Chapitre 4

Chaines de caractère et autres séquences

4.1 Opérations communes

Les types `int`, `float`, `bool` sont des *types scalaires*.

Les types `list` (listes) et `str` (chaînes de caractère) sont des structures de données de *type séquentielles*.

Tous les objets de type séquentiel ont en commun :

Opération	Résultat
`s[i]`	élément d'indice `i` de `s`
`s[i:j]`	Tranche de `i` (inclus) à `j` (exclus)
`s[i:j:k]`	Tranche de `i` à `j` par pas de `k`
`len(s)`	Longueur de `s`
`max(s), min(s)`	Plus grand et plus petit élément de `s`
`x in s`	`True` si `x` est dans `s`, `False` sinon
`x not in s`	`True` si `x` n'est pas dans `s`, `False` sinon
`s+t`	Concaténation de `s` et `t`
`s*n, n*s`	Concaténation de `n` copies de `s`
`s.index(x)`	Indice de la 1^{ere} occurrence de `x` dans `s`
`s.count(x)`	Nombre d'occurrences de `x` dans `s`

où `s` et `t` sont des objets séquentiels de même type, et `i, j, k, n` sont des entiers.

4.1.1 Exemples : concaténation et duplication

```
In [1]: L = [1,2,3] + [4,5]
In [2]: print(L)
[1,2,3,4,5]
In [3]: L * 2
[1,2,3,4,5,1,2,3,4,5]
```

• Pour une liste `L`, un élément `x` et une liste `L2` :
`L.append(x)` a même effet que `L += [x]`
`L.extend(L2)` a même effet que `L += L2`

• concaténation et duplication marchent aussi avec une chaine de caractère :

```
In [4]: ch = 'To' + 'to' ; print(ch)
Toto
In [5]: print(ch*3)
TotoTotoToto
```

4.1.2 les chaînes de caractères

Les objets de type `str`, chaînes de caractère, sont des structures de données séquentielles :

```
>>> chaine = 'Amanda'
>>> len(chaine)
6
>>> print(chaine[::-1])
adnamA
>>> chaine.count('a')
2
>>> print(chaine*2)
AmandaAmanda
>>> max(chaine)
'n'    # minuscules sont > majuscules puis ordre alphabétique
```

• On accède à leur élément par leur indice entre crochets :

```
>>> ch = 'Amanda'
>>> print(ch[0], ch[-1])
A a
```

• On peut les parcourir à l'aide d'une boucle `for` :

```
>>> for c in ch[:3]:
>>>    print(c)
...
A
m
a
```

• Attention : les chines sont **non-modifiables** : changer un élément produit une erreur `TypeError` :

```
>>> chaine[0] = 'Z'
TypeError: 'str' object does not support item assignment
```

4.1.3 Exemple : les palindromes

Un mot est un palindrome s'il est identique lu de gauche à droite et de droite à gauche (à la casse près) :

Exemples : radar, rotor, ressasser

Exemple : Ecrire une fonction qui teste si une chaine est un palindrome :

```
>>> def palindrome(c):
...    if (c == c[::-1]): return True
...    else: return False
```

• Pour que la fonction ne tienne pas compte de la casse, on peut utiliser l'une des méthodes `lower()` ou `upper()` des chaines de caractère, qui convertit en minuscule/majuscule :

```
def palindrome(c):
    ch = c.upper()
    return (ch == ch[::-1])
```

Il vaut aussi mieux s'abstenir du test `if ... else`.

```
In [1]: palindrome('Ressasser')
Out[1]: True
```

Une phrase est un palindrome si après en avoir supprimé les espaces on obtient un mot palindrome.

Exemple : "Engage le jeu que je le gagne".

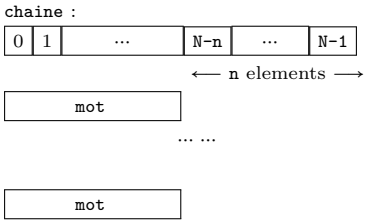
Exercice : écrire une fonction prenant en paramètre une chaine de caractère et qui renvoie `True` ou `False` selon si c'est ou non une phrase palindrome.

```
def Palindrome(ch):
    mot = ""    # Chaine de caractère vide
    for c in ch:    # Parcours des caractères de ch
        if c != " ":    # Si ce n'est pas un espace
            mot += c.upper()    # Ajout après capital.
    return palindrome(mot)    # appel de la fonction préc.
```

4.1.4 Exemple : Recherche d'un mot dans une chaîne

La recherche d'un mot dans une chaîne de caractère est un problème essentiel en informatique qui admet des algorithmes élaborés et efficients. Donner une solution naïve, pas très efficiente n'est pas difficile :

```
def contient(chaine,mot):
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
        if (mot == chaine[i:i+n]):
            return True
    return False
```



Une meilleure solution, (mais toujours naive) est :

```
def cherche(chaine,mot):
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
        k = 0
        while k<n:
            if mot[k] != chaine[i+k]:
                break
            k += 1
        if k == n:
            return True
    return False
```

On compare à chaque position, mais on passe à la position suivante dès que

2 caractères diffèrent.

Il existe des algorithmes beaucoup plus efficients (et plus compliqués).

4.2 Les `tuple`

En français `t-uplet`. Ce sont des objets séquentiels. On les définit par la liste de leurs éléments entre parenthèses `(.)`.

```
>>> seq = (0,1,2,3) ; print(seq)
(0, 1, 2, 3)
>>> print(seq[0])
0
>>> print(seq*2)
(0, 1, 2, 3, 0, 1, 2, 3)
```

Ils diffèrent des listes surtout en ce qu'ils sont non-modifiables.

```
>>> seq[0] = 1
[...]
TypeError: 'tuple' object does not support item assignment
```

Les parenthèses sont optionnelles :

```
>>> a = 2 ; 2*a, 3*a, 4*a    # retourne un t-uplet
(4, 6, 8)
```