`) s'ils existent.
`nom_chaine.lstrip([chars])` agit comme `.strip` mais simplement en début de chaîne.
`nom_chaine.rstrip([chars])` agit comme `.strip` mais simplement en fin de chaîne.

Pour être un peu plus complet, voici encore quelques méthodes qu'on utilise avec les chaines de caractères mais qui nous serons moins utiles que les précédente.

Exemple 6.7.

```
>>> nombre=str(3)
>>> nombre
'3'
>>> int('218')
218
>>> float('852.0')
852.0
>>> list('BCPST 852')
['B','C','P','S','T',' ','8','5','2']
```

`str(...)` convertit un objet en chaîne de caractères.

Il existe d'autres méthodes sur les chaînes. On peut accéder à toutes celles-ci en tapant `dir(str)`. En voici des exemples :

```
>>> ph="Bonjour comment vas tu ?"
>>> ph.upper()
'BONJOUR COMMENT VAS TU ?'
>>> ph.lower()
'bonjour comment vas tu ?'
```

Et pour finir, `index` (`nom_chaine.index('a')` *index de la première occurrence de* `'a'` *dans* `nom_chaine`), `count` (`nom_chaine.count('a')` *compte les occurrences de* `'a'` *dans* `nom_chaine`).

Comme on a pu le constater ces méthodes s'utilisent toutes de la même façon :

`nom_chaine.methode`

Remarque : Les objets de type `str` sont non mutables, l'application d'une méthode à une chaîne retourne forcément un nouvel objet.

Exercice 6.2 *Programmation de la méthode* `count`

Écrire un programme qui prend en entrée une chaîne de caractères et un caractère, et qui affiche en sortie le nombre d'occurrence de ce caractère. *On s'interdira bien évidemment d'utiliser la méthode* `count`

III Formatage d'une chaîne

En pratique, on a souvent besoin de **formater** une chaîne de caractères. C'est en effet utile dans tous les cas où on doit construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses. Regardons l'exemple qui suit :

```
>>> jour='mardi'
>>> numero=14
>>> mois='juillet'
>>> annee=1789
>>> heure='16:00'
>>> print("Cela s'est produit le \n {} {} {} de l'année {},\n à {}".format(jour,
     numero,mois,annee,heure))
Cela s'est produit le
 mardi 14 juillet de l'année 1789,
 à 16:00
```

On peut aussi construire cette chaîne en assemblant des morceaux à l'aide de l'opérateur de concaténation (le symbole `+`), mais il vous faudra alors utiliser aussi la fonction intégrée `str()` pour convertir en chaîne de caractères les valeurs numériques. Ceci reste quand même moins pratique que le `.format` qui permet en outre plusieurs possibilités.

En effet, les balises `{}` peuvent aussi contenir des indications de formatage. Par exemple, on peut limiter la précision du résultat final, forcer l'utilisation de la notation scientifique, fixer le nombre total de caractères, etc. :

```
>>> r=4.7
>>> ch ="L'aire d'un disque de rayon {} est égale à {:8.2f}."
>>> print(ch.format(r, pi * r**2))
L'aire d'un disque de rayon 4.7 est égale à ____69.40.
```

Dans l'exemple, le résultat est formaté de manière à comporter un total de 8 caractères (en ajoutant des espaces devant si nécessaire), dont 2 chiffres après le point décimal.

Formatage des chaînes « à l'ancienne »

Les versions de Python antérieures à la version 3.0 utilisaient une technique de formatage légèrement différente et un peu moins élaborée, qui reste encore utilisable. Elle consiste à formater la chaîne en assemblant deux éléments à l'aide de l'opérateur `%`. À gauche de cet opérateur, la chaîne « patron » contenant des balises commençant toujours par `%`, et à droite (entre parenthèses) le ou les objets que Python devra insérer dans la chaîne, en lieu et place des balises.

Exemples 6.8.

```
>>> a = 'python'
>>> 'ceci est un %s'%a
'ceci est un python'
>>> b = 3
>>> 'Un %s de %s m de long'%(a,b)
'Un python de 3 m de long'
>>> '%d est un entier'%3
'3 est un entier'
>>> '%.2f est un flottant avec une précision de %.2f'%(pi,0.01)
'3.14 est un flottant avec une précision de 0.01'
```

La balise `%s` joue le même rôle que `{}` dans la nouvelle technique. Elle accepte n'importe quel objet (chaîne, entier, float, liste...). Vous utilisez aussi d'autres balises plus élaborées, telles que `%.2f` ou `%+d`, qui correspondent à `{:.2f}` ou `{:+d}` de la nouvelle technique. C'est donc équivalent pour les cas les plus simples, mais les possibilités de la nouvelle formulation sont beaucoup plus étendues.

IV Code ASCII

Il faut aussi savoir que les caractères sont des chaînes particulières de longueur 1 et comme l'ordinateur ne manipule que des nombres, une première abstraction est d'associer à chaque caractère un code numérique. En première approche, pour les caractères non accentués de l'alphabet romain, les entiers et les symboles de ponctuation classiques, on peut considérer qu'il s'agit du code ASCII.

Le code ASCII d'un caractère s'obtient alors avec la primitive `ord()` et réciproquement le caractère de code ASCII donné s'obtient avec la primitive `chr()`.

```
>>> for i in range(97,123):
...     print(chr(i),end='')
...
abcdefghijklmnopqrstuvwxyz
>>> for i in 'abcdefghijklmnopqrstuvwxyz':
...     print(ord(i),end=',')
...
97,98,99,100,101,102,103,104,105,106,107,108,109,
110,111,112,113,114,115,116,117,118,119,120,121,122,
```

V Recherche d'un motif

5.1 Recherche d'un élément

Exercice 6.3 *Programmation de l'opérateur* `in` *pour un seul caractère*

1. Écrire une fonction `appartient` qui détermine si un caractère donné appartient à une chaîne de caractères donnée. (La fonction renverra `True` ou `False`)
   Soyez efficaces si votre chaîne comporte 10^{10} caractères et que la lettre que vous cherchez arrive en première position est-ce bien nécessaire de parcourir toute la chaine pour répondre à la question ?
2. *Et sur une liste ?*
   Tester votre fonction sur une liste de nombres par exemple `appartient(12,[1,54,10,12,4])`
3. Écrire une fonction `positions` renvoyant la liste éventuellement vide des positions d'un caractère dans une chaîne. Par exemple `positions("a","abracadabra")` renvoie `[0, 3, 5, 7, 10]`

5.2 Recherche d'un motif

Le problème est le suivant : on dispose d'une « grande » chaîne de caractères (par exemple un fichier de texte, ou un génome séquencé complet) et d'une « plus petite » (par un exemple un mot ou un gène). On souhaite déterminer si la petite chaîne apparaît à l'intérieur de la grande, et si oui en quelle(s) position(s).

Prenons les exemples suivants. Le « mot » `odil` est un sous-mot, ou motif de `crocodile`. De même la liste `[1,3,5]` peut être vue comme une sous-liste, ou motif de la liste `[12,15,1,3,5,19]` (ici, par sous-liste, on entend « en un seul morceau » : `[1,3,5]` n'est pas vue comme une sous-liste de `[1,2,3,4,5]`).

Le but est donc dans un premier temps d'écrire une fonction `sousmot(motif,texte)` renvoyant `True` ou `False` suivant que le motif apparait ou non dans le texte. Ensuite on créera une autre fonction qui renverra la liste éventuellement vide des positions où apparaissent le motif.

Exercice 6.4

1. Écrire une fonction `positioni(motif,texte,i)` qui teste si le sous-mot `motif` apparait dans `texte` à la position `i`. Pour se faire :
   - ⋆ On vérifiera dans un premier temps que `i+len(motif)` n'est pas trop grand.
   - ⋆ Ensuite il faut que `texte[i]=motif[0], texte[i+1]=motif[1]` ... dès qu'on voit une différence on peut sortir en renvoyant `False` sinon si tout est identique on renvoie `True`.
2. En vous servant de la fonction que vous venez d'écrire, écrire une fonction `sousmot(motif,texte)` prenant en entrée deux chaînes de caractères et retournant `True` ou `False` selon que la première est ou n'est pas un sous-mot de la seconde. Pour vérifier votre fonction, on testera les cas suivants :

`sousmot("cro","crocodile"), sousmot("acro","crocodile"), sousmot("roc","crocodile"), sousmot("rocd","crocodile"), sousmot("ile","crocodile"), sousmot("iles","crocodile")`

3. Tester votre fonction sur des listes.
4. Écrire une fonction `listepositions(motif,texte)` prenant en entrée deux chaînes de caractères et retournant la liste (éventuellement vide) des positions dans `texte` où on trouve le mot `motif`. Par exemple `listepositions("ta", "taratata")` renvoie `[0, 4, 6]`. (*Il suffit de rajouter quelques lignes à votre fonction* `sousmot`)

Exercice 6.5 *Recherche d'une séquence génétique dans un brin d'un fragment d'ADN créé au hasard.*

1. Aller chercher le bout de code suivant dans le dossier de partage et exécuter la ligne de commande qui suit.

```
def acgt_aleatoire(longueur):
    """séquence"""
    bases = 'ACGT'
    chaine=''
    for i in range(longueur):
        chaine += bases[randint(0,3)]
    return chaine
```

```
>>> acgt_aleatoire(10**4)
```

2. Créer une première chaîne aléatoire `seq` de longueur 5 avec uniquement les lettres `ACGT` et une deuxième `fragment` de même type mais de longueur 10^4. Déterminer le nombre de positions de `seq` auxquelles on trouve `seq`.

VI D'autres exercices

Exercice 6.6 *À l'envers*

1. Écrire un programme Python qui demande un mot à l'utilisateur puis qui l'écrit à l'envers.
2. En utilisant la fonction précédente écrire une fonction `palindrome` pour savoir si un mot est ou non un palindrome.

Exercice 6.7 On souhaite écrire un programme qui détermine la lettre de l'alphabet la plus fréquente dans une chaîne de caractères. (*On pourra ne pas compter du tout les caractères accentués*)

Voici un exemple de structure que peu avoir votre programme :

- ⋆ On va d'abord créer une fonction `occurrences` qui renverra un tableau contenant le nombre d'occurrences de toutes lettres. Pour ce faire :
  - ▷ On convertit la chaîne en minuscule.
  - ▷ On crée un tableau constitué de 26 zéros correspondant aux occurrences de chaque lettre.
  - ▷ On parcourt la chaîne et on met à jour le tableau, c'est à dire par exemple quand on rencontre un `t` on rajoute `+1` à l'endroit correspondant à `t` dans le tableau.
    On pourra par exemple écrire quelque chose comme :

```
for lettre in chaine:
    if 97<=ord(lettre)<=122:
        tab[ord(lettre)-97]+=1
```

    Les entiers de l'intervalle $[\![97, 122]\!]$ correspondent au code ascii des lettres de `a` à `z`.
    C'est au niveau du `if` qu'il faudrait travailler si on veut compter par exemple un `é` comme un `e`, ou un `ç` comme un `c`.
- ⋆ On utilise ensuite la fonction `positions_maxi` écrite au cours du TP5 (*disponible dans le dossier de partage du serveur*) qui renvoie le maximum d'un tableau et les indices où le maximum est atteint.
- ⋆ On convertit le résultat obtenu pour afficher les caractères apparaissant le plus fréquemment, ainsi que le nombre de fois qu'ils apparaissent.

Exercice 6.8 En utilisant la fonction créée dans le précédent exercice écrire une fonction `anagramme(mot1,mot2)` qui teste si le `mot2` est un anagramme du `mot1`.