Initiation au développement

8 — Slices et compréhensions

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

1. Slices
2. Itérables en compréhension

Python pas à pas

Slices

Notion de slice

- Pour une liste, une chaîne ou un tuple `t`, `t[i]` donne accès à l'élément en position `i` ;
- Les **slices** donnent accès à une **portion** du même itérable ;

Syntaxe des slices (liste, chaîne ou tuple)

Le slice `t[debut:fin]` donne la sous-liste (ou la sous-chaîne, ou le sous-tuple) compris(e) entre les indices `debut` et `fin-1`.

Exemple

```
>>> s = 'bonjour'
>>> print(s[2:5])
'njo'
```

- **Attention :**
  1. c'est `fin-1` comme pour les `range` ;
  2. les "mauvais indices" fonctionnent ;
  3. ça ne fonctionne pas pour les dictionnaires ;

Indices négatifs

- Python permet aussi d'utiliser des **indices négatifs** ;
- L'indice `-i` est le même que `len(t) - i` (sauf pour 0) ;

Exemple

0	1	2	3	4	5	6
'b'	'o'	'n'	'j'	'o'	'u'	'r'
-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'bonjour'
>>> print(s[-2])
'u'
>>> print(s[1:-2])
'onjo'
```

- Attention, les mêmes règles de validité d'indices s'appliquent (-10 dans une liste de 5 éléments provoque une erreur) ;

Paramètres par défaut des slices

- Dans un slice, le début et la fin ont des valeurs par défaut :
  - `t[:4]` est la même chose que `t[0:4]` (les 4 premiers éléments) ;
  - `t[4:]` est la même chose que `t[4:len(t)]` (tout l'itérable à partir de la position 4) ;

- La bonne manière d'interpréter les indices négatifs dans les slices est la suivante :
  - `t[:-2]` désigne tous les éléments "**moins** les 2 **derniers**" ;
  - `t[-5:]` désigne les 5 **derniers** éléments ;

- `t[:]` désigne toute la liste et nous servira dans les fonctions sur les listes (voir plus loin) ;

Paramètres par défaut des slices

- Enfin, il existe un troisième paramètre qui permet comme dans les `range` de spécifier la taille des “sauts” :

Exemple

```
>>> s = "bonjour"
>>> s[::]
'bonjour'
>>> s[::1]
'bonjour'
>>> s[::2]
'bnor'
>>> s[::3]
'bjr'
```

Slices et copies

Les slices réalisent des copies d'objets **ou** de références en mémoire !

- Que l'on affecte ou non le slice `iterable[i:j]` à une nouvelle variable :
  - les **valeurs** des éléments non modifiables de `iterable[i:j]` sont dupliquées ;
  - les **références** des éléments modifiables de `iterable[i:j]` sont dupliquées ;
- Le résultat est qualifié de **copie superficielle** (*shallow copy*), par opposition à **copie profonde** (*deep copy*, voir prochain slide) ;
- Il faut donc bien faire attention à la consommation en mémoire et aux modifications involontaires ;

[Visualisation sur PythonTutor]

Copies “superficielles” et “profondes”

- Comme on l’a déjà vu, la manière dont on réalise les copies de listes a une importance en pratique :

Instruction	**Résultat**
`lst2 = lst`	deux variables désignant le même objet

`lst2 = list(lst)`
`lst2 = lst.copy()`
`lst2 = lst[:]`
`lst2 = copy.copy(lst)`
	`lst2` est une copie superficielle de `lst`

`lst2 = copy.deepcopy(lst)`	`lst2` est une copie profonde de `lst`

- Une **copie profonde** est ce qu’on qualifierait de “vraie copie” : toutes les **valeurs** des objets contenus dans `lst` sont dupliquées ;

Modification de listes dans les fonctions

- Comme on l'a vu, `lst[:]` désigne toute la liste ;
- On peut l'utiliser pour modifier `lst` dans un appel de fonction :

Exemple (incorrect)

```
def effacer_liste(ma_liste):
    ma_liste = list()
```

Exemple (correct)

```
def effacer_liste(ma_liste):
    ma_liste[:] = list()
```

Python pas à pas

Itérables en compréhension

Itérables en compréhension

- L'idée est d'avoir un moyen de décrire un itérable comme on décrit un **ensemble en mathématiques** :

$$E = \{2^i \mid i \in \{0, 1, \ldots, 10\}\}$$
$$F = \{x \in \{1, 2, \ldots, 10\} \mid x \text{ impair}\}$$

- La syntaxe générale est :

Syntaxe des compréhensions

pour une liste : `[expression for element in iterable]`
pour un ensemble : `{expression for element in iterable}`
pour un dictionnaire : `{cle: valeur for cle, valeur in iterable}`

Si on veut autre chose, on peut transtyper la compréhension.

N'utilisez pas les parenthèses comme délimiteurs.

Itérables en compréhension

- Utilisons les compréhensions pour construire les deux ensembles donnés ci-dessous :

$$E = \left\{ 2^i \mid i \in \{0, 1, \ldots, 10\} \right\}$$
$$F = \{x \in \{1, 2, \ldots, 10\} \mid x \text{ impair}\}$$

- Pour le premier :

```
E = {2**i for i in range(11)}
```

- Pour le second :

```
F = {i for i in range(1, 11, 2)}
```

Compréhensions conditionnelles (1)

- On peut filtrer les éléments à rajouter à l'aide de conditions (facultatives) :

Syntaxe des compréhensions conditionnelles (1)

Comme avant :

```
gauche expr for element in iterable if conditions droite
```

où les conditions s'emploient comme d'habitude (voir **`if`** et **`while`**).

- On aurait donc pu obtenir l'ensemble des nombres impairs entre 1 et 10 comme suit :

```
F = {i for i in range(11) if i % 2}
```

Compréhensions conditionnelles (2)

On peut aussi utiliser les conditions pour changer la manière d'ajouter des éléments :

Syntaxe des compréhensions conditionnelles (2)

```
gauche expr1 if conditions else expr2 for element in
                        iterable droite
```

Exemple

Convertissons les minuscules en majuscules et les majuscules en minuscules :

```
chaine = "".join([
    lettre.upper() if lettre.islower() else lettre.lower()
    for lettre in chaine
])
```

Place du `if`

Attention à ne pas confondre les deux versions !

La place du `if` importe :

1. `[... for x in lst if BLA]` : ne récupérer que les éléments de `lst` vérifiant `BLA` ;
2. `[a if BLA else b for x in lst]` : récupérer tous les éléments de `lst`, mais générer `a` si `BLA` est vrai et `b` sinon ;

Compréhensions multiples

- On peut utiliser plusieurs **for** :

```
C = [(i, j) for i in range(5) for j in range(4)]
```

- Les **for** sont évalués dans le sens de lecture, comme pour les boucles imbriquées ;
- Ce qui précède équivaut donc à :

Exemple

```
C = []
for i in range(5):
    for j in range(4):
        C.append((i, j))
```

- Ne dépassez pas deux niveaux d'imbrication ;

Attention aux compréhensions multiples

- Les compréhensions multiples nous permettent de déclarer une matrice en une seule ligne ;
- Attention, il ne faut pas oublier qu'on veut déclarer une liste de listes !

Exemple (matrice en une ligne)

```
>>> [[i for i in range(5)] for j in range(4)]  # correct
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]]
>>> [i for i in range(5) for j in range(4)]    # FAUX
[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4,
 4, 4]
```

Récapitulation des compréhensions

- `L = [x for x in iterable if condition]` équivaut à :

```
L = []
for x in iterable:
    if condition:
        L.append(x)
```

- `S = {x for x in iterable if condition}` équivaut à :

```
S = set()
for x in iterable:
    if condition:
        S.add(x)
```

- `D = {cle: valeur for cle, valeur in iterable if condition}` équivaut à :

```
D = dict()
for cle, valeur in iterable:
    if condition:
        D[cle] = valeur
```