Séance 3 : Boucles for et chaînes de caractères

L1 – Université Côte d'Azur

Boucle `for` ou boucle `while` ?

De manière générale, il est préférable d'éviter d'utiliser une boucle `while` là où une boucle `for` suffit, à la fois par soucis de lisibilité et de **simplicité** du code (la boucle permet de s'assurer que l'on sortira de la boucle et évite de gérer la variable de boucle). Pour vous aider à prendre de bonnes habitudes, dans ce TD, il est **interdit d'utiliser la boucle `while`**.

Exercice 1 — Jouons avec le `range` (⋆)

1. On considère le code suivant :

```
for i in range(…) :
    print(i)
```

   Que faut-il ajouter à l'intérieur du `range(…)` dans le code ci-dessus de façon à afficher (avec un nombre par ligne et sans les virgules) :

   a) 1, 2, 3, 4, 5, 6, 7

   b) 1, 3, 5, 7, 9, 11, 13

   c) 17, 14, 11, 8, 5, 2, -1

2. Mêmes questions, mais avec le code ci-dessous en modifiant le `print`

```
for i in range(7) :
    print(…)
```

Exercice 2 — Comptine (⋆)

1. Écrivez une fonction `affiche_comptine(n)` qui prend en argument un entier n et qui **affiche** les n premières lignes d'une comptine un peu usante à la longue. On prendra garde à faire l'accord du pluriel.

```
>>> affiche_comptine(4)
"1 kilomètre à pied, c'est long."
"2 kilomètres à pied, c'est très long."
"3 kilomètres à pied, c'est très très long."
"4 kilomètres à pied, c'est très très très long."
```

2. Modifiez votre fonction pour en faire une fonction `comptine(n)` qui **renvoie** une unique chaîne de caractères contenant la comptine, de sorte que l'on puisse répondre à la première question en faisant simplement

```
def affiche_comptine(n) :
    print(comptine(n))
```

Exercice 3 — Un exercice renversant (⋆)

1. Écrivez une fonction `affiche_miroir(s)` qui prend une chaîne de caractères `s` et qui **affiche** la chaîne `s` et son image miroir (`s` à l'envers) ; vous proposerez deux solutions, une avec un pas de boucle de -1, et l'autre avec un pas de boucle de 1.

```
>>> affiche_miroir('abc')
abc cba
```

2. Écrivez une fonction `miroir(s)` qui cette fois-ci **retourne** la chaîne de caractères `s` à l'envers, de sorte que l'on puisse répondre à la question 1 par

```
def affiche_miroir(s):
    print(s,miroir(s))
```

3. Écrivez une fonction `est_un_palindrome(s)` qui retourne `True` si `s` est un palindrome, autrement dit un mot qui est égal à son image miroir. Proposez une solution qui n'utilise pas la fonction `miroir` et qui ne crée aucune nouvelle chaîne de caractères. Écrire quatre tests avec `assert`.

Exercice 4 — La disparition (⋆⋆)

1. Écrivez une fonction `nombre_apparitions(c,s)` qui prend en paramètre un caractère `c` et une chaîne de caractères `s` et qui renvoie le nombre de fois où `c` apparait dans `s`. Par exemple,

```
>>> nombre_apparitions('e','les revenentes')
5
```

```
def nombre_apparitions(c,s) :
    res = 0
    for i in range(len(s)) :
        if s[i] == c :
            res = res + 1
    return res
```

On peut itérer directement sur les caractères. Cette variante pythonnesque n'est pas possible dans tous les langages.

```
def nombre_apparitions_bis(c,s) :
    res = 0
    for c2 in s :
        if c == c2 :
            res = res + 1
    return res
```

2. En déduire une fonction `absence_de_e(s)` qui prend en paramètre une chaîne de caractères `s` et renvoie `True` si `s` ne contient ni le caractère `e` ni `E`. Écrire des tests.

```
assert absence_de_e("")==True # On essaye la chaîne vide
assert absence_de_e("Salut")==True # On essaye un chaîne sans e
assert absence_de_e("eSalut")==False # On essaye e/E au début au mileu et à la fin
assert absence_de_e("SalEut")==False
assert absence_de_e("Salute")==False
```

```
def absence_de_e_if(s) :
    if nombre_apparitions('e',s) == 0  and nombre_apparitions('E',s) == 0 :
        return True
    else :
        return False
```

Il est inutile de faire un test pour renvoyer un booléen. Autant renvoyer directement de booléen !

```
def absence_de_e(s) :
    return nombre_apparitions('e',s) == 0 and nombre_apparitions('E',s) == 0
```

3. Si n est le nombre de caractères de `s`, quelle est la complexité [1] de votre solution ?
   Complexité : on lit 2n caractères systématiquement.
4. Proposez une autre solution qui n'a cette complexité que dans le cas où la fonction renvoie `True`, mais potentiellement une meilleure complexité quand elle renvoie `False`.
   Si on peut répondre (`False`) avant d'avoir lu tout `s`, on le fait.

```
def absence_de_e_efficace(s) :
    for i in range(len(s)) :
        if s[i] in 'eE' :
            return False
    return True
```

Exercice 5 — Entrelacement (⋆⋆)

Écrivez une fonction `entrelacement(s1,s2)` qui prend en paramètres deux chaînes de caractères `s1` et `s2` de même longueur et qui renvoie la chaîne qui contient en alternance un caractère de `s1` suivi d'un caractère de `s2`. Par exemple, `entrelacement('abc','123')` renvoie `'a1b2c3'`.

```
def entrelacement(s1,s2) :
    # Le assert est facultatif. Il permet de s'assurer que la condition est bien vérifiée.
    assert len(s1) == len(s2)
    res = ''
    for i in range(len(s1)) :
        res = res + s1[i] + s2[i]
    return res
```

Exercice 6 — Ponctuation (⋆⋆)

Écrivez une fonction `est_bien_ponctuée(s)` qui prend en paramètre une chaîne de caractères `s` et renvoie `True` si chaque point qui apparait dans la chaîne de caractères est suivi d'un espace, ou sinon est le dernier caractère de la chaîne.

```
>>> est_bien_ponctuée('Un. Deux.')
True
```

```
>>> est_bien_ponctuée('Trois.Quatre. ')
False
```

*Remarque : il est important de tester d'abord `i != len(s)-1` **avant** `s[i+1] != ' '`. Sinon, lorsque `i` sera égale à `len(s)-1`, l'évaluation de `s[i+1]` conduira à une erreur.*

```
def est_bien_ponctuée(s) :
    for i in range(len(s)) :
        if s[i] == '.' and i != len(s) - 1 and s[i+1] != ' ' :
            return False
    return True
```

une autre façon de l'écrire, qui n'utilise pas le connecteur logique `and`.

```
def est_bien_ponctuée_bis(s) :
    for i in range(len(s)) :
        if s[i] == '.' :
            if i != len(s) - 1 :
                if s[i+1] != ' ' :
                    return False
    return True
```

1. Pour évaluer la complexité, on évaluera le nombre d'accès mémoire : lecture/écriture de variable, lecture d'un caractère dans une chaîne de caractères, etc.

Exercice 7 — Pangrammes (⋆⋆)

Écrivez une fonction `est_pangramme(s)` qui prend en paramètre une chaîne de caractères `s` et qui renvoie `True` si `s` contient toutes les lettres de l'alphabet (on ne tient pas compte des caractères accentués).

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> est_pangramme(alphabet)
True
>>> est_pangramme('Portez ce vieux whisky au juge blond qui fume.')
True
```

Indication : vous pourrez utiliser la chaîne de caractères contenue dans la variable alphabet ci-dessus, ainsi que la méthode `s.lower()`, ou (plus compliqué) vous générerez vous même la chaîne `alphabet` à l'aide de `chr` et `ord`.

```python
def est_pangramme(s) :
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    s = s.lower()
    # on aurait pu écrire "for i in range(len(alphabet))" puis s[i]
    # au lieu de c dans la suite
    for c in alphabet :
        if not (c in s) :
            return False
    return True
```

En générant l'alphabet :

```python
def est_pangramme_bis(s) :
    s  = s.lower()
    for i in range(26):
        if chr(ord('a') + i) not in s :
            return False
    return True
```

Exercice 8 — Parenthèses (⋆ ⋆ ⋆)

Un mot `w` est bien parenthésé si l'une des trois conditions suivantes est satisfaite :

- `w == ''`
- `w == '(' + w1 + ')'` et `w1` est bien parenthésé
- `w == w1 + w2` et `w1`, `w2` sont bien parenthésés

Écrivez une fonction `est_bien_parenthesée(s)` qui prend en argument une chaîne de caractères `s` contenant uniquement les caractères `'('` et `')'` et qui renvoie `True` si `s` est bien parenthésée.

```python
def est_bien_parenthesée(s) :
    n = 0 # <- n est le nombre de parenthèses ouvertes non encore fermées dans s[:i]
    for i in range(len(s)) :
        if s[i] == '(' :
            n = n + 1
        elif s[i] == ')' :
            n = n - 1
            if n < 0 :
                return False
    return n == 0
```