Programmation en Python - Semestre 3

Institut Villebon - *Georges Charpak*

Premier semestre 2024 – 2025

Mise en situation 1/2

Mise en pratique :

- Chaque binôme, puis groupe de quatre étudiants, dispose de 5 à 10 cartes, alignées, face cachées, et mélangées.

But du jeu :

- Par des manipulations élémentaires (retourner une carte, échanger deux cartes), trier les cartes par ordre croissant en minimisant le nombre d'actions.

Règles du jeu :

- À aucun moment, il ne peut y avoir plus de deux cartes face visible sur la table.
- Il n'est permis de déplacer sur la table que des cartes visibles.

Mise en situation 2/2

Déroulement :

- Mélanger les cartes et les placer face cachée sur la table.
- Mettre au point une méthode permettant de trier les cartes
  - (5 minutes) en binôme.
  - (10 minutes) par groupes de 4.
- (5 minutes) Appliquer trois ou quatre fois votre (ou vos) méthode(s) de tris, tout en notant :
  - le nombre de comparaisons réalisées ;
  - le nombre de retournements réalisés ;
  - le nombre de déplacements de cartes réalisés.
- (5 minutes) Décrire cette méthode en quelques phrases aussi claires que possible.
- (25 minutes) Restitution en classe entière de quelques stratégies découvertes, et discussion.

Les listes en Python : le type `list`

Objectif : désigner avec une seule variable une collection de valeurs

Liste : suite indexée (numérotée) d'objets quelconques
(type `list` en python)

- Élements "rangés" dans des "cases" numérotées de 0 à $n-1$
- En mémoire : tableau à n cases, chacune contenant une référence (*"flèche"*) vers une valeur
- Peut contenir des objets de plusieurs types différents
- Mutable : peut être modifiée, agrandie, raccourcie, ...

Création et affichage

- Création : suite entre `[` et `]` d'expressions séparées par `,`

```
>>> print(lst)
[3, 'toto', 4.5, False, None]
```

- Liste vide `[]` : liste ne contenant aucun objet

```
>>> lst2 = []
>>> print(lst2)
[]
```

- Une liste peut contenir d'autres listes !

```
>>> lst = ['test', [1, [2], 3]]
>>> print(lst)
['test', [1, [2], 3]]
```

- Exemple

Accès aux éléments

- Accès à un élément donné : indexation

```
>>> lst = [3, 'toto', 4.5]
>>> lst[1]
'toto'
```

- Attention : les éléments sont indexés (numérotés) à partir de 0
  L'accès à un indice supérieur ou égal au nombre d'éléments provoque une erreur

Accès aux éléments d'une liste

Accès à une "tranche" : *slicing*

- Syntaxe : `lst[i:j]` construit une liste contenant les éléments d'indices `i` à `j-1` de `lst`
- Attention, l'élément d'indice `i` est inclus mais celui d'indice `j` est exclu !
- Si `i` est omis, il prend la valeur par défaut `0`
- Si `j` est omis, il prend la valeur par défaut `len(lst)`

Exemple

```
>>> lst = [3, 'toto', 4.5]
>>> lst[1:len(lst)]
['toto', 4.5]
>>> lst[0:1]
[3]
```

```
>>> lst[:len(lst)-1]
[3, 'toto']
>>> lst[:]
[3, 'toto', 4.5]
```

Opérations de base sur les listes

- Longueur d'une liste : fonction `len`

```
>>> lst = [3, 'toto', 4.5, False, None]
>>> len(lst)
5
>>> len([])
0
```

- Concaténation

```
>>> [3, 'toto', 4.5] + [False, None]
[3, 'toto', 4.5, False, None]
>>> [] + [3, 'toto', 4.5] + []
[3, 'toto', 4.5]
```

- Répétition

```
>>> 3 * ['a', 'b']
['a', 'b', 'a', 'b', 'a', 'b']
>>> [0] * 13
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Modification d'un élément

- Les listes sont Mutable : elles peuvent être modifiées, agrandies, raccourcies, ...
- On peut modifier le i-ème élément de `lst` par une affectation :

```
>>> lst = [3, 'toto', 4.5, False, None]
>>> lst[2]
4.5
>>> lst[2] = 'titi'
>>> lst
[3, 'toto', 'titi', False, None]
```

Attention, ceci ne crée pas une nouvelle liste mais modifie la liste sur place !

```
>>> lst = [3, 'toto', 4.5, False, None]
>>> lst_bis = lst
>>> lst[2] = 'titi'
>>> lst_bis
[3, 'toto', 'titi', False, None]
```

Agrandir ou rétrécir une liste

Plusieurs instructions ont un effet sur la taille de la liste :

- L'instruction `lst.append(elem)` ajoute l'élément `elem` à la fin de la liste `lst`.
- L'instruction `lst.pop()` supprime le dernier élément de `lst` et renvoie sa valeur.
- L'instruction `lst.pop(i)` supprime le `i`-ème élément de `lst` et renvoie sa valeur.

  append *et* pop *sont appelées méthodes, ou fonctions s'appliquant à un objet (nous en verrons probablement d'autres dans les cours suivants)*

Attention, ces instructions ne créent pas une nouvelle liste mais modifient la liste sur place !
Attention, ne pas confondre `x = lst[2]` et `x = lst.pop(2)` !

Agrandir ou rétrécir une liste

Exemple

```
>>> lst = [3, 'toto', 4.5, False, None]
>>> lst_bis = lst
>>> lst.append(1)
>>> print(lst_bis)
[3, 'toto', 4.5, False, None, 1]
>>> elem = lst_bis.pop(2)
>>> print(elem)
4.5
>>> print(lst)
[3, 'toto', False, None, 1]
```

Parcours de listes 1/3

Exercice : Écrire une fonction qui affiche tous les éléments d'une liste (un par ligne)

Parcours de listes 1/3

Exercice : Écrire une fonction qui affiche tous les éléments d'une liste (un par ligne)

```
def affiche_liste(lst):
    i = 0
    while i < len(lst):
        print(lst[i])
        i = i + 1
```

```
def affiche_liste(lst):
    for i in range(len(lst)):
        print(lst[i])
```

```
>>> lst = [3, 'toto', 4.5, False, None]
[3, 'toto', 4.5, False, None]
>>> affiche_liste(lst)
3
toto
4.5
False
None
```

- Voir le fonctionnement en interne

Parcours de listes 2/3

- syntaxe simplifiée pour parcourir un à un tous les éléments d'une séquence.

```
def affiche_liste(lst):
    for i in range(len(lst)):
        print(lst[i])
```

```
def affiche_liste(lst):
    for elt in lst:
        print(elt)
```

```
>>> lst = [3, 'toto', 4.5, False, None]
[3, 'toto', 4.5, False, None]
>>> affiche_liste(lst)
3
toto
4.5
False
None
```

- Voir le fonctionnement en interne

Parcours de listes 3/3 : la vérité sur range

- A t-on besoin d'avoir accès au i-ième élément d'une liste ?

Accès à `lst[i]`

```
def affiche_liste(lst):
    for i in range(len(lst)):
        print(lst[i])
```

Pas d'accès à `lst[i]`

```
def affiche_liste(lst):
    for elt in lst:
        print(elt)
```

- La fonction `range` fabrique des intervalles d'entiers
  - `range(i)` : entiers de 0 à `i-1`
  - `range(i, j)` : entiers de `i` à `j-1`
  - `range(i, j, k)` : entiers de `i` à `j-1` par pas de `k`
- Opérations autorisées sur un `range r`:
  - `x in r`, `x not in r`, `r[i]`
  - `len(r)`, `min(r)`, `max(r)`, `r.index(x)`, `r.count(x)`
  - `list(r)` (conversion en liste)

Test d'appartenance

Remarque : On pourrait coder cette fonctionnalité "à la main" mais elle existe déjà en Python

- `val in lst` vaut `True` si `val` apparaît dans `lst`, `False` sinon
- Réciproquement, on peut écrire `val not in lst`

```
>>> lst = ['Hildegarde', 'Cunégonde', 'Médor']
>>> 'Cunégonde' in lst
True
>>> 'Rex' in lst
False
>>> 'Rex' not in lst
True
>>> if 'Médor' in lst:
        print('Bon chien !')
Bon chien !
```