

Institut Préparatoire aux Études d'Ingénieurs de Tunis



Leila Ben Othman

Première année de classes préparatoires
MP-PC

Année universitaire: 2016 - 2017

Les séquences en PYTHON

Plan

① Les séquences en PYTHON

- Initialisation/création

- Accès à un élément ou à une sous-séquence

- Opérations sur les séquences

- Récapitulatif

② Les listes

- Ajout dans une liste

- Extension d'une liste

- Suppression

- Modification

- Autres méthodes

- Création de liste

- Parcours de liste

- Copie superficielle/profonde d'un objet mutable

③ Les chaînes de caractères

④ Les tuples

⑤ Conversion-Transtypage

Les séquences: chaînes, listes et tuples

- Une **séquence** est une structure de données linéaire constituée d'une suite d'objets (éléments).
- Les séquences partagent tous le même modèle d'accès (via des **index**): un élément est accessible par son index. Un index désigne la position de l'élément dans la séquence.

Les séquences: chaînes, listes et tuples

Les séquences en PYTHON:

- Les **chaînes de caractères**: suite d'objets de type homogène (caractères), délimités par de simples ou doubles côtes.
- Les **listes**: suite d'objets de types hétérogènes, délimités par des crochets.
- Les **tuples**: suite d'objets de types hétérogènes, délimités par des parenthèses.

Les listes sont des objets **mutables** (modifiables), tandis que les chaînes de caractères et les tuples sont des objets **non mutables** (non modifiables).

Comment initialiser/créer une séquence?

Séquence vide

```
In [1]: L=list(); # ou bien L=[]
Out [1]: []
```

```
In [2]: C=str();C # ou bien C=""
Out [2]: ""
```

```
In [3]: T=tuple();T # ou bien
T=()
Out [3]: ()
```

Type d'une séquence

```
In [4]: type(L)
Out [4]: <class 'list'>
```

```
In [5]: type(C)
Out [5]: <class 'str'>
```

```
In [6]: type(T)
Out [6]: <class 'tuple'>
```

Comment initialiser/créer une séquence?

```
L=[1,2,3,'a','B', 'Bonjour',[5,6,'ici']] # une liste de 7 éléments
```

```
C='Bonjour les futurs ingénieurs'  
# une chaîne de caractères
```

```
T=(L,c,167,('A4',0)) # un tuple de 4 éléments
```

Taille - Accès à un élément de la séquence

- Une séquence peut contenir une autre séquence;
- On calcule la taille d'une séquence S (le nombre d'éléments) à l'aide la fonction `len` (`len(S)`);
- On accède à chaque élément de la séquence par son index:
 - $S[0]$: premier élément de S
 - $S[\text{len}(S) - 1]$: dernier élément de S
- Lorsque l'index est négatif, le parcours est effectué à rebours à partir de la fin:
 - $S[-1]$: premier élément de S (de droite vers la gauche).
 - $S[-\text{len}(S)]$: dernier élément de S (de droite vers la gauche).

Taille - Accès à un élément de la séquence

```
In [1]: len(L)  
Out [1]: 7
```

```
In [2]: L[0] # ou L[-len(l)]  
Out [2]: 1
```

```
In [3]: L[len(L)-1] # ou L[-1]  
Out [3]: [5,6,'ici']
```

```
In [4]: len(C)  
Out [4]: 29
```

```
In [5]: C[0] # ou C[-len(C)]  
Out [5]: 'B'
```

```
In [6]: C[len(C)-1] # ou C[-1]  
Out [6]: 's'
```

Taille - Accès à un élément de la séquence

```
In [7]: len(T)
```

```
Out [7]: 4
```

```
In [8]: T[0] # ou T[-len(T)]
```

```
Out [8]:
```

```
[1,2,3,'a','B','Bonjour',[5,6,'ici']]
```

```
In [9]: T[len(T)-1] # ou T[-1]
```

```
Out [9]: ('A4',0)
```

```
In [10]: L[-1][0]
```

```
Out [10]: 5
```

```
In [11]: T[0][-1]
```

```
Out [11]: [5,6,'ici']
```

```
In [12]: T[0][-1][1]
```

```
Out [12]: 6
```

Découpage ou Slicing

- PYTHON permet le découpage en tranches avec la syntaxe suivante $S[debut : fin]$
- $S[debut : fin]$ désigne la sous-séquence de S constituée des éléments dont les index sont compris entre $debut$ et $fin - 1$.
- Quand $debut$ n'est spécifié, alors la tranche commencera du début de S
- Quand fin n'est spécifié, alors la tranche finira à la fin de S
- La syntaxe $S[debut : fin : pas]$ possède un troisième paramètre (par défaut, égal à 1), indiquant le pas de sélection.

Découpage ou Slicing

```
In [1]: L[:]  
Out [1]: [1,2,3,'a','B','Bonjour',[5,6,'ici']]
```

```
In [2]: L[3:]  
Out [2]: ['a','B','Bonjour',[5,6,'ici']]
```

```
In [3]: L[:5]  
Out [3]: [1,2,3,'a','B']
```

```
In [4]: L[2:5]  
Out [4]: [3,'a','B']
```

```
In [5]: L[::2]  
Out [5]: [1,3,'B',[5,6,'ici']]
```

```
In [6]: L[:5]  
Out [6]: [1,2,3,'a','B']
```

Découpage ou Slicing

```
In [7]: L[-5:]  
Out [7]:  
[3,'a','B','Bonjour',[5,6,'ici']]
```

```
In [8]: L[:5]  
Out [8]:[1,2]
```

```
In [9]: L[-4:-1]  
Out [9]: ['a','B','Bonjour']
```

```
In [10]: L[-6::2]  
Out [10]: [2,'a','Bonjour']
```

```
In [11]: L[::-1]  
Out [11]:  
[[5,6,'ici'],'Bonjour','B','a',3,2,1]
```

Opération de concaténation

- L'opération de concaténation se note +
- Syntaxe: $S1 + S2$
- Cette opération permet de concaténer deux séquences **de même type**.
- Le résultat est une nouvelle séquence qui contient les contenus combinés de $S1$ et $S2$

```
In[12]: [1,2,3]+[4,5,6]  
Out[12]: [1,2,3,4,5,6]
```

Opération de répétition

- L'opération de répétition se note $*$
- Syntaxe: $S * n$ (n est un entier)
- Cette opération permet d'avoir une nouvelle séquence composée de n copies consécutives de S .

```
In[13]: 'Bonjour'*3  
Out[13]: 'BonjourBonjourBonjour'
```

Appartenance (in, not in)

- On peut vérifier si un objet est membre ou non d'une séquence en utilisant les opérateurs **in**/ **not in**.

```
In[14]: 'B' in c
```

```
Out[14]: True
```

```
In[15]: 'i' in L[6][2]
```

```
Out[15]: True
```

```
In[16]: 6 in L[6][1]
```

```
Out[16]: TypeError: argument of type 'int' is not iterable
```

Les méthodes `count`/`index`

- La méthode `L.count(obj)` retourne le nombre d'occurrence de `obj` dans `S`.
- La méthode `L.index(obj)` retourne la position de la première occurrence de `obj` dans `S`. Si `obj` n'existe pas, alors cette méthode génère une exception de type `ValueError`.

```
In[17]: L.count(6)
```

```
Out[17]: 0
```

```
In[18]: L.index(2)
```

```
Out[18]: 1
```

```
In[19]: L.index(6)
```

```
Out[19]: ValueError: 6 is not in list
```

Récapitulatif (1)

Opération	Fonction
<code>len(S)</code>	Nombre d'éléments de la séquence S
<code>S[i]</code>	Élément à la position i
<code>S[len(S)-1]</code>	Dernier élément de S
<code>S[:]</code>	Toute la séquence S
<code>S[a:]</code>	Éléments d'indice a jusqu'à la fin
<code>S[:b]</code>	Éléments d'indice 0 jusqu'à $b - 1$
<code>S[a:b]</code>	Éléments d'indice a jusqu'à $b - 1$
<code>S[a:b:c]</code>	Éléments d'indice a jusqu'à $b - 1$ (pas= c)
<code>S[:n]</code>	Les n premiers éléments de S
<code>S[-n:]</code>	Les n derniers éléments de S
<code>S[::-1]</code>	La séquence S inversée

Récapitulatif (2)

Opération	Fonction
$S * n$	S répétée n fois
$S1 + S2$	Concatène les séquences $S1$ et $S2$
$obj \text{ in } S$	Teste si obj est membre de S
$obj \text{ not in } S$	Teste si obj n'est pas membre de S
$S.count(obj)$	Retourne le nombre d'ocurences de obj dans S
$S.index(obj)$	Retourne la position de obj dans S . Si obj n'est pas membre de S , cette méthode génère une erreur.

Les listes

- Une liste en PYTHON est une structure de données **mutable**, c'est-à-dire qu'on peut modifier son contenu sans changer son adresse mémoire.
- Modifier une liste $\Rightarrow$ ajouter/remplacer/supprimer un élément ou une suite d'éléments.

Ajout

- Les méthodes *append* et *insert* permettent d'ajouter un élément dans une liste.
- Ces méthodes modifient la liste sur laquelle s'applique la méthode.
 - *L.append(obj)* ajoute *obj* à la fin de *L*.
 - *L.insert(i, obj)* ajoute *obj* à la position *i* dans *L*.

```
In [1]: L=[1,2,3]
```

```
In [2]: L.append('cc')
```

```
In [3]: L
```

```
Out [3]: [1,2,3,'cc']
```

```
In [4]: L.insert(1,('bb',5))
```

```
In [5]: L
```

```
Out [5]: [1,('bb',5),2,3,'cc']
```

Extension d'une liste

- La méthode *extend* permet d'étendre le contenu d'une liste.
- *L.extend(seq)* ajoute le contenu de *seq* (itérable) à la fin de *L*

```
In [1]: L=[1,2,3]
```

```
In [2]: L.extend('Bonjour')
```

```
In [3]: L
```

Out [3]:

```
[1,2,3,'B','o','n','j','o','u','r']
```

In [4]:

```
L.extend(('coucou',4))
```

In [5]: L

Out [5]:

```
[1,2,3,'B','o','n','j','o','u','r','coucou',4]
```

```
In[6]: L.extend(67)
```

Out[6]: **TypeError: 'int' object is not iterable**

Suppression

- La méthode `L.remove(obj)` supprime la première occurrence de `obj`;
- La méthode `L.pop()` supprime le dernier élément dans `L` et le retourne;
- La méthode `L.pop(i)` supprime l'élément à la position `i` et le retourne;

```
In [1]: L=[1,2,3,4,5,6]
```

```
In [2]: L.remove(1)
```

```
In [3]: L  
Out[3]: [2,3,4,5,6]
```

```
In [4]: L.pop()  
Out [4]: 6
```

```
In [5]: L  
Out [5]: [2,3,4,5]
```

```
In [6]: L.pop(1)  
Out[6]: 3
```

```
In [7]: L  
Out [7]: [2,4,5]
```

Suppression

Suppression d'un élément

In [1]: L=[1,2,3,4,5,6]

In [2]: del L[4] # equivalent à L.pop(4)

In [3]: L

Out [3]: [1,2,3,4,6]

Suppression de plusieurs éléments

In [4]: del L[2:4]

In [5]: L

Out [5]: [1,2,6]

Modification

Modification d'un élément

```
In [1]: L=[1,2,3,4,'a','b','c','d']
```

```
In [2]: L[4]='A'
```

In [3]: L

Out [3]: [1,2,3,4,'A','b','c','d']

Modification de plusieurs éléments

In [4]: $L[2:4] = 30, 40$

In [5]: L

Out [5]: [1,2,30,40,'A','b','c','d']

Autres méthodes

- `L.sort()` trie les éléments de `L`
- `L.reverse()` inverse les éléments de `L`

```
In [1]: L=[1,8,3,4,9,5,2]
```

```
In [2]: L.sort()
```

```
In [3]: L
```

```
Out [3]: [1,2,3,4,5,8,9]
```

```
In [4]: L.reverse()
```

```
In [5]: L
```

```
Out [5]: [9,8,5,4,3,2,1]
```

Création de liste - par énumération

Par énumération en utilisant la méthode *list(iterable)*

```
In [1]: L=list(range(0,11))
```

```
In [2]: L
```

```
Out[2] [0,1,2,3,4,5,6,7,8,9,10]
```

```
In [3]: L=list(range(0,11,2))
```

```
In[4]: L
```

```
Out [4]: [0,2,4,6,8,10]
```

```
In [5]: L=list('Bonjour')
```

```
In [5]: L
```

```
Out [5]: ['B','o','n','j','o','u','r']
```

Création de liste - par compréhension

La création de liste par compréhension permet de parcourir un objet itérable en renvoyant une liste.

```
In [1]: L=[i*2 for i in range(0,11)]
```

```
In [2]: L
```

```
Out[2]:
```

```
[0,2,4,6,8,10,12,14,16,18,20]
```

```
In [3]: L=[i for i in 'Bonjour']
```

```
In[4]: L
```

```
Out [4]: ['B','o','n','j','o','u','r']
```

```
In [5]: L=[i+i for i in 'Bonjour']
```

```
In [6]: L
```

```
Out [6]: ['BB','oo','nn','jj','oo','uu','rr']
```

```
In [6]: L=[(i,j) for i in range(3) for j in range(2)]
```

```
In [7]: L
```

```
Out [7]:
```

```
[(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]
```

Création de liste compréhension

La création de liste par compréhension permet de renvoyer une liste dont le contenu peut-être filtré:

```
In [1]: L=[i for i in range(0,11) if i%2==0]
```

```
In [2]: L
```

```
Out[2] [0,2,4,6,8,10]
```

```
In [3]: L=[i for i in 'Bonjour' if i=='B' or i=='j']
```

```
In[4]: L
```

```
Out [4]: ['B','j']
```

Parcours de liste

```
L=[1, 'a', 'coucou',4]
for i in range(len(L)):
    print(L[i])
1
'a'
'coucou'
4
```

```
L=[1, 'a', 'coucou',4]
for i in L:
    print(i)
1
'a'
'coucou'
4
```

Copie d'un objet mutable

- Une affectation $x = y$ n'implique pas la création d'un nouvel objet, mais plutôt la création d'une nouvelle étiquette y (référence) vers l'objet déjà créé.
- Ceci ne pose pas de problème pour les objets non mutables. Par contre, ceci peut poser un problème pour les objets mutables, tels que les listes, où la modification d'une liste implique la modification de l'autre liste.

```
In [1]: L1=[i for i in range(0,6)]
```

```
In [2]: L1
```

```
Out[2] [0,1,2,3,4,5]
```

```
In [3]: L2=L1
```

```
In[4]: L1[0]='A'
```

```
In [5]: L1
```

```
Out[5]: ['A',1,2,3,4,5]
```

```
In [6]: L2
```

```
Out [6]: ['A',1,2,3,4,5]
```

Copie d'un objet mutable

- **Solution:** Créer une copie de l'objet mutable indépendante de l'originale.
- Différentes méthodes :

```
In [7]: L3=L1[:]  
In [8]: L4=[i for i in L1]  
In [9]: L5=list()  
In[10]: for i in L1:  
         L5.append(i)  
In [11]: import copy as c  
In[12]: L6=c.copy(L1)
```

⇒ vérifier les *id* des objets créés.

Copie superficielle/profonde

- Si une liste contient des objets eux-mêmes mutables, la copie que nous venons d'effectuer est insuffisante $\Rightarrow$ copie superficielle

```
In [1]: L1=[[1,2,3],[4,5,6]]  
In [2]: L2=L1[:]  
In [3]: L1.append(56)  
In[4]: L1,L2  
Out [4]: [[1,2,3],[4,5,6],56],[[1,2,3],[4,5,6]]  
In[5]: L1[0].append(66)  
In[6]: L1,L2  
Out[6]:[[1,2,3,66],[4,5,6],56],[[1,2,3,66],[4,5,6]]
```

- **Solution:** Effectuer une copie profonde, en utilisant la fonction `deepcopy` du module `copy`.

```
In [6]: L3=c.deepcopy(L1)
In [7]: id(L1[0])==id(L3[0])
Out [7]: False
In[8]: L1[0].append(80)
In[9]: L1,L3
Out[10]: [[1,2,3,66,80],[4,5,6],56],[[1,2,3,66],[4,5,6],56]]
```

Les chaînes - Rappel

- Les **chaînes de caractères**: suite d'objets de types homogènes (caractères), délimités par de simples ou doubles côtes.
- Les chaînes sont des objets **non mutables** (non modifiables).
- Les chaînes sont des séquences. Par conséquent, toutes les opérations relatives aux séquences sont applicables aux chaînes:
 - L'accès à un élément/le découpage;
 - La fonction *len*;
 - L'appartenance d'un élément (in, not in);
 - Les opérations de concaténation/répétition;
 - Les méthodes count/index;
- PYTHON offre plusieurs méthodes de manipulation de chaînes de caractères.

Les chaînes de caractères

c: une chaîne de caractères.

Méthode	Description
<i>c.split()</i>	Retourne une liste composée des mots composant la chaîne <i>c</i> . Par défaut, le séparateur de mots est le caractère espace.
<i>c.split(':')</i>	Même définition, sauf que cette fois, le séparateur de mots est le caractère ':'.
<i>c.find(cc)</i>	Détermine si <i>cc</i> est une sous-chaîne de <i>c</i> . Retourne l'indice si <i>cc</i> est trouvée, -1 sinon.
<i>c.find(cc,deb,fin)</i>	Même définition, sauf que cette fois, la recherche commence à partir de la position <i>deb</i> et finit à la position <i>fin</i> - 1.
<i>c.replace(str1,str2)</i>	remplace toutes les occurrences de <i>str1</i> dans <i>c</i> par <i>str2</i> .
<i>c.replace(str1,str2,n)</i>	remplace les <i>n</i> premières occurrences de <i>str1</i> dans <i>c</i> par <i>str2</i> .

Les chaînes de caractères

Méthode	Description
<code>c.upper()</code>	Convertit toutes les minuscules de <code>c</code> en majuscules.
<code>c.lower()</code>	Convertit toutes les majuscules de <code>c</code> en minuscules.
<code>c.title()</code>	Retourne une version de <code>c</code> où chaque initiale de chaque mot est une majuscule, comme dans un titre en anglais.
<code>c.isupper()</code>	Retourne <i>True</i> si <code>c</code> contient au moins un caractère alphabétique et que tous les caractères soient en majuscules, <i>False</i> , sinon.
<code>c.islower()</code>	Retourne <i>True</i> si <code>c</code> contient au moins un caractère alphabétique et que tous les caractères soient en minuscules, <i>False</i> , sinon.
<code>c.istitle()</code>	Retourne <i>True</i> si toutes les initiales des mots de <code>c</code> sont en majuscules, <i>False</i> , sinon.

Les chaînes de caractères

Méthode	Description
<code>c.isnumeric()</code>	Retourne <i>True</i> si <i>c</i> ne contient que des caractères numériques, <i>False</i> , sinon.
<code>c.isalpha()</code>	Retourne <i>True</i> si <i>c</i> contient au moins un caractère et que tous les caractères soient alphabétiques, <i>False</i> , sinon.
<code>c.isalnum()</code>	Retourne <i>True</i> si <i>c</i> contient au moins un caractère et que tous les caractères soient alphanumériques, <i>False</i> , sinon.

Remarque: N'oubliez pas que la chaîne est un objet non mutable. Par conséquent, toutes ces méthodes ne modifient pas la chaîne *c* mais renvoient une nouvelle chaîne.

Les tuples - Rappel

- Les **tuples**: suite d'objets de types hétérogènes, délimités par des parenthèses.
- Les tuples sont des objets **non mutables** (non modifiables).
- Les tuples sont des séquences. Par conséquent, toutes les opérations relatives aux séquences sont applicables aux tuples:
 - L'accès à un élément/le découpage;
 - La fonction *len*;
 - L'appartenance d'un élément (in, not in);
 - Les opérations de concaténation/répétition;
 - Les méthodes count/index;

Les tuples

```
In [1]: T=tuple();T # ou bien T=() Tuple vide
```

```
Out [1]: ()
```

```
In [2]: type(T)
```

```
Out [2]: <class 'tuple'>
```

```
In [3]: T=('Bonjour', 14, ['coucou', 15], (67, 'salut'));T # Tuple  
composé de 4 éléments
```

```
Out [3]: ('Bonjour', 14, ['coucou', 15], (67, 'salut'))
```

Les tuples

```
In [4]: T=('Bonjour');type(T) # Attention, ceci n'est pas tuple
```

```
Out [4]: <class 'str'>
```

```
In [5]: T=('Bonjour,');type(T) # Tuple composé d'un seul élément,  
remarquez la virgule à la fin
```

```
Out [5]: <class 'tuple'>
```

Les tuples

- On peut également créer un tuple par énumération, en utilisant la méthode **tuple** avec en paramètre un objet itérable :
`T=tuple(iterable).`
- La création de tuple par compréhension est également possible.

```
In [6]: T=tuple('Bonjour');T
```

```
Out [6]: ('B', 'o', 'n', 'j', 'o', 'u', 'r')
```

```
In [7]: T=tuple(range(11));T
```

```
Out [7]: (0,1,2,3,4,5,6,7,8,9,10)
```

```
In [8]: T=tuple([1,6, ['Bonjour', 65]]);T
```

```
Out [8]: (1,6,['Bonjour',65])
```

```
In [9]: T=tuple((i,i) for i in range(11) if i%2==0);T
```

```
Out [9]: ((0,0), (2,2), (4,4), (6,6), (8,8), (10,10))
```

Conversion-Transtypage

- Les méthodes `list()`, `str()` et `tuple()` sont des méthodes de création respectivement de liste, de chaîne et de tuple.
- Ces méthodes peuvent également être considérées comme des méthodes de transtypage, c'est-à-dire de conversion de types.

```
In [1]: T=(1, 'Bonjour', 80 )
```

```
In [2]: L=list(T);L
```

```
Out [2]: [1, 'Bonjour', 80]
```

In [3]: L=['Bonjour', 14, ['coucou', 15], (67, 'salut')]

```
In [4]: T=tuple(L);T
```

```
Out [4]: ('Bonjour', 14, ['coucou', 15], (67, 'salut'))
```