Introduction à la programmation avec Python

CNAM UTC503 – Paradigmes de programmation

Xavier Crégut <prenom.nom@enseeiht.fr>

Séquence

Définition

Une **séquence** est un type de données qui permet de regrouper dans un même objet un nombre fini d'objets.
Ces objets sont repérés par leur **position** (aussi appelée **indice** ou **index**).

En Python

Plusieurs types de séquences existent :

1. des séquences modifiables : liste (list)
2. des séquences non modifiables : n-uplet (tuple), chaîne (str), intervalle (range)

Exemples

```
s = [4, 'x', 2, False, 2, 7]  # une liste (list)
c = 'bonjour'                 # une chaîne (str)
t = ('a', 1, 2.5)             # un n-uplet (tuple)
r = range(1, 10, 2)           # un intervalle (range)
```

- Les **éléments** d'une séquence peuvent être de types différents (typage dynamique).
- Le même élément peut apparaître plusieurs fois (cas de 2 dans s).
- Une séquence peut être vide : `[]` `''` `""` `()` `range(5, 1)`

Taille et indices

Taille d'une séquence : `len(s)`

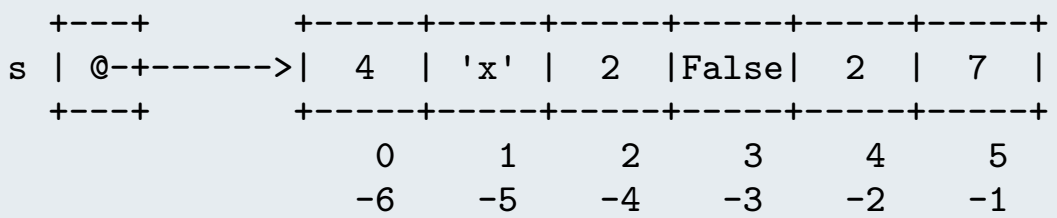
La fonction len() donne la taille de la séquence s fournie en paramètre.
C'est le nombre d'éléments que la séquence contient.

Exemples

```
assert len(s) == 6
assert len(c) == 7
assert len(t) == 3
assert len(r) == 5
assert len([]) == 0
```

Indices

- **Définition :** Un indice permet de désigner un emplacement d'une séquence : `s[i]`
- Un **indice valide** sur une séquence s est un entier i tel que `-len(s) <= i < len(s)`
  - Les indices positifs repèrent les élements de la gauche vers la droite (0 le plus à gauche)
  - Les indices négatifs repèrent les éléments de la droite vers la gauche (-1 le plus à droite)




```
len(s) == 6
s[0] == s[-6] == 4
s[5] == s[-1] == 7
```

- Premier élément (le plus à gauche) : s[0] (ou s[-len(s)], moins pratique)
- Dernier élément (le plus à droite) : s[-1] (ou s[len(s) - 1], moins pratique)
- Si l'indice n'est pas valide, l'exception `IndexError` est levée.

Opérations élémentaires sur les séquences modifiables

- Préambule : Tous les exemples sont exécutés avec `s = [4, 'x', 2, False, 2, 7]`.

Principe : s[i] est équivalent à une variable (nom) qui désigne l'emplacement d'indice i de s.

- `s[indice]` : accès à l'élément à la position `indice` de la séquence s
  - `assert s[0] == 4`
  - `v = s[6]` : lève l'exception `IndexError`
- `s[indice] = expression` : remplacer un élément d'une séquence...
  - associer à `s[indice]` la valeur de `expression`
  - l'élément à la position indice de la séquence est donc la valeur de l'expression
  - `s[3] = 2 ** 3;  assert s[3] == 8 ; assert s == [4, 'x', 2, 8, 2, 7]`
  - `s[0] = s[0] + 1; assert s[0] == 5 ; assert s == [5, 'x', 2, False, 2, 7]`
  - on parle de *left value* (à gauche de l'affectation, ~ variable) et *right value* (à droite, ~ expression)
- `del s[indice]` : supprimer l'élément à un indice donné
  - `del s[3] ; assert s == [4, 'x', 2, 2, 7];  assert s[3] == 2`
  - alternative : `s.pop(indice)` qui retourne l'élément supprimé

- `s.append(expression)` : ajouter la valeur de expression à la fin de la séquence s
  - append est une méthode, d'où la notation pointée.
  - `s.append(5);  assert s == [4, 'x', 2, False, 2, 7, 5];  assert s[-1] == 5`
  - Attention : append retourne None (instruction) : `r = s.append(5);  assert r == None`

Exercices

Exercice : Trouver le type

Indiquer le type des objets suivants :

- `(1.5, 2, "dix")`
- `'liste'`
- `range(3, 10, 2)`
- `[1.5, 2, "dix"]`

trous_sequence.py

```python
# Remplacer les ... pour que le programme s'exécute
# sans aucune erreur signalée par les assert

s = [10, 3, 4, 7, 3, 5]

taille = ...          # la taille de s

assert taille == 6

premier = ...       # le premier élément de s
dernier = ...       # le dernier élément de s

assert premier == 10
assert dernier == 5

indice7 = ...       # entier qui correspond à l'indice de 7 dans s

assert s[indice7] == 7

...                 # remplacer 4 par 421 dans s

assert s == [10, 3, 421, 7, 3, 5]

...                 # supprimer 421 de s

assert s == [10, 3, 7, 3, 5]

...               # ajouter 0 à la fin de s

assert s == [10, 3, 7, 3, 5, 0]
print('ok')
```

Exploiter les éléments d'une séquence

Affectation multiple pour destructurer une séquence

On peut initialiser plusieurs noms avec une séquence.

```
a, b, c = [1, 2, 3]          ; assert a == 1 and b == 2 and c == 3
```

Il doit y avoir autant de noms que d'objets dans la séquence

```
a, b = [1, 2, 3]     # ValueError: too many values to unpack (expected 2)
a, b, c = [1, 2]     # ValueError: not enough values to unpack (expected 3, got 2)
```

On peut avoir un nom préfixé de *. Il correspond à une liste et absorbe les éléments en surplus

```
p, *m, d = [1, 2, 3, 4]    ; assert p == 1 and m == [2, 3] and d == 4
p, *m, d = [1, 2]          ; assert p == 1 and m == [] and d == 2
```

Mais il faut assez d'éléments à droite et au plus une * à gauche.

```
p, *m, d = [1] # ValueError: not enough values to unpack (expected at least 2, got 1)
p, *m, d, *n [1, 2, 3, 4] # SyntaxError: two starred expressions in assignment
```

for et séquence

On peut utiliser un **for** avec une séquence :

```
for x in s:     ## x est associé successivement à chaque objet de la séquence s
    print(x)
```

Exercice : Somme des cubes

Calculer la somme des cubes des éléments d'une séquence d'entiers.

Exemples :

```
séquence              somme
[ 4, -2, 0]     ->    56
[]              ->    0
(1, 2, 3, 4)    ->    100
```

Exercice : Destructuration d'une séquence

trous_sequence_destructuration.py

```
1  # Remplacer les ... par une seule instruction
2  # sans écrire de constantes littérales (1, 2...)
3  # Le programme doit s'exécute sans aucune erreur
4  s = [1, 2, 3, 4]
5
6  ...
7
8  assert a0 == 3 and b0 == 2 and c0 == 1 and d0 == 4
9
10 ...
11
12 assert a1 == 1 and b1 == [2, 3, 4]
13
14 ...
15
16 assert a2 == 4 and b2 == 1 and c2 == [2, 3]
17
18 ...
19
20 assert a3 == 4 and b3 == [1, 2] and c3 == 3
21
22 print('ok')
```

Exercice (6 minutes)

D'après les deux exemples suivants (4 programmes), quelle forme du `for` préférer :

1. `for` sur les éléments (programmes de gauche) ou
2. `for` sur les indices (programmes de droite)

Exemple : Calculer la somme des élements d'une liste d'entiers

Exemple : si liste == [4, 6, 1, 9, 3] alors somme == 33

```
sequence = ... # à définir
somme = 0
for element in sequence:
    somme += element
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    somme += sequence[indice]
```

Exemple : Calculer la somme des élements d'indice pair d'une liste d'entiers

Exemple : si liste == [4, 6, 1, 9, 3] alors somme == 8

```
sequence = ... # à définir
somme = 0
indice = 0
for element in sequence:
    if indice % 2 == 0:
        somme += element
    indice += 1
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    if indice % 2 == 0:
        somme += sequence[indice]
```

enumerate

Quelle forme du for préférer ?

La réponse est le **for** sur les éléments car :

- la ligne du **for** est plus simple (pas de range)
- l'accès à l'élément est direct donc plus lisible (pas d'indice) : `element` vs `sequence[i]`
- elle est légèrement plus efficace : `sequence[i]` oblige à un calcul pour trouver l'élément

Elle comporte un défaut quand on doit manipuler l'indice : il faut le gérer explicitement !

enumerate

enumerate(s, start) où s est une séquence et start un entier (0 si non précisé)
crée une séquence de couples de la forme (start, s[0]), (start+1, s[1)...
Par exemple, `enumerate([4, -2, 0])` crée `[(0, 4), (1, -2), (2, 0)]`.

Grâce à enumerate, la version **for** sur élément redevient plus lisible :

```
sequence = ... # à définir
somme = 0
for indice, element in enumerate(sequence):
    if indice % 2 == 0:
        somme += element
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    if indice % 2 == 0:
        somme += sequence[indice]
```

1. Comment comprendre `for indice, element in enumerate(sequence):` ?
2. Y a-t-il des cas où il faut utiliser **for** avec indices ?

Comment comprendre `for indice, element in enumerate(sequence):` ?

- Il s'agit d'une destructuration d'une séquence
- En effet, enumerate est une séquence de couples
- On fait un `for` dessus
- On fait donc `indice, element = couple`
- Donc indice correspond un premier élément (le numéro de séquence)
- et element au deuxième (un élément de la séquence)

On aurait pu l'écrire ainsi (qui rend la destructuration explicite) :

```
for couple in enumerate(sequence):
    indice, element = couple
```

Le `for` avec indice est utile !

Exercice (3 minutes)

1. Est-ce que les deux programmes suivants répondent à l'énoncé ?
2. Qu'en conclure sur l'utilisation de `for` ?

Exemple : RAZ

Remplacer tous les éléments d'une liste par 0

Exemples :

```
séquence avant        séquence après
[ 4, -2, 0]      ->   [0, 0, 0]
[]               ->   []
['a', True]      ->   [0, 0]
```

```python
sequence = ... # à définir
for element in sequence:
    element = 0
```

```python
sequence = ... # à définir
for indice in range(len(sequence)):
    sequence[indice] = 0
```

Réponse

Constatation :

- Le programme de gauche laisse la liste inchangée.
- Le programme de droite donne le bon résultat.

Pourquoi ?

- `sequence[indice] = 0` change bien l'élément de la liste par 0
- `element = 0` change l'objet associé à `element` mais ne change pas la liste
  - C'est comme quand on fait

```
x = 5     # équivalent de sequence[indice] (un élément de la liste)
y = x     # équivalent de element (un nom qui référence un élément de la liste)
y = 0     # Cette affectation change la valeur de y, pas celle de x
```

Conséquence :

- Toujours utiliser l'indice si on veut changer un élément de la liste
  - Remarque : l'indice peut être obtenu par enumerate.

Conclusion

1. Préférer le `for` sur les éléments
   - il est plus général et marchera avec d'autres types que les séquences
2. Sauf si on doit s'arrêter avant d'avoir parcouru tous les éléments de la séquence
3. Utiliser un indice (via range ou enumerate) pour changer un élément de la séquence

Exercice : Nombre d'occurrences d'un élément dans une séquence.

Calculer le nombre d'occurrences de x dans une séquence s.

Exemples :

- si s = [1, 3, 5, 3] et x = 3 alors le résultat est 2
- si s = [1, 3, 5, 3] et x = 1 alors le résultat est 1
- si s = (3, 2, 5, 5, -5) et x = 0 alors le résultat est 0
- si s = 'dernier' et x = 'e' alors le résultat est 2

Exercice : Est-ce qu'un élément est présent dans une séquence ?

1. Déterminer si un élément x est présent dans une séquence s.

Exemples :

- si s = [1, 3, 5, 0] et x = 5 alors la réponse est oui
- si s = [1, 3, 5, 0] et x = 4 alors la réponse est non
- si s = [1, 3, 5, 0] et x = 0 alors la réponse est oui
- si s = [1, 3, 5, 0] et x = 1 alors la réponse est oui

2. Dans le dernier exemple, combien le programme fait de comparaisons sur les éléments de la séquence avant de donner le résultat.
3. Il faudrait que le programme arrête de parcourir la séquence dès qu'il a trouvé l'élément. Modifier le programme si ce n'est pas le cas.

Exercice : Remplacer toutes les occurrences d'un élément par un autre

1. Remplacer dans une séquence s toutes les occurrences de x par n (comme nouveau).

Exemples :

* `si s = [5, 3, 8, 2, 3, 1], x = 3 et n = 0 alors s devient [5, 0, 8, 2, 0, 1]`
* `si s = [], x = 3 et n = 0 alors s devient []`

2. Ce programme fonctionne-t-il avec un n-uplet (tuple) ou une chaîne (str) ?

Exercice : Indice d'un élément dans une séquence

Déterminer l'indice d'un élément x dans une séquence s. Si l'élément est présent plusieurs fois, on donnera l'indice le plus petit. Si l'élément n'est pas trouvé, on répondra None.

Exemples :

- si s = [5, 3, 8, 2, 3, 1] et x = 8 alors l'indice est 2
- si s = [5, 3, 8, 2, 3, 1] et x = 3 alors l'indice est 1
- si s = [5, 3, 8, 2, 3, 1] et x = 9 alors l'indice est None

Chaînes litérales

- Utiliser apostrophe (') ou guillemet (") pour délimiter les chaînes

```
1   s1 = "Bonjour"       # avec des guillemets
2   s2 = 'Bonjour'       # avec apostrophe
3   s3 = 'Bon' "jour"    # concaténation implicite
4   s4 = 'Bon' + "jour" # concaténation explicite
5   s5 = 'Bon'  \        # \ : caractère de continuation
6       "jour"
7   s6 = ('Bon'              # caractère inutile si (, [ ou { ouvert
8       "jour")
9   s7 = """Une chaîne
10      sur plusieurs
11      lignes"""
12  s8 = '''Avec une ' (apostrophe)
13      dedans'''
14  s9 = "des caractères spéciaux : \t, \n, \", \'..."
15  sA = 'valeur : ' + str(12)  # conversion explicite en str()
```

- Variante avec **triple délimiteur** : permet de continuer la chaîne sur plusieurs lignes.
- Les chaînes avec triple délimiteurs sont utilisées pour la **documentation**.
- Il n'y a **pas de type caractère** en Python (idem chaîne de longueur 1)

Opérateurs sur str (comme séquence immuable)

Opération	Résultat	Exemples
x in s	True si x est une sous-chaîne de s	'bon' in 'bonjour'
x not in s	True si x n'est pas une sous-chaîne de s	'x' not in 'bon'
s + t	concaténation de s avec t	
s * n ou n * s	équivalent à ajouter s à elle-même n fois	'x' * 3 donne 'xxx'
s[i]	ième caractère de s, i == 0 pour le premier	'bon'[-1] donne 'n'
len(s)	la longueur de s (nombre de caractères)	len('bon') donne 3
min(s)	plus petit caractère de s	min('onjour') donne 'j'
max(s)	plus grand caractère de s	max('onjour') donne 'u'
s.index(x[, d[, f]])	indice de la première occurrence de x dans s	'bonjour'.index('o') donne 1
	(à partir de l'indice d et avant l'indice f)	'bonjour'.index('o', 2) donne 4
s.count(x)	nombre total d'occurrence de x dans s	'bonjour'.count('o') donne 2

- i est un indice valide sur s ssi -len(s) $\leq$ i $<$ len(s), sinon IndexError !
- `'bonjour'.index('o', 2, 4)` lève l'exception ValueError car non trouvé
- **Remarque :** Toutes ces opérations sont présentes sur toute séquence.
- Les chaînes de caractères sont des **séquences immuables** de caractères.

Méthodes spécifiques de str

```
s = ' ET '.join( ['un' , 'deux', 'trois'] ) # concaténer avec ce séparateur
assert s == 'un ET deux ET trois'

assert 'chat' < 'chien'    # (Ordre lexicographique, celui du dictionnaire)
assert 'chat' < 'chats'

code = ord('0')       # ord: obtenir le code d'un caractère
assert code == 48     # ... mais quel intérêt de connaître sa valeur précise ? Aucun !
c = chr(code)         # chr : obtenir le caractère correspondant à un code
assert c == '0'
                              # Intérêt de ord et chr :
c = '5'                       # - passer d'un chiffre caractère à l'entier corrspondant
chiffre = ord(c) - ord('0')
assert chiffre == 5
chiffre = 9                   # - et inversement
c = chr(ord('0') + chiffre)
assert c == '9'

r = 'bonjour'.replace('o', 'OO')   # remplacer toutes les occurrences de 'o' par 'OO'
assert r == 'bOOnjOOur'

r = '   xx yy   zz  '.strip()  # supprimer les blancs du début et de la fin
assert r == 'xx yy  zz'

r = '   xx yy   zz  '.split()  # découper une chaîne en liste de chaînes
assert r == ['xx', 'yy', 'zz']
```

Mais aussi lower, islower, upper, isupper, isdigit, isalpha, etc. Voir help('str').

Exercices

1. Comment obtenir le dernier caractère d'une chaîne ?
   - Exemple : 'bonjour' -> 'r'
2. Remplacer tous les `'e'` d'une chaîne par `'*'`.
   - Exemple : `'une chaîne'` -> `'un* chaîn*'`
3. Idem avec `'ne'` deviennent `'...'`.
   - Exemple : `'une chaîne'` -> `'u... chaî...'`
4. Étant donné une chaîne et un caractère, trouver la position de la deuxième
   - occurrence de ce caractère dans la chaîne.
   - Exemple : 'bonjour vous' et 'o' -> 4
5. Indiquer combien il y a de mots dans une chaîne de caractères.
   - Exemple : 'bonjour vous' -> 2
   - Exemple : ' il fait très beau ' -> 4
6. Indiquer le nombre d'occurrences d'une lettre dans une chaîne.
   - Exemple : "C'est l'été, n'est-ce pas ?" contient 1 'a', 3 'e', 0 'v', 3 '", 2 'st', etc.

N-uplet (tuple) : séquence immuable

- **Définition :** Un n-uplet (tuple) est une séquence immuable d'objets quelconques

```
t = (1, 'deux', 10.5)   # t est un tuple composé de 3 objets
u = 1, 'deux', 10.5     # les parenthèses peuvent être omises (si pas ambigu)
v = (1, )   # tuple avec un seul élément (virgule obligatoire)

a, b, c = t # destructurer le tuple : a == 1, b == 'deux', c == 10.5
_, x, _ = t # x == 'deux' et _ == 10.5 (_ pour dire que l'on ne s'en servira pas)
t[0]        # 1
t[-1]       # 10.5
```

- Représenter une date avec le numéro du jour, du mois et de l'année :

```
date_v3 = (3, 12, 2008) # ici, on choisit (jour, mois, année). À documenter !
j, m, a = date_v3       # retrouver ses constituants
```

- On peut construire un tuple à partir d'une séquence

```
tuple('abc')    # ('a', 'b', 'c')
tuple([1, 'X']) # (1, 'X')
```

- Le tuple n'est pas modifiable... mais ses objets peuvent l'être

```
t = (1, [1])   # t référencera toujours cet entier et cette liste
t[1][0] = 2    # L'objet liste est modifiable !
assert t == (1, [2])   # C'est le même objet liste... qui a changé
t[1] = []      # TypeError: 'tuple' object does not support item assignment
```

Liste (list) : séquence modifiable

Création d'une liste

```
l1 = [ 1, 2, 3]       # liste [1, 2, 3]
l2 = []               # une liste vide
```

Opérations sur une liste

```
l1[0]           # 1 : premier élément de la liste
len(l1)         # 3
p, *m, d = l1   # p == 1 and m == [2] and d == 3

l1 += [4, 5]    # l1 == [1, 2, 3, 4, 5]     # concaténer (ou l1.extend([4, 5])), même liste
l1 = l1 + [6]   # l1 == [1, 2, 3, 4, 5, 6]  # mais création d'une nouvelle liste !
del l1[3]       # l1 == [1, 2, 3, 5, 6]
l1[1] = 'x'     # l1 == [1, 'x', 3, 5, 6]         # hétérogène !
l1.append('x')  # l1 == [1, 'x', 3, 5, 6, 'x']    # ajouter un élément à la fin
l1.count('x')   # 2                               # nombre d'occurrences d'un élément
l1.insert(1, 9) # l1 == [1, 9, 'x', 3, 5, 6, 'x'] # insérer l'élément à l'indice
l1.remove('x')  # l1 == [1, 9, 3, 5, 6, 'x'] # supprimer la première occurrence
x = l1.pop(1)   # x == 9 and l1 == [1, 3, 5, 6, 'x']  # supprime l'élément à indice
```

Séquence modifiable

Une liste est une séquence modifiable. Elle a donc toutes les opérations d'une séquence immuable + des opérations de modification.

Questions

Étant donnée une séquence s, par exemple s = [10, 5, 7, 15, 7, 1]

1. Comment obtenir le dernier élément ?
   - Le premier élément de s est 10.
2. Comment obtenir le dernier élément ?
   - Le dernier élément de s est 1.
3. Quel est l'indice de 15 dans s ?
4. Comment obtenir le nombre d'occurrences (la fréquence) d'un élément x ?
   - La fréquence de 7 dans s est 2.
5. Comment obtenir l'indice de la première occurrence de x dans s ?
   - L'indice de la première occurrence de 7 dans s est 2.
6. Comment obtenir l'indice de la deuxième occurrence de x dans s ?
   - L'indice de la deuxième occurrence de 7 dans s est 4.
7. Comment savoir si x est dans s ?
   - 7 est un élément de s. 11 n'est pas un élément de s. . . .

Exercices

Indiquer, après l'exécution de chaque ligne, la valeur de la liste s.

```
1  s = []
2  s.append(2)
3  s.insert(0, 4)
4  s.insert(2, 1)
5  s[1] = 'deux'
6  s[2] /= s[2]
7  s.count(1)
8  s[0], s[1] = s[1], s[0]
9
10 p, _, d = s              # p ? _ ? d ?
11 premier, *suite = s      # premier ? suite ?
12
13 b = [False, True]
14 s += b
15 s2 = [2, 3, 5]
16 i, s2[i], x = s2         # s2 ? i ? x ?
17 s.append(s2)
18 s2.append(s)             # s2 ?  print(s2) ?
19
20 s = list('Fin.')         # s ?  s2 ?
```

Intervalle (range) : séquence immuable d'entiers

- range permet de définir des séquences immuables d'entiers.
- Il est généralement utilisé pour les répétitions (`for`).
- Appels possibles :

```
range(start, stop[, step]) -> range object      # step == 1 si non fourni
range(stop) -> range object                     # start == 0 and step == 1
```

- construit la séquence d'entiers de start inclus à stop exclu de step en step
- Quelques exemples :

```
r1 = range(4)
r1                  # range(0, 4)
list(r1)            # [0, 1, 2, 3]
tuple(r1)           # (0, 1, 2, 3)
r1[-1]              # 3

list(range(4, 8))           # [4, 5, 6, 7]
list(range(8, 4))           # []
list(range(2, 10, 3))       # [2, 5, 8]
list(range(5, 2, -1))       # [5, 4, 3]
list(range(2, 3, -1))       # []
```

Les slices (tranches)

Motivation : Référencer une partie des objets contenus dans une séquence.

Forme générale : `sequence[debut:fin:pas]`

- les élements de `sequence` de l'indice `debut` inclu, à l'indice `fin` exclu avec un pas
- les indices peuvent être positifs (0 pour le premier élément) ou négatifs (-1 pour le dernier)
- possibilité d'utiliser les opérateurs classiques : **del**, =, etc.

Exemples :

```
s = list(range(10))                 # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
s[2:4]                                      # [2, 3]
s[2:]                                       # [2, 3, 4, 5, 6, 7, 8, 9]
s[:-4]                                      # [0, 1, 2, 3, 4, 5]
s[2::4]                                     # [2, 6]
s[::-1]                                     # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
s[8:2:-2]                                   # [8, 6, 4]               (parcours de la fin vers le

del s[2::2]                                 # [0, 1, 3, 5, 7, 9]    (supprimer les éléments de l
s[-4:] = s[:4]                      # [0, 1, 0, 1, 3, 5]
s[:-1] = s[1:]                      # [1, 0, 1, 3, 5, 5]
s[1:] = s[:-1]                      # [1, 1, 0, 1, 3, 5]
s[1:3] = [9, 8, 7, 6]     # [1, 9, 8, 7, 6, 1, 3, 5]
s[::2] = list(range(9)) # ValueError: attempt to assign sequence of size 9 to
                        # extended slice of size 4
range(0, 4)[::-1]                   # range(3, -1, -1)
```

Pour en savoir plus...

Exercices

1. Étant donnée une liste L, comment obtenir la liste de tous les éléments sauf le premier et le dernier ?
   - Exemple : L = [2, 3, 4, 5] donne [3, 4]
2. Étant donnée une liste L, comment obtenir deux listes, la première qui contient les éléments d'indice pair (L1) et la seconde les éléments d'indice impair (L2) ?
   - Exemple : L = [-5, 2, 1, 18, 0] donne L1 = [-5, 1, 0] et L2 = [2, 18]

Compréhension

Exercice : Obtenir la liste des carrés des entiers d'une liste de nombres

Solution : On sait faire ;-)

```
nombres = [4, 2, 1, 5, 6]  # la liste de nombres
carres = []
for n in nombres:
    carres.append(n ** 2)
assert carres == [16, 4, 1, 25, 36]
```

Et en math : Comment notons-nous ceci (en utilisant des ensembles) ?

- $N = \{4, 2, 1, 5, 6\}$: c'est une définition **en extension**
- $C = \{x^2 | x \in N\}$: c'est une définition **en compréhension**

Compréhension en Python :

```
carres = [ x ** 2 for x in nombres ]
assert carres == [16, 4, 1, 25, 36]
```

Intérêts de la compréhension [1] **:**

- formulation plus concise
- plus facile à utiliser car c'est une expression (voir invariant)
- MAIS attention à ne pas avoir des expressions trop compliquées

1. Le mécanisme sous-jacent, les **générateurs** a d'autres intérêts.

Compréhension (suite)

Compréhension avec filtrage : Ajout d'une condition sur les éléments à conserver.

Exemple : Les cubes des entiers pairs d'une séquence

```
nombres = [4, 2, 1, 5, 6]
cubes = tuple(x ** 3 for x in nombres if x % 2 == 0)
assert cubes == (64, 8, 216)
```

est équivalent à (les deux formulations sont très proches) :

```
cubes = []
for x in nombres:
    if x % 2 == 0:
        cubes.append(x ** 3)
cubes = tuple(cubes)
```

Remarque : Il faut écrire explicitement `tuple` sinon on obtient un générateur et non un n-uplet.

Matrices

Principe : On peut utiliser une liste de listes.

```
# Une matrice comme une liste de listes (en extension)
# Chaque liste représente une ligne de la matrice
# (ou une colonne, c'est une convention)
matrice = [[0,  1,  2,  3],
           [10, 11, 12, 13],
           [20, 21, 22, 23]]

# On remarque que dans matrice[i][j] on a i * 10 + j
assert matrice[2][1] == 21

# La même matrice en compréhension
matrice2 = [ [i * 10 + j for j in range(4) ] for i in range(3)]
assert matrice2 == matrice

# La même sans compréhension
matrice3 = []
for i in range(3):
    ligne = []
    for j in range(4):
        ligne.append(i * 10 + j)
    matrice3.append(ligne)
assert matrice3 == matrice
```

Remarque : Il existe des modules spécialisés comme Numpy.

Sequence et for

- destructuration et `for` : plusieurs noms si séquence de séquences

```
for nom, age in [('Paul', 18), ('Sarah', 19), ('Nicolas', 21), ('Emma', 16)]:
    print(nom, 'a', age, 'ans', end='.  ')
```

Affiche : 'Paul a 18 ans. Sarah a 19 ans. Nicolas a 21 ans. Emma a 16 ans.'

Exercice : Consigne
Traiter les deux exercices suivants en utilisant 1) un `while` et 2) un `for` puis comparer les deux solutions.

Exercice : équivalent de `count`
On veut calculer le nombre d'occurrences d'un élément x dans une séquence s.

Exercice : équivalent de `index`
On veut trouver l'indice de la première occurrence d'un élément x dans une séquence s.
L'indice sera None[2] si x n'est pas dans s.
On accèdera au plus une fois aux éléments de s.

2. l'opération index de Python lève une exception au lieu de retourner `None` si l'élément n'est pas dans la liste.

Solution pour équivalent de count

- Avec un `while`

```
indice: int = 0     # indice sur s.
frequence: int = 0      # nombre de x dans s[0:indice]
while indice < len(s):  # encore des éléments à considérer
    # Variant : len(s) - indice
    # Invariant : frequence = nombre d'apparition de x dans s[0:indice]
    if s[indice] == x:
        frequence += 1
    indice += 1
```

- Avec un `for`

```
frequence: int = 0      # nombre de x trouvés dans s
for elt in s:
    if elt == x:
        frequence += 1
```

- **Constat :** La version avec `for` est plus naturelle
- **Compréhension :**

```
frequence = sum(1 for elt in s if elt == x)
```

Solution pour équivalent de index

- Avec un `while`

```
indice: int = 0    # indice sur s.
while indice < len(s) and s[indice] != x:  # encore des éléments ET pas le bon
    # Variant : len(s) - indice
    # Invariant : x not in s[0:indice]
    indice += 1
if indice >= len(s): # x n'a pas été trouvé
    indice = None
```

- Avec un `for`

```
indice: int = 0    # indice sur s.
for elt in s:
    # Invariant : x not in s[0:indice]
    if elt == x:
        break
    indice += 1
else:
    indice = None
```

- **Constats :**
  - on ne peut pas préciser de condition de sortie dans un `for` d'où l'utilisation de `break`
  - le bloc `else` ne sera exécuté que si aucun `break` n'a été exécuté (et donc x non trouvé)
  - la gestion de l'indice alourdit le `for` !

enumerate et zip

- **enumerate** est une fonction qui « retourne » [3] autant de couples qu'il y a d'éléments dans une séquence. Chaque couple est composé d'un numéro d'ordre et d'un élément de la séquence.

```
assert list(enumerate('ABC')) == [(0, 'A'), (1, 'B'), (2, 'C')]
```

- Nouvelle solution pour index avec un `for` et enumerate : simplifie le code !

```
for indice, elt in enumerate(s):
    if elt == x:
        break
else:
    indice = None
```

- **zip** : « retourne » [4] les couples formés en prenant successivement un élément de chacune des séquences fournies en paramètre.
- la plus petite séquence détermine le nombre de couples

```
assert list(zip(range(1, 4), 'ABCD',  [3, 2, 5, 7])) \
        == [(1, 'A', 3), (2, 'B', 2), (3, 'C', 5)]
```

3. C'est en fait un générateur.
4. C'est en fait un générateur.