Université de Gabés
Faculté des Sciences de Gabés

LGLSI 1

Semestre 2

Programmation Python

Dhikra KCHAOU
dhikrafsegs@gmail.com

2023-2024

Chapitre 4

Les séquences dans Python

Les listes, les tuples et les chaines de caractères

Plan du chapitre

1. **Les listes**
   - Opérations sur les listes
   - Fonctions associées aux listes
   - Fonction range() et list()
2. **Les tuples**
3. **Les chaines de caractères**

Définition d'une liste

- Une **liste** est une structure de données qui contient une série de valeurs.
- Python autorise la construction de **liste** contenant des **valeurs de types différents** (par exemple entier et chaîne de caractères).
- Une liste est déclarée par une série de valeurs séparées par des virgules, et le tout encadré par des crochets.

```
>>> animaux=["girafe","tigre","singe","souris"]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> tailles=[5,2.5,1.75,0.15]
>>> tailles
[5, 2.5, 1.75, 0.15]
>>> mixte =["girafe",5,"tigre",2.5,"singe",1.75,"souris",0.15]
>>> mixte
['girafe', 5, 'tigre', 2.5, 'singe', 1.75, 'souris', 0.15]
>>>
```

Accès aux éléments d'une liste

- L'accès à un élément d'une liste se fait à travers sa position. Ce numéro est appelé **indice** (ou **index**) de la liste.

```
liste  : ["girafe", "tigre", "singe", "souris"]
indice :        0         1         2         3
```

- Les indices d'une liste de n éléments commence à 0 et se termine à n-1

```
>>> animaux=["girafe","tigre","singe","souris"]
>>> animaux[0]
'girafe'
>>> animaux[4]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    animaux[4]
IndexError: list index out of range
>>>
```

Opérations sur les listes

- les listes supportent l'opérateur **+** de concaténation, ainsi que l'opérateur ***** pour la duplication:

```
>>> ani1 = ["girafe", "tigre"]
>>> ani2 = ["singe", "souris"]
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

Ajout d'éléments à une liste

- Une liste vide se définit par **[ ]** ou **list()**. Pour créer une liste vide:

```
L=[]
L
[]
```

```
L=list()
L
[]
```

- Pour ajouter deux éléments, l'un après l'autre, d'abord avec la concaténation :

```
L=L+[15]
L
[15]
L=L+[-5]
L
[15, -5]
```

- Ou bien avec la méthode append() :

```
L.append(13)
L
[15, -5, 13]
```

Ajout d'éléments à une liste

- **Remarque**: la méthode **list.append()** prend exactement un argument.

```
>>> L=[]
>>> L=L+[5,8]
>>> L
[5, 8]
>>> L.append(5)
>>> L
[5, 8, 5]
>>> L.append('a',8)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    L.append('a',8)
TypeError: list.append() takes exactly one argument (2 given)
>>>
```

Indexing négatif

- La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
liste           : ["girafe", "tigre", "singe", "souris"]
indice positif  :        0        1        2         3
indice négatif  :       -4       -3       -2        -1
```

- Les indices négatifs reviennent à compter **à partir de la fin**.
- Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice **-1** sans pour autant connaître la longueur de cette liste.
- L'avant-dernier élément posséde l'indice -2, l'avant-avant dernier l'indice -3, etc.

Tranches « slicing »

- On peut récupérer une tranche d'une liste en suivant le modèle:

Liste [début : fin : pas]

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
```

- **Remarque**: lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

```
>>> animaux[0:]

>>> animaux[:]

>>> animaux[1:]

>>> animaux[1:-1]
```

Préciser le pas

- On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier.

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]

>>> x[::2]

>>> x[::3]

>>> x[1:6:3]

```

Exemples

```
>>> L= [11, 'abc', 'ddd',4,12]
>>> print(L[1])

>>> print(L[-1])

>>> print(L[:])

>>> print(L)
```

```
>>> print(L[2:])

>>> print(L[:3])

>>> print(L[-1:-3])
```

```
>>> print(L[:-3])

>>> print(L[-3:-1])

>>> print(L[::2])

>>> print(L[1:3:2])
```

print (L[: :-1]) ??

```
>>> L=[11,'abc', "ddd", 4,12]
>>> L[::-1]
[12, 4, 'ddd', 'abc', 11]
```

Fonctions associées aux listes

- **len(L)** renvoie la longueur de la liste L
- **min(L)** renvoie l'élément minimum dans la liste L
- **max(L)** renvoie l'élément maximum dans la liste L
- **sum(L)** renvoie la somme des éléments de la liste L
- **L.count(elem)** compte le nombre de l'élément elem dans la liste L
- **list** & **range** : génère une liste d'entiers
- **elem in L**, **elem not in L** : vérifie l'appartenance d'un élément elem dans L
- **L.append(elem)** ajoute l'élément elem à la fin de la liste L
- **L.insert(ind, elem)** insère l'élément elem dans l'indice ind de L

Fonctions associées aux listes

- **L.extend(seq)** ajoute le contenu d'une séquence seq à L
- **L.sort()** trie la liste L par ordre croissant
- **sep.join(L)** convertie la liste L en une chaine de caractères en ajoutant un séparateur sep entre chaque élément
- **L.remove(elem)** supprime le premier élément trouvé elem de la liste L (s'il existe)
- **L.pop()** supprime et retourne le dernier élément de L
- **L.pop(i)** supprime et retourne l'élément à la position i de L
- **L.reverse()** inverse l'ordre des éléments de la liste L
- **reversed (L)** affiche la liste inversé de L sans l'affecter

Exemples sur les fonctions associées aux listes

```
l1=["AB",20,6,1]
print(len(l1))   # 4
print(max(l1))   ⊗

l2=[11,46,1,1]
print(max(l2))  # 46
print(sum(l2))  # 59
print(l2.count(1))# 2

l3=list(range(1,4))
print(l3)   # [1, 2, 3]
```

- **Remarque**: les fonctions min()et max() peuvent prendre plusieurs arguments entiers et / ou floats.

```
>>> min(2,1.5)
1.5
```

- Attention! On ne peut pas déterminer le min ou le max d'une liste qui mélange entiers et chaines de caractères. cela renvoie une erreur :

```
>>> L= [11, 'abc', 'ddd',4,12]
>>> max(L)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    max(L)
TypeError: '>' not supported between instances of 'str' and 'int'
```

Exemples sur les fonctions

```
l=["AA",20,31]
print("AA" in l)     # True
print(13 not in l) # True

l.append(20)
print(l)    # ['AA', 20, 31, 20]
l.insert(2,"kk")
print(l)  # ['AA', 20, 'kk', 31, 20]

l.sort() ⊗
l1=["BB","DD","AA"]
l1.sort()
print(l1) # ['AA', 'BB', 'DD']
l1.sort(reverse=True)
print(l1) # ['DD', 'BB', 'AA']

ch="-".join(l1)
print(ch) # DD-BB-AA
```

```
l=[14, 7, 12.1, 7, "KK","DD"]
l.remove(3) ⊗
l.remove(7)
print(l) # [14, 12.1, 7, 'KK', 'DD']

val1=l.pop()
val2=l.pop(2)
print(val1, val2, l)
        # DD 7 [14, 12.1, 'KK']
l.reverse()
print(l)  # ['KK', 12.1, 14]

reversed(l)
print(l)    # ['KK', 12.1, 14]
for i in reversed(l):
    print(i, end=" ") # 14 12.1 KK
```

La fonction range() et list()

- **Rappel**: La fonction **range()** est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle.
- Lorsqu'elle est utilisée en combinaison avec la fonction list(), on obtient une liste d'entiers.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- La syntaxe de la fonction range() est:
  - **range([début,] fin[, pas])**
  - Les arguments entre crochets sont optionnels.
  - Les valeurs des arguments optionnels sont par défaut 0 pour début et 1 pour pas.

La fonction range() et list()

```
>>> list(range(10,0))
[]
```

- **Remarque**:
  - La liste est vide car Python a pris la valeur du pas par défaut qui est de 1.
  - Si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0. Python génère ainsi une liste vide.
  - Pour éviter ça, il faudrait, par exemple, préciser un pas de -1 pour obtenir une liste d'entiers décroissants :

```
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Parcours d'une liste

- **Parcours d'une liste :**

```
>>> L=[5,8,1]
>>> for i in L:
...     print(i)
...
...

5
8
1
```

Soit

```
>>> L=[5,8,1]
>>> for i in range(len(L)):
...     print(L[i])
...
...

5
8
1
```

```
>>> for i in range(len(L)):
...     print(i, end=" ")
...     print(L[i])
...
...

0 5
1 8
2 1
```

- **Parcours d'une tranche d'une liste:**

```
1 >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> for animal in animaux[1:3]:
3 ...     print(animal)
4 ...
5 tigre
6 singe
```

Parcours en utilisant indice + element

- 1ére méthode:

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> for i in range(len(animaux)):
...     print(f"L'animal {i} est un(e) {animaux[i]}")
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

- 2ème méthode: utilisation de la fonction **enumerate()**

enumerate_list.py

```
animaux = [" girafe ", " tigre ", " singe ", " souris "]
for i , animal in enumerate ( animaux ):
    print (f"L' animal {i} est un (e) { animal }")
```

```
L' animal 0 est un (e)  girafe
L' animal 1 est un (e)  tigre
L' animal 2 est un (e)  singe
L' animal 3 est un (e)  souris
```

Exercice

- Donner la différence entre ces deux codes:

```
L = [13, 12, 6, 9, 1]

for i in reversed(L):
    print(i)
print(L)
```

```
1
9
6
12
13
[13, 12, 6, 9, 1]
```

```
L = [13, 12, 6, 9, 1]
L2=list(reversed(L))
print(L)
print(L2)
```

```
[13, 12, 6, 9, 1]
[1, 9, 6, 12, 13]
```

Les tuples

Les tuples

- Les tuples (« n-uplets » en français) sont des **objets** séquentiels correspondant aux listes (**itérables, ordonnés** et **indexables**) mais ils sont toutefois non modifiables.
- L'intérêt des tuples par rapport aux listes réside dans leur immutabilité.
- Pratiquement, on utilise les parenthèses au lieu des crochets pour créer un tuple.

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> type(t)
<class 'tuple'>
>>> t[2]
3
>>> t[0:2]
(1, 2)
>>> t[2] = 15
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Les tuples

- L'affectation et l'indiçage fonctionnent comme avec les listes. Mais si on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur.
- Si vous voulez ajouter un élément (ou le modifier), vous devez créer un nouveau tuple :

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> id(t)
139971081704464
>>> t = t + (2 )
>>> t
(1, 2, 3, 2)
>>> id(t)
139971081700368
```

Les tuples

- Il est possible de créer un tuple sans les parenthèses, si ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

- Les opérateurs + et * fonctionnent comme pour les listes (concaténation et duplication) :

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 4
(1, 2, 1, 2, 1, 2, 1, 2)
```

Parcours d'un tuple

Parcours d'un tuple sur les élements

```
>>> t=(5,8,1)
>>> t
(5, 8, 1)
>>> for i in t:
...     print (i)
...
...

5
8
1
```

parcours d'un tuple en utilisant l'indice

```
>>> t=(5,8,1)
>>> for i in range(len(t)):
...     print(t[i])
...
...

5
8
1
```

Parcours d'un tuple en utilisant la fonction enumerate()

```
>>> t=(5,8,1)
>>> for nombre in enumerate (t):
...     print(nombre)
...
...

(0, 5)
(1, 8)
(2, 1)
```

Fonctions associées aux tuples

- len(tup) calcule la longueur du Tuple tup
- min(tup) retourne l'élément le plus petit dans tup
- max(tup) retourne l'élément le plus grand dans tup
- tup.count(elem) retourne le nombre d'occurrences de l'élément elem dans tup
- tup.index(elem) : retourne la première occurrence de l'élément elem s'il existe dans tup, Erreur sinon
- sep.join(tup) convertie le tuple en une chaine de caractères en ajoutant un séparateur sep entre chaque élément

Les tuples vs les listes

- Les tuples sont peu adaptés lorsqu'on a besoin d'ajouter, retirer, modifier des éléments.
- La création d'un nouveau tuple à chaque étape s'avère lourde puisque les tuples sont non modifiables. Pour ce genre de tâche, les listes sont clairement mieux adaptées.
- **Pourquoi utiliser alors les tuples plutôt qu'une liste?**
  - lorsque les données ne doivent pas être modifiées : un tuple est plus judicieux.
  - comme clés pour un dictionnaire(les listes ne peuvent pas)
  - comme valeurs de retour d'une fonction

Les chaines de caractères

Les chaines de caractères

- Les chaînes de caractères :
  - peuvent être considérées comme **des séquences** (de caractères).
  - représentent toutes les informations qui ne sont pas numériques
  - sont comprises entre guillemets simples ' ' , doubles " " et triples """ """

```
chaine1 = 'aaaa'
chaine2 = "aaaaaaaa"
chaine3 = """aaaaa bbbb cccccc
ddddd eeeee ppppppp"""

print (chaine1) ; print(chaine2) ; print(chaine3)
```

Textes sur plusieurs lignes

```
aaaa
aaaaaaaa
aaaaa bbbb cccccc
ddddd eeeee ppppppp
```

Attention !

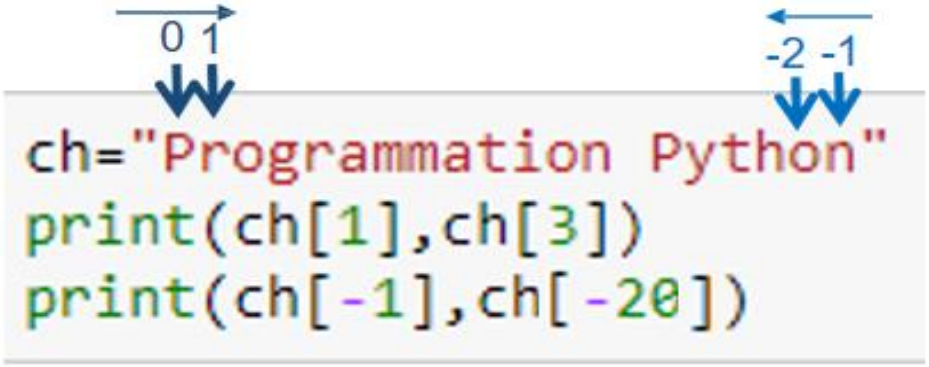
```
>>> print('aujourd'hui')
...
    SyntaxError: unterminated string literal (detected at line 1)
>>>
```

- **Solutions**:
  - Insérer caractère antislash « \ » ou bien
  - print ("Aujourd'hui")

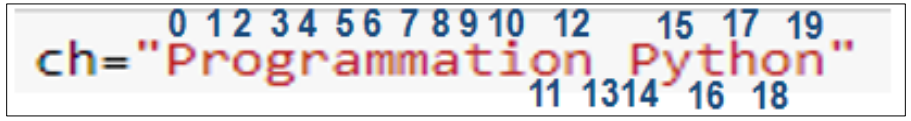
```
>>> print('Aujourd\'hui')
Aujourd'hui
```

Indexing et slicing

- Accès aux caractères d'une chaine:



- Slicing ou découpage en tranche [ind1:ind2] → ind1… ind2-1




```
print(ch[0:3])
print(ch[2:5])
print(ch[2:])
print(ch[2::4])
print(ch[2::-2])
```

```
# Pro
# ogr
# ogrammation Python
# omiPo
# oP
```

Concaténation, répétition et modification

- Les opérateurs + et * fonctionnent comme pour les listes (concaténation et duplication):

```
a="AA"+"BB"
b="AB"*3
print (a)    #AABB
print (b)    #ABABAB
```

- Modification d'un ou plusieurs caractère(s):

```
mot = "pytton"
print (mot[3])
mot[3]="h"
print (mot[3])
```

'str' object does not support item assignment

```
mot1 = "Programmation"
mot2 = " Python"
mot1=mot1+mot2
print (mot1)
```

Remarque : Une fois une chaîne de caractères est définie, vous ne pouvez plus modifier un de ses éléments, mais on peut définir une nouvelle chaîne avec le même identificateur.

Fonctions associées aux chaines de caractères

- len(ch) renvoie la longueur de ch
- ch.lower() convertit une chaîne ch en minuscule
- ch.upper() convertit une chaîne ch en majuscule
- ch.title() convertit en majuscule l'initiale de chaque mot dans ch
- ch.capitalize() convertit en majuscule seulement la première lettre de la chaîne ch
- ch.swapcase() convertit toutes les majuscules dans ch en minuscules, et vice-versa

Exemples

```
ch="Python Pour programmer"
print("Longueur de ch=",len(ch))
print(ch.lower())
print(ch.upper())
print(ch.capitalize())
print(ch.title())
print(ch.swapcase())
```

```
Longueur de ch= 22
python pour programmer
PYTHON POUR PROGRAMMER
Python pour programmer
Python Pour Programmer
pYTHON pOUR PROGRAMMER
```

Autres fonctions sur les chaines

- ch.find(sch) cherche la position d'une sous chaine sch dans la chaîne de caractères ch

```
012345
mot = "python"
print(mot.find ("y"))    #1
print(mot.find ("tho")) #2
print(mot.find ("a"))    #-1
print(mot.find ("hoi")) #-1
```

L'élément recherché est non trouvé

  - **Remarque** : Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurrence est renvoyé.

- ch.index(shr) retrouve l'indice de la première occurrence de la chaine shr dans la chaîne de caractères ch

```
print(mot.index("y"))
print(mot.index("yl"))
```

#1

ValueError: substring not found

```
"pypythonp".index("p",3)
```

#8

Autres fonctions sur les chaines

- ch.count(sch) compte le nombre d'une sous-chaine sch dans la chaîne de caractères ch

```
mot = "python onn"
print(mot.count("on"))  #2
print(mot.count("n"))   #3
```

- **sch in/ not in ch** vérifie si une sous-chaîne sch existe dans une chaine de caractères ch
- ch.startswith(scr) vérifie si une chaîne de caractères commence par une sous-chaine sch

```
mot = "Hi python"
print ("HI" in mot)              # False
print ("HI" not in mot)          # True
print (mot.startswith("Hi"))     # True
print (mot.startswith("Hi  "))   # True
```

Autres fonctions sur les chaines

- ch.split() découpe une chaîne de caractères en plusieurs éléments selon n'importe quelle combinaisons d'espaces blancs (' ', '\n', '\t')
- ch.split(sep) découpe une chaîne de caractères en plusieurs éléments selon une chaîne séparatrice sep

```
chaine= "AA DD;BB;CC  RR"
chaine1=chaine.split()
chaine2=chaine.split(";", maxsplit=1)
print (chaine1)  # ['AA', 'DD;BB;CC', 'RR']
print (chaine2)  # ['AA DD', 'BB;CC RR']
```

L'argument **maxsplit** indique le nombre de fois qu'on souhaite découper la chaine.

La fonction ch.isdigit()

- La fonction ch.isdigit() vérifie si la chaîne se compose uniquement de chiffres.

```
ch = "123456"
print (ch.isdigit())
```

```
ch = "AABfs2234BBDDS"
print (ch.isdigit())
```