Séquences
Corrigé

Objectifs

— Comprendre et savoir utiliser les séquences
— Écrire et tester des sous-programmes (toujours !)
— Raffiner (toujours !)

1 Comprendre les séquences

Exercice 1 : Opérations sur les séquences

Compléter le programme `comprendre-sequence-a-trous.py` ci-après pour écrire les instructions Python qui réalisent les objectifs exprimés en commentaires. On utilisera **`assert`** pour indiquer le résultat des expressions écrites ou l'effet des instructions écrites.

```
'''Opérations sur les séquences (avec une liste).
Utiliser assert pour indiquer les résultats attendus.'''

# Initialiser un nom s avec une liste contenant dans l'ordre 9, 1, 5, 2, 1 et 3
...

# Obtenir la taille de la liste s (le nombre d'éléments qu'elle contient)
assert 6 == ...

# Savoir si l'élément 2 est présent dans s.  Idem pour l'élément 4
...

# Obtenir le premier élément de s
assert 9 == ...

# Obtenir le dernier élément de s
```

```python
assert 3 == ...

# Obtenir la fréquence (nombre d'occurences) de 1 dans s
assert 2 == ...

# Supprimer l'élément à l'indice 2 de s
...
assert s == [9, 1, 2, 1, 3]

# Supprimer l'élément 2 de s
...
assert s == [9, 1, 1, 3]

# Ajouter 7 à la fin de s
...
assert s == [9, 1, 1, 3, 7]

# Ajouter 6 en position 2 dans s
...
assert s == [9, 1, 6, 1, 3, 7]

# Ajouter 0 avant la première occurrence de x dans s (x pourrait être 1, 3...)
for x in (1, 3):
    ...
assert s == [9, 0, 1, 6, 1, 0, 3, 7]

# Obtenir la somme des carrés des nombres de s ?
...
assert somme_carres == 177
```

Solution :

```python
'''Opérations sur les séquences (avec une liste).
Utiliser assert pour indiquer les résultats attendus.'''

# Initialiser un nom s avec une liste contenant dans l'ordre 9, 1, 5, 2, 1 et 3
s = [9, 1, 5, 2, 1, 3]

# Obtenir la taille de la liste s (le nombre d'éléments qu'elle contient)
assert 6 == len(s)

# Savoir si l'élément 2 est présent dans s.  Idem pour l'élément 4
assert 2 in s
    # Ceci signifie « est que 2 in s est vrai ? ».
    # Écrire « assert 2 in s == True » serait donc un pléonasme
assert 4 not in s
    # Écrire « assert 4 in s == False » serait moins naturel (et donc déconseillé)
    # En fançais, on dit « 4 n'est pas dans dans s », non ?
    # On aurait aussez pu écrire : assert not 4 in s (un peu moins naturel)

# Obtenir le premier élément de s
```

```python
assert 9 == s[0]

# Obtenir le dernier élément de s
assert 3 == s[-1]          # C'est la bonne solution !
assert 3 == s[len(s) - 1]  # Marche aussi mais beaucoup plus lourd !
assert 3 == s[5]           # Ne marche que si s est de longueur 6 => Moins général !

# Obtenir la fréquence (nombre d'occurences) de 1 dans s
assert 2 == s.count(1)

# Supprimer l'élément à l'indice 2 de s
s.pop(2)    # ou `del s[2]`
assert s == [9, 1, 2, 1, 3]

# Supprimer l'élément 2 de s
s.remove(2)
assert s == [9, 1, 1, 3]

# Ajouter 7 à la fin de s
s.append(7)
assert s == [9, 1, 1, 3, 7]

# Ajouter 6 en position 2 dans s
s.insert(2, 6)
assert s == [9, 1, 6, 1, 3, 7]

# Ajouter 0 avant la première occurrence de x dans s (x pourrait être 1, 3...)
for x in (1, 3):
    s.insert(s.index(x), 0)
assert s == [9, 0, 1, 6, 1, 0, 3, 7]

# Obtenir la somme des carrés des nombres de s ?
# - Solution en utilisant une boucle
somme_carres = 0
for nombre in s:
    somme_carres = somme_carres + nombre ** 2
assert somme_carres == 177

# - Solution en utilisant une liste en compréhension et la fonction sum
somme_carres = sum(nombre ** 2 for nombre in s)
assert somme_carres == 177
```

Exercice 2 : Le type list

Indiquer l'effet des instructions du programme suivant :

```python
s = []
s.append(2)             # s ?
s.insert(0, 4)          # s ?
s.insert(20, 7)         # s ?
s[1] = 'd'              # s ?
```

```
s[2] /= s[2]            # s ?
s.count(1)              # ?
s[0], s[1] = s[1], s[0] # s ?

p, _, d = s             # p ? _ ? d ?
premier, *suite = s     # premier ? suite ?

b = [False, True]
s.extend(b)             # s ?

x = s.pop(1)            # x ? s ?

s2 = [2, 3, 5]
i, s2[i], x = s2        # i ? s2 ? x ?

s.append(s2)            # s ?
s2.append(s)            # s2 ? s ?

t = tuple(b)            # t ?
s = list('Fin.')        # s ? s2 ?
```

Solution :

```
s = []
s.append(2)             # s ?
assert s == [2]         # 2 ajouté à la fin de la liste

s.insert(0, 4)          # s ?
assert s == [4, 2]      # 4 ajouté avant l'élément en position 0 (au début donc)

s.insert(20, 7)         # s ?
assert s == [4, 2, 7]   # 1 ajouté avant la position 20 (ici à la fin)

s[1] = 'd'              # s ?
assert s == [4, 'd', 7] # remplace l'élément à la position 1 par 'd'

s[2] /= s[2]            # s ?
assert s == [4, 'd', 1.0]  # s[2] = s[2] / s[2]

s.count(1)              # ?
assert 1 == s.count(1)  # fréquence de 1 (1.0 est bien égal à 1)

s[0], s[1] = s[1], s[0] # s ?
assert s == ['d', 4, 1.0]  # permutation des deux premiers éléments


p, _, d = s             # p ? _ ? d ?
assert p == 'd'
assert _ == 4
assert d == 1.0

```

```
premier, *suite = s     # premier ? suite ?
assert premier == 'd'
assert suite == [4, 1.0]


b = [False, True]
s.extend(b)             # s ?
assert s == ['d', 4, 1.0, False, True]  # concaténation


x = s.pop(1)            # x ? s ?
assert x == 4                       # retourne élément à la position 1
assert s == ['d', 1.0, False, True]  # et le supprime de la liste


s2 = [2, 3, 5]
i, s2[i], x = s2        # i ? s2 ? x ?
assert i == 2
assert s2 == [2, 3, 3]
assert x == 5
        # d'abord la partie droite est évaluée : [2, 3, 5]
        # elle sert à initialiser les variables à gauche de « = »
        # les variables (à gauche) sont initialisées de gauche à droite

        # Attention : ce type d'expression est fortement déconseillé !


s.append(s2)            # s ?
assert s == ['d', 1.0, False, True, [2, 3, 3]]  # une liste dans une liste

s2.append(s)            # s2 ? s ?
assert str(s2) == "[2, 3, 3, ['d', 1.0, False, True, [...]]]"
    # Chaque liste contient l'autre. On a donc créé un cycle.
    # Lors de l'affichage, Python utilisera « [...] » pour dire que la liste
    # a déjà été affichée (et éviter un affichage sans fin).
assert str(s) == "['d', 1.0, False, True, [2, 3, 3, [...]]]"
assert s2 == [2, 3, 3, ['d', 1.0, False, True, s2]]
assert s2 == [2, 3, 3, ['d', 1.0, False, True, [2, 3, 3, s]]]


t = tuple(b)            # t ?
assert t == (False, True)          # Création d'un tuple à partir d'une séquence

s = list('Fin.')        # s ? s2 ?
assert s == ['F', 'i', 'n', '.']   # Création d'une liste à partir d'une chaîne
assert s2 == [2, 3, 3, ['d', 1.0, False, True, s2]] # s2 non modifiée
        # et la liste associée avant à s continue d'exister !

```

Exercice 3 Répondre de manière concise et précise aux questions suivantes.

3.1. Expliquer ce qu'est un objet immuable. On donnera des exemples.
Solution : Un objet immuable est un objet qui ne peut pas être modifié. Les objets de type *str*, *tuple* et *range* sont immuables.
3.2. Quelles sont les séquences proposées par Python ? Donner un exemple de chaque.
Solution :

1. list (liste) : séquence modifiable, une liste d'objets quelconques.
   Exemple : `[1, 'a', False, [4, 2, 1]]`
2. tuple (n-uplet) : séquence immuable, une liste d'objets quelconques.
   Exemple : `(1, 'a', False, [4, 2, 1])`
3. str (chaîne de caractères) : séquence immuable de caractère.
   Exemple : `'une chaîne'`
4. range (intervalle) : séquence immuable d'entiers.
   Exemple : `range(1, 10, 2)`

3.3. Indiquer ce que signifient `is` et `==` en Python. Donner un exemple.
Solution : `is` correspond à l'égalité physique (est-ce la valeur à gauche et la valeur à droite correspondent au même objet) et `==` correspond à l'égalité physique (les valeurs à gauche et droite ne sont pas forcément le même objet mais les deux objets ont les mêmes caractéristiques, par exemple 2 liste avec les mêmes éléments).
Exemple :

```
s1 = [1, 3, 5]
s2 = [1, 3, 5]
s3 = s2

assert s1 is not s2  # pas les mêmes objets
assert s1 == s2      # mais les deux listes ont les mêmes valeurs
```

Ne pas hésiter à utiliser `http://www.pythontutor.com/visualize.html#mode=edit` pour exécuter ce programme. On pourra essayer de modifier s1[0], puis s2[0] puis s3[0] et regarder ce que sont les listes s1, s2 et s3 après chaque modification.

Exercice 4 : Éléments d'un n-uplet

Considérons le n-uplet (tuple) `t = (5, 'x', [])`.
4.1. Indiquer comment accéder aux composants du n-uplet : `5`, `'x'` et `[]`.
Solution : On peut y accéder via des indices : `t[0], t[1], t[2]`.
On peut aussi utiliser l'affectation :

```
chiffre, lettre, liste = t
```

Cette deuxième solution est souvent plus lisible.
4.2. Indiquer si les instructions suivantes sont autorisées et, dans l'affirmative, ce que vaut `t`.

```
t[0] = 6
del t[1]
t[2].append(-1)
```

Solution : Les deux premières sont interdites car un n-uplet est un objet inaltérable en Python.

La dernière est acceptée. Le n-uplet devient (5, 'x', [-1]). Le n-uplet n'est pas modifiable mais les objets qu'il contient peuvent l'être. Ici t référencera toujours la même liste mais cet objet liste peut être modifié.

2 Algorithme de tri

Exercice 5 : Tri par sélection

Soit A un vecteur (une liste en Python) de N entiers relatifs quelconques, l'objectif est de trier le vecteur A en utilisant le tri par sélection. Le vecteur A est trié si $A[i] \leq A[i+1]$.

Le tri par sélection est un tri en $(N-1)$ étapes. L'étape i consiste à ranger à sa place le i^{e} plus petit élément du vecteur.

Exemple : Voici les différentes valeurs du vecteur 8 2 9 5 1 7 après chaque étape (la partie encadrée correspond à la partie du vecteur déjà traitée et donc triée) :

vecteur initial	:	8 2 9 5 $\underline{1}$ 7
après l'étape 1	:	$\boxed{1}$ $\underline{2}$ 9 5 8 7
après l'étape 2	:	$\boxed{1\ 2}$ 9 $\underline{5}$ 8 7
après l'étape 3	:	$\boxed{1\ 2\ 5}$ 9 8 $\underline{7}$
après l'étape 4	:	$\boxed{1\ 2\ 5\ 7}$ $\underline{8}$ 9
après l'étape 5	:	$\boxed{1\ 2\ 5\ 7\ 8\ 9}$

5.1. Écrire un sous-programme qui trie un vecteur en utilisant le tri par sélection. On complètera le fichier `tri_selection.py` et on utilisera le fichier `test_tri_selection.py`. On affichera la liste après chaque étape du tri pour vérifier que c'est bien l'algorithme demandé qui a été implanté. On testera avec :

```
pytest --doctest-modules test_tri_selection.py
```

Solution :

```
1  def trier_selection(liste):
2      '''Trier la liste dans l'ordre croissant.
3
4      :param liste: la liste à trier (in out)
5      :type liste: liste d'éléments comparables
6
7      Algorithme utilisé :
8          le tri par sélection.  Utilisation seulement de l'accès au ième élément
9          (consultation ou modification)
10
11     >>> trier_selection([8, 2, 9, 5, 1, 7])
12     [1, 2, 9, 5, 8, 7]
13     [1, 2, 9, 5, 8, 7]
14     [1, 2, 5, 9, 8, 7]
```

```
    [1, 2, 5, 7, 8, 9]
    [1, 2, 5, 7, 8, 9]
    '''
    for etape in range(len(liste) - 1):
        # Placer dans liste[etape] le plus petit élément de liste[etape:]
        #    Trouver l'indice imin du plus petit élément de liste[etape:]
        imin = etape          # Indice du plus petit élément
        mini = liste[etape]   # Valeur du plus petit élément
        for i in range(etape+1, len(liste)):
            if liste[i] < mini:
                imin = i
                mini = liste[i]

        #    Permuter les éléments aux indices etape et imin
        liste[etape], liste[imin] = mini, liste[etape]

        print(liste)
```

3 Pour aller plus loin...

Exercice 6 : Le jeu du pendu

Écrire un programme qui permet de jouer au pendu, la machine étant le bourreau.

Le jeu du pendu se joue à deux. Un premier joueur, le *bourreau*, choisit un nom commun qu'il garde secret. Il indique seulement à l'autre joueur, le *joueur*, le nombre de lettres du mot. Le joueur propose une lettre. Si cette lettre correspond à une lettre du mot, alors le bourreau indique le nombre d'occurrences de la lettre dans le mot et la position de chacune des occurrences. Si la lettre ne fait pas partie du mot ou si la lettre a déjà été proposée, le bourreau ajoute une pièce à la potence. Si la potence est complète (11 pièces), le joueur perd : il est pendu ! S'il trouve toutes les lettres du mot, il gagne.

Dessiner la potence et le pendu représente 11 tracés qui sont dans l'ordre :

— la *potence* : le socle, le montant vertical, la barre horizontale, le renfort ;
— le *pendu* : le corps, la jambe gauche, la jambe droite, le bras gauche, le bras droit, la tête ;
— la *corde* : lorsque la code est placée, la partie est finie : le joueur est pendu !

Avant chaque demande de proposition au joueur, l'ordinateur affichera les lettres déjà identifiées du mot à trouver. En fait, il affiche le mot à trouver en mettant un tiret « – » à la place des lettres non encore trouvées. C'est ce qu'on appelle le « mot du joueur ».

Par exemple, si le mot à trouver est « élève » et que le joueur propose successivement « A », « E », « T », « R », « C », « L », « V », le programme affichera les lignes suivantes (l'affichage de la potence n'est pas reproduit).

```
Mot :  - - - - -

Une lettre ? A
J'ajoute une pièce à la potence !

Mot :  - - - - -

Une lettre ? E
```

```
La lettre "E" est dans le mot.

Mot :  E - E - E

Une lettre ? T
J'ajoute une pièce à la potence !

Mot :  E - E - E

Une lettre ? R
J'ajoute une pièce à la potence !

Mot :  E - E - E

Une lettre ? C
J'ajoute une pièce à la potence !

Mot :  E - E - E

Une lettre ? L
La lettre "L" est dans le mot.

Mot :  E L E - E

Une lettre ? V
La lettre "V" est dans le mot.

Bravo, vous avez trouvé "ELEVE".
```

Attention : Pour faciliter la lecture du mot, les lettres (ou les tirets) seront affichées en étant séparées par un espace.

Simplifications :

1. On suppose que les mots à trouver seront *toujours* orthographiés sans accents et en majuscules. Ainsi pour le mot « Élève », le mot à trouver est « ELEVE ».
2. On suppose également que les mots à trouver ne comportent que des lettres et aucun autre signe. On ne peut donc pas avoir de mots tels que « aujourd'hui », « chasse-neige »...

Solution : Dans cet exercice, il faut décider des informations que l'on va utiliser. Il faudra forcément conserver les lettres proposées. À partir du mot à trouver et des lettres déjà proposées, on pourra afficher le mot à l'utilisateur en faisant apparaître les '-' aux positions des lettres non encore proposées.

On peut aussi décider d'ajouter une variable qui correspond au mot tel qu'il est connu du joueur. C'est ce que nous faisons dans la solution ci-dessous. Le « mot du joueur » est la liste des lettres du mot avec des tirets pour les lettres non encore découvertes.

Les raffinages n'ont pas été reproduits ici mais ils apparaissent dans le programme au travers des sous-programmes écrits et des commentaires.

```python
'''Jouer au jeu du pendu.'''

from typing import List
from potence import afficher_potence, NB_ETAPES
from mots import un_mot


def jouer_au_pendu() -> None:
    '''
    Jouer au pendu : l'utilisateur doit trouver le mot.
    '''
    # choisir un mot
    mot = un_mot()

    # faire trouver ce mot
    nb_erreurs = 0              # nombre d'erreurs commises par le joueur
```

```
17      ... # TODO à compléter
18      lettres_proposees: List[str] = []   # Les lettres proposées par l'utilisateur
19      mot_utilisateur = ['-'] * len(mot)  # Aucune lettre trouvée
20      while '-' in mot_utilisateur and nb_erreurs < NB_ETAPES:
21          # afficher le mot
22          print()
23          print('Mot :', end='')
24          for lettre in mot_utilisateur:
25              print('', lettre, end='')
26          print()
27
28          # demander une lettre à l'utilisateur
29          message = 'Une lettre ? '
30          reponse = input(message).strip().upper()
31          while len(reponse) != 1 or not ('A' <= reponse[0] <= 'Z'):    # réponse invalide
32              # signaler l'erreur
33              if len(reponse) != 1:
34                  print("J'attends exactement un caractère")
35              else:
36                  print("Il faut donner une lettre de A à Z")
37
38              # redemander une lettre
39              reponse = input(message).strip().upper()
40          lettre = reponse
41
42          # déterminer si le coup est gagnant
43          gagnant = lettre not in lettres_proposees and lettre in mot
44
45          # afficher le verdict et mettre à jour l'état du jeu
46          if gagnant:
47              print(f'La lettre "{lettre}" est dans le mot.')
48              lettres_proposees.append(lettre)
49              # dévoiler les lettres du mot
50              for indice, lettre_mot in enumerate(mot):
51                  if lettre_mot == lettre:
52                      mot_utilisateur[indice] = lettre
53          else:
54              print("J'ajoute une pièce à la potence !")
55              nb_erreurs += 1
56              afficher_potence(nb_erreurs)
57
58      # Afficher le résultat du jeu
59      if nb_erreurs >= NB_ETAPES:
60          print('Perdu !')
61      else:
62          print()
63          print(f'Bravo, vous avez trouvé "{mot}".')
64
65  if __name__ == '__main__':
66      jouer_au_pendu()
```