Les Structures de Données Python
Cours 1 : Les structures de données séquentielles

Halim Djerroud

ESIEE
PARIS

révision : 0.1

Plan

1. Définition des structures de données séquentielles
2. Les Listes
3. Les Tuples
4. Les intervalles
5. Les chaînes de caractères (*strings*)

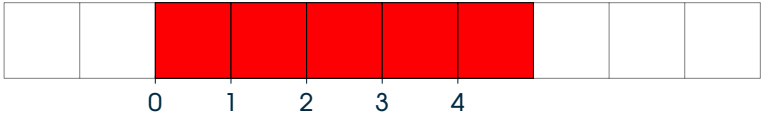
Les types en python

En fonction du nombre d'éléments stockés dans une variable, il existe deux catégories de types :

1. Les types primitifs
   - int, float, bool, str
2. Les types non primitifs (collections)
   - Dictionnaire
   - Set
   - **Séquence**
     - Liste
     - Tuples
     - String

Les structures de données séquentielles

1. Les informations sont stockées en mémoire d'une façon séquentielle (les cases sont adjacentes)



2. Parmi les structures de données séquentielles on trouve : Les listes, les tuples, les chaînes de caractères, les intervalles ...
3. Elles ont toutes une longueur
4. Les éléments de ces structures peuvent être désignés par un indice
5. On peut effectuer des traitements itératifs

Définition

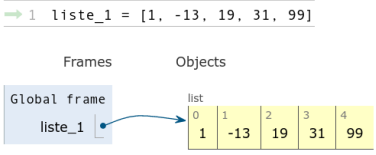
- Une liste : Une variable structurée regroupant des données les unes à la suite des autres
- Chaque valeur est identifiée par sa position dans la liste.

Exemple :

```
liste_1 = [1, -13, 19, 31, 99]     # liste d'entiers
liste_2 = [-1.1, 1.15, 5.4, 3.14] # liste de floats
liste_3 = ["Paris", "Brest", "Lyon"] # liste de strings
```

Déclaration d'une liste

- Python présente les listes comme une suite de valeurs séparées par des virgules, entourées par des crochets. Un indice (position) est associé à chaque valeur de la liste.
- Les indices sont attribués dans un ordre séquentiel croissant de gauche à droite et commencent par 0.
- Le nom de la variable fonctionne comme un pointeur vers l'espace mémoire où sont stockées les valeurs (la première valeur)



Caractéristiques des listes

- Seuls les listes et les tuples peuvent contenir des éléments de types hétérogènes. Les chaînes de caractères ne contiennent que des caractères, et les intervalles que des nombres entiers.

  Exemple :

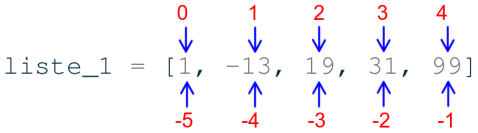
```
liste_1 = [1, -13, "Paris", 12.60, "ESIEE"]
```

- Liste vide :

```
liste_1 = []  # Creation d'une liste vide
```

Les indices (*indexing*)

- Les valeurs d'une liste sont accessibles via un index qui commence par 0
- Les valeurs d'une liste sont également accessibles de la fin (droite) vers le début (gauche) en utilisant des indices négatifs.
  - -1 est l'indice du dernier élément
  - -2 est l'indice de l'avant dernier élément ...



Exemple :

```
liste_1[0]          # =>   1
liste_1[1]          # => -13
liste_1[-1]         # =>  99
liste_1[-2]         # =>  31
```

Découpage d'une liste (*Slicing*)

Le *Slicing* permet de découper une liste afin d'en extraire une partie

- Soit la liste : `list_name=[x1, x2, ...., xn]`
- `list_name[n:p]` retourne la liste des éléments numérotés de `n` à `p-1`
- `list_name[:]` retourne tout les éléments de la liste
- `list_name[n:]` retourne la liste des éléments de n jusqu'à la fin de la liste
- `list_name[:p]` retourne la liste des éléments numérotés de `0` (début de la liste) à `p-1` (tous les éléments jusqu'au (`p-1`) nième.)

Il est possible aussi de rajouter un **pas** :

- `list_name[n:p:i]` retourne la liste des éléments numérotés de `n` à `p-1` avec un pas de *i*

Quelques exemples de Slicing

Exemple : Soit la liste : `liste_1=[1, -13, 19, 31, 99, 12, 15 ,48]`

```
#index:  0   1   2   3   4   5   6   7
#index: -8  -7  -6  -5  -4  -3  -2  -1
```

```
liste_1[:]      # ?
liste_1[3:]     # ?
liste_1[:5]     # ?
liste_1[3:5]    # ?
liste_1[3:-3]   # ?
liste_1[3:-3]   # ?
liste_1[::1]    # ?
liste_1[::-1]   # ?
liste_1[2:6:2] # ?
```

Quelques exemples de Slicing

Exemple : Soit la liste :

```
liste_1=[1, -13, 19, 31, 99, 12, 15 ,48]
#index:  0   1   2   3   4   5   6   7
#index: -8  -7  -6  -5  -4  -3  -2  -1
```

```
liste_1[:]      # => [1, -13, 19, 31, 99, 12, 15, 48]
liste_1[3:]     # => [31, 99, 12, 15, 48]
liste_1[:5]     # => [1, -13, 19, 31, 99]
liste_1[3:5]    # => [31, 99]
liste_1[3:-3]   # => [31, 99]
liste_1[3:-3]   # => [31, 99]
liste_1[::1]    # => [1, -13, 19, 31, 99, 12, 15, 48]
liste_1[::-1]   # => [48, 15, 12, 99, 31, 19, -13, 1]
liste_1[2:6:2] # => [19, 99]
```

Opérations sur les listes (addition)

- L'addition de deux listes : permet de concaténer deux listes avec l'opérateur **+**

Exemple :

```
liste_1 = [2, 4, 6]
liste_2 = [10, 12, 14]
liste_3 = liste_1 + liste_2
# liste_3 => [2, 4, 6, 10, 12, 14]
```

Seul l'opérateur arithmétique **addition (+)** est supporté, il n'est pas possible de soustraire ou de multiplier des listes entre elles

Opérations sur les listes (multiplication)

- Il est possible de multiplier une liste par un entier
  nbr * liste = Liste plus longue avec comme contenu les valeurs de Liste repetees *nbr* fois

Exemple :

```
liste_1 = [2, 4]
liste_2 = liste_1  * 3
# liste_2 =>  [2, 4, 2, 4, 2, 4]
```

Cette technique est souvent utiliser une initialiser une longue liste à 0 par exemple :

```
liste_1 = [0] * 10
# liste_1 => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Opérations sur les listes (in)

- L'opérateur **in** permet tester l'existence d'une valeur dans une liste

```
if valeur in nom_liste
```

Exemple :

```
villes = ["Paris", "Brest", "Lyon", "Marseille"]
print("Saisir votre ville :")
ville = input()
if ville in villes:
    print("Votre ville existe dans la liste")
else:
    print("Votre ville n'existe pas dans la liste")
```

Parcourir une liste (boucle)

- Il est possible de parcourir les éléments d'une liste grace à la boucle **for** en utilisant l'opérateur **in**

Exemple :

```
villes = ["Paris", "Brest", "Lyon", "Marseille"]
for ville in villes:
    print(ville, end=" ")


# => Paris Brest Lyon Marseille
```

Parcourir une liste (boucle)

- Il est possible de parcourir les éléments d'une liste grace à la boucle **for** en utilisant l'index des éléments

Exemple :

```
villes = ["Paris", "Brest", "Lyon", "Marseille"]
for i in range (len(villes)):
    print(villes[i], end=" ")


# => Paris Brest Lyon Marseille
```

Parcourir une liste (boucle)

- La fonction **enumerate** permet de renvoyer des tuples de chacun des éléments liste avec son index

Exemple :

```
villes = ["Paris", "Brest", "Lyon", "Marseille"]
for index, ville in enumerate(villes):
    print(index, ':', ville, end=" ")


# => 0 : Paris 1 : Brest 2 : Lyon 3 : Marseille
```

Parcourir une liste (boucle)

- La fonction **zip** permet de renvoyer des tuples de chacun des éléments des deux listes
- La boucle for s'arrête à la fin de la plus petite des deux listes

Exemple :

```
dep_num = [75, 29, 69, 13]
villes  = ["Paris", "Brest", "Lyon", "Marseille"]
for dep, ville in zip(dep_num, villes):
    print(dep, ':', ville, end=", ")


# => 75 : Paris, 29 : Brest, 69 : Lyon, 13 : Marseille,
```

Les méthodes des listes

Il existe de nombreuses méthodes applicables sur les listes parmi elles, on trouve :

Méthode	**Decription**
append	Ajout d'un élément à la liste
insert(i, x)	Insérer l'élément `x` à la position `i`
remove(x)	Supprime le premier élément de la liste dont la valeur est égale à x
clear	Supprimer tous les éléments de la liste
copy	Renvoie une copie superficielle de la liste
extend	Étendez la liste en ajoutant tous les éléments de l'itérable
index(x(,début(,fin))	Renvoie l'index dans la liste du premier élément dont la valeur = x
sort	Trier les éléments de la liste en place

Les tuples

- Les **tuples** ressemblent aux listes
- Les valeurs d'un **tuple** avec un couple de parenthèses même si cela n'est pas obligatoire
- Les **tuples** peuvent contenir différents types de valeurs (des nombres, des chaînes de caractères, des flottants ...)
- La différence entre un **tuple** et une **liste** est qu'un **tuple** est une donnée non non modifiable (**immuable**) à la différence d'une liste qui est modifiable.

```
tuple_1 = 2, 4, 6          # => (2, 4, 6)
tuple_2 = (2, 4, 6)        # => (2, 4, 6)
tuple_3 = ("Un", 2, 3.0)   # => ('Un', 2, 3.0)
```

Tuple vide ou à une valeur

- Pour créer un tuple vide, on utilisera une paire de parenthèses vides
- Pour créer un tuple avec une seule valeur, il faudra faire suivre cette valeur d'une virgule.

Exemple :

```
tuple_vide = ()         # => ()
tuple_unique = (5,)     # => (5,)
```

Déballage de séquence

- Une façon rapide d'affecter les différentes valeurs d'un tuple dans des variables séparées

Exemple :

```
enregistrement = ("Lea", 5, ["Tennis", "Natation", "Dance"])
prenom, age, sport = enregistrement
        # prenom => 'Lea'
        # age    => 5
        # sport  => ['Tennis', 'Natation', 'Dance']
```

Définition des chaînes de caractères (*Strings*)

- Les chaînes de caractères peuvent être considérées comme des listes (de caractères) particulières
- Chaque caractère est accessible par sa position dans la chaîne de caractères
- Les indices des caractères commencent à 0
- Donc :
  - Il est possible de parcourir une chaîne de caractères comme une liste
  - Il est possible d'utiliser les *slices* sur les chaînes de caractères
  - Il est possible d'accéder à un caractère d'une chaîne de caractère via son indice
- Mais :
  - Il n'est pas possible de modifier un caractère d'une chaîne de caractère

ASCII code

- Codage «universel» permettant d'associer à chaque caractère un numéro
- Tous les caractères (non accentués) sont réunies dans la table ASCII
- Le reste des caractères sont codé dans la table ASCII étendue

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	`	j	t	~
7:	%	/	9	C	M	W	a	k	u	DEL
8:	&	0	:	D	N	X	b	l	v	
9:	'	1	;	E	O	Y	c	m	w	

La table ASCII

- À chaque caractère est associé un code hexadécimal et son équivalent en décimal
- Les lettres majuscules ont des codes ASCII différents des lettres minuscules
- Les codes des lettres majuscules sont dans l'intervalle décimal (65, 90)
- Les codes des lettres minuscules sont dans l'intervalle décimal (97, 122)
- La différence entre une lettre majuscule d'une lettre minuscule est toujours 32

Code ASCII d'un caractère

- En Python, la fonction ord(caractère)donne le code décimal d'un caractère

```
print("le code de la lettre a est :", ord("a")) # => 97
print("le code de la lettre A est :", ord("A")) # => 65
print("le code du caractere # est :", ord("#")) # => 35
```

Caractère correspondant à un code ASCII

- En Python, la fonction chr(code)donne le caractère correspondant au code.

```
print("Le caractere correspondant au code 65 :",chr(65)) # => A
print("Le caractere correspondant au code 97 :",chr(97)) # => a
print("Le caractere correspondant au code 35 :",chr(35)) # => #
```