Chapitre 1

Révisions sur les séquences

1.1 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.

1.2 Les listes

1.2.1 Exemples : Algorithmes sur les listes

• Recherche d'un élément dans une liste.

```
def recherche(L,e):
    """Recherche si la liste L contient un element de même
    valeur que e"""
    for x in L:
        if x == e:
            return True
    return False
```

Remarque. Les séquences sont *itérables* : l'instruction **`for x in L:`** permet de faire parcourir à `x` toutes les valeurs de `L`.

Exemple.

```
In [1]: recherche([1,5,7,3,1,0,2], 3.0)
Out[1]: True
```

• Recherche du maximum dans une liste numérique.

```
def maximum(L):
    """Renvoie l'element maximal de L"""
    m = L[0]
    for k in range(1,len(L)):
        if L[k] > m:
            m = L[k]
    return m
```

• Recherche du second maximum dans une liste numérique.

```
def maximum(L):
    """Renvoie le second element maximal de L"""
    if L[0] <= L[1]:
        m, M =L[0], L[1]
    else:
        m, M = L[1], L[0]
    for k in range(2,len(L)):
        if L[k] > M:
            m = M
            M = L[k]
        elif L[k] > m:
```

```
            m = L[k]
    return m
```

• Tri par insertion d'une liste numérique.

```
def tri_Insertion(L):
    """Tri L dans l'ordre croissant"""
    for i in range(2,len(L)):
        k = i
        x = L[i]
        while k>0 and L[k-1] > x:
            L[k] = L[k-1]
            k -= 1
        L[k] = x
```

C'est un tri *en place* et *stable*, de *complexité quadratique*.

1.2.2 Concaténation et duplication

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

1.2.3 Méthodes spécifiques aux listes

Les listes disposent des méthodes :

Méthode	Action
`append`	`L.append(e)` ajoute `e` en fin de `L`
`pop`	`L.pop()` supprime et renvoie l'élément en fin de `L`
	`L.pop(i)` supprime et renvoie l'élément de `L` d'indice `i`
`extend`	`L.extend(L1)` ajoute les éléments de la liste `L1` en fin de `L`

Exemple. Liste des 1000 premiers termes de la suite de Fibonacci :

$$u_0 = 0, \quad u_1 = 1, \quad \forall n \in \mathbb{N},\ u_{n+2} = u_n + u_{n+1}$$

```
F = [0, 1]
for _ in range(998):
    F. append(F[-2]+F[-1])
```

• Pour une liste `L`, un élément `x` et une liste `L2` :
`L.append(x)` a même effet que `L += [x]`
`L.append(x)` et `L += [x]` ont une exécution plus rapide que `L = L + [x]`
`L.extend(L1)` a même effet que `L += L1`
`L.extend(L1)` et `L += L1` ont une exécution plus rapide que `L = L + L1`.

1.2.4 D'autres spécificités des listes

• La possibilité de définir une liste *par compréhension* :
Exemple. La liste des carrés des entiers pairs entre 0 et 100 :

```
In [1]: L = [ x**2 for x in range(0,101) if x % 2 == 0 ]
```

qui a le même effet que :

```
L = []
for x in range(0,101):
    if x % 2 == 0:
        L.append(x**2)
```

• Attention au phénomène d'aliasing :

```
In [1]: L1 = [1, 2, 3]
In [2]: L2 = L1
In [3]: L2[0] = 0
In [4]: L1
[0, 2, 3]
```

Pour y pallier effectuer une *copie en profondeur*.

```
In [1]: L1 = [1, 2, 3]
In [2]: L2 = L1[:]
In [3]: L2[0] = 0
In [4]: L1
[1, 2, 3]
```

Mais attention quand même :

```
In [1]: L1 = [[1,2], [3, 4]]
In [2]: L2 = L1[:]
In [3]: L2[0][0] = 0
In [4]: L1
[[0,2], [3,4]]
```

1.3 Les chaînes de caractère

1.3.1 Spécificités

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

```
In [1]: chaine = 'Amanda'
In [2]: len(chaine)
6
In [3]: chaine[::-1])
adnamA
In [4]: chaine.count('a')
Out[4]: 2
In [5]: chaine*2
AmandaAmanda
In [6]: max(chaine)
Out[6] 'n'
```

Les chaînes de caractère sont ordonnables selon l'ordre lexicographiques (grosso-modo). Mais attention : les minuscules sont > aux majuscules (puis ordre alphabétique.

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

```
In[1] : ch = 'Amanda'
In[2] : ch[0], ch[-1]
('A', 'a')
```

• Ils sont itérables :

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

• Attention : les chaînes sont **non-mutables** : une fois crées on ne peut plus les modifier : tenter de changer un élément produit une erreur `TypeError` :

```
In [4]: ch[0] = 'Z'
TypeError: 'str' object does not support item assignment
```

Aussi pour "modifier" des chaines au sein d'un programme, on créera de nouvelles chaines, par concaténation :

Exemple. Un mot est un palindrome s'il est identique lu de gauche à droite et de droite à gauche :

Exemples : radar, rotor, ressasser

Voici une fonction qui teste si une chaine est un palindrome :

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

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".

Écrivons 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       # Ajout du caractère
   return palindrome(mot)  # appel de la fonction prec.
```

1.3.2 Méthode des chaines de caractères

Pour que la fonction précédente 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 tous les caractères alphabétiques :

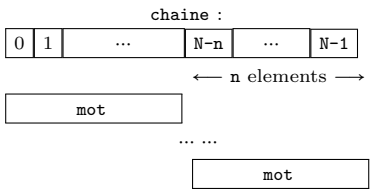
```
In [1] : palindrome("Ressasser".lower())
Out[1]: True
```

1.3.3 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).

1.4 Les `tuple`

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

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

Ils diffèrent des listes surtout en ce qu'ils sont non-mutables :

```
In [4]: seq[0] = 1
[...]
TypeError: 'tuple' object does not support item assignment
```

Les parenthèses sont optionnelles :

```
In [1]:  a = 2 ;  2*a, 3*a, 4*a # retourne un t-uplet
(4, 6, 8)
```

En fait une commande du type (affectation multiple) :

```
a, b = 0, 1
```

consiste en l'affectation d'un `tuple` :

```
(a,b) = (0,1)
```