Séquences et mutabilité

Cours 1

- v1.1

Lycée polyvalent Franklin Roosevelt, 10 Rue du Président Franklin Roosevelt, 51100 Reims

1 Variables de type séquence itérable

On va ici parler des principaux types de séquences itérables utilisables en Python : les n-uplets (`tuple`), les chaines de caractères (`str`), les parcours (`range`) et les listes (`list`).

Fonctions et méthodes génériques La fonction `len(s)` permet de connaitre la longueur d'une séquence.

Indexation L'indexation se fait avec des crochets. Sucre syntaxique : si l'indice est négatif, l'indice est relatif à la fin de la séquence (`-1` correspond au dernier élément, `-2` à l'avant-dernier, `-3` à l'antépénultième, etc.).

Attention

La numérotation des indices commence à 0.

Concaténation L'opérateur `+` est utilisé pour concaténer des séquences. L'opérateur `*` est utilisé pour répéter la concaténation.

```
>>> L0, L1 = [2, 3], [5, 6]
>>> [0]*4, L1+L0+L1
([0, 0, 0, 0], [5, 6, 2, 3, 5, 6])
```

a) Les n-uplets ou `tuple`

Les n-uplets sont des séquences immuables : elles acceptent toutes les types d'objets à l'intérieur. Elles peuvent être entourées de parenthèses, mais ces dernières ne sont pas obligatoires si on a au moins 2 objets.

```
>>> N = 1, 3.14, 'text'
>>> M = (1, 3.14, 'text')
>>> N==M, N[-1], N[0]
(True, 'text', 1)
```

b) Les chaines de caractères ou `str`

Les chaines de caractères sont également immuables. Elles sont entourées de guillemet simple `'`, double `"` ou triple `'''` ou `"""`. Elles servent à traiter du texte.

On peut noter la fonction `ord(c)` qui prend en entrée une chaine d'un caractère pour renvoyer son code Unicode (un entier) et la fonction `chr(code)` qui fait l'inverse.

Beaucoup de méthodes sont associées aux chaines de caractères. Nous avons résumé les plus importantes dans le tableau ci-dessous.

Nom	Description
`str.replace(old, new)`	Renvoie une copie de la chaine dont toutes les occurrences de la sous-chaine `old` sont remplacées par `new`.
`str.split(sep=None)`	Renvoie une liste des mots de la chaine, en utilisant `sep` comme séparateur de mots (séparation par les espaces si aucun argument n'est donné).

c) Les parcours ou `range`

Le type `range` représente une séquence immuable de nombres et est couramment utilisé pour itérer un certain nombre de fois dans les boucles `for`.

La fonction associée est `range([start=0,] stop[, step=1])`. Cela crée un parcours d'entiers compris entre `start` (inclus) et `stop` (exclu) avec un pas `step`. Si seuls 2 arguments sont fournis, `step` est à 1 par défaut, si un seul argument est fourni `start` est à 0 par défaut.

d) Les listes ou `list`

Les listes sont des séquences génériques muables et elles acceptent toutes les types d'objet à l'intérieur. Elles sont délimitées par des crochets.

Voici une liste des principales opérations pouvant être effectuées sur les listes :

Opération	Description
`s[i] = x`	L'élément numéro `i` de `s` est remplacé par `x`
`s.append(x)`	Ajoute `x` à la fin de la séquence
`s.pop([i=-1])`	Renvoie l'élément numéroté `i` et le supprime de `s` (le dernier par défaut).
`s.reverse()`	Inverse sur place les éléments de `s`. Renvoie `None`.

2 Muabilité en Python

En Python, il existe 2 grands types de variables :

- les immuables (*immutable*) : nombre, n-uplets, chaine de caractères et les parcours ;

- les muables (*mutable*) : les listes mais aussi les dictionnaires et les tableaux numpy que l'on verra plus tard.

L'immuabilité ne signifie pas que les variables restent constantes, mais la seule manière de changer une valeur d'un objet immuable est de l'écraser par une nouvelle valeur. Cela mène à une programmation assez intuitive, mais c'est souvent gênant pour une gestion efficace de la mémoire, surtout si on traite des objets de taille importante.

Les objets muables ont la possibilité d'être modifié, et donc d'évoluer. La gestion de la mémoire est bien plus efficace si on souhaite modifier un élément de l'objet, mais cela peut complexifier un peu la programmation à cause des problèmes de la copie.

En pratique, qu'est-ce que ça change ? Les éléments d'une séquence immuable sont accessibles à la lecture, mais pas à l'écriture, alors que les éléments d'une séquence muable sont accessible à l'écriture.

```
>>> s = "test" # définition d'un str (immuable)
>>> L = [2, 4, 3, 1] # définition d'une list (muable)
>>> t = (2, 4, 3, 1) # définition d'un tuple (immuable)
>>> s[1], L[1], t[1] # Accès en lecture du 2ème élément
('e', 4, 4)
>>> L[1] = 9 # accès en écriture d'un objet muable (list)
>>> L # la liste a bien été changée
[2, 9, 3, 1]
>>> t[1] = 9 # accès en écriture impossible d'un objet immuable (tuple)
Traceback (most recent call last):
File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s[1] = 'o' # accès en écriture impossible d'un objet immuable (str)
Traceback (most recent call last):
File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

3 Copie superficielle et copie profonde

Le problème de la copie est omniprésente pour les objets muables : en effet, en Python, les noms ne sont que des références : `a = b` permet de déclarer que `a` réfère au même objet que `b`. Pour les objets immuables (numériques, n-uplets et chaine de caractères en particulier), cela ne crée jamais de comportement contre-intuitif (l'objet n'est jamais changé : à chaque fois que vous avez l'impression de modifier un objet muable, vous ne faite qu'écrasez une ancienne référence avec une nouvelle).

Cela n'est malheureusement pas le cas pour les objets muables tel que les listes.

```
>>> a = [1,2,3]
>>> b = a
>>> a[1] = 20 ; b[2] = 300
>>> a, b
([1, 20, 300], [1, 20, 300])
```

La solution simple est de faire une copie superficielle :

```
>>> a = [1,2,3]
>>> b = a.copy()
>>> a[1] = 20 ; b[2] = 300
>>> a, b
([1, 20, 3], [1, 2, 300])
```

Cela ne marche malheureusement pas pour des structures imbriquées (liste de listes par exemple). C'est pour cela qu'on parle de copie superficielle : on ne fait que copier les références du niveau 0 sans s'intéresser aux niveaux plus profonds.

```
>>> a = [[1], [2], [3]]
>>> b = a.copy()
>>> a[1][0] = 20 ; b[2][0] = 300
>>> a, b
([[1], [20], [300]], [[1], [20], [300]])
```

Pour cela, on a besoin d'une copie profonde (aussi appelée copie récursive). Une fonction permettant de faire ce type de copie se trouve dans le module `copy` sous le nom de `deepcopy`.

```
>>> from copy import deepcopy
>>> a = [[1], [2], [3]]
>>> b = deepcopy(a)
>>> a[1][0] = 20 ; b[2][0] = 300
>>> a, b
([[1], [20], [3]], [[1], [2], [300]])
```

4 Tranche de listes

Il est possible de prendre des tranches avec `i:j:k`, qui correspond à tous les indices compris entre `i` inclus et `j` exclu avec un pas de `k`. Quelques notations réduites sont bien pratiques : `i:j` correspond à `i:j:1` (avec un pas de 1), `:j` correspond à `0:j` et `i:` correspond à `i:len(s)` avec `len(s)`, la longueur de la séquence `s` considérée et `:` correspond donc à `0:len(s)`.

```
>>> L=[1,3,4,5,9]
>>> L[::2],L[:3],L[:-1]
([1, 4, 9], [1, 3, 4], [1, 3, 4, 5])
```

Sur les listes, une sélection par tranche (*slice*) crée une copie superficielle de la sous-liste. Ce qui signifie que `a[:]` et `a.copy()` sont deux manières de créer une copie d'une liste.

Trivia à lire à la maison

> There should be one – and preferably only one – obvious way to do it. Although that way may not be obvious at first unless you're Dutch.
>
> Extrait de *Zen of Python* par Tim Peters (accessible avec `import this`)

> In context, "Dutch" means a person from the Netherlands, or one imbued with Dutch culture (begging forgiveness for that abuse of the word). I would have said French, except

that every French person I asked "how do you make a shallow copy of a list ?" failed to answer

`alist[:]`

so I guess that's not obvious to them. It must be obvious to the Dutch, though, since it's obvious to Guido van Rossum (Python's creator, who is Dutch), and a persistent rumor maintains that everyone who posts to comp.lang.python is in fact also Dutch. The French people I asked about copying a list weren't Python users, which is even more proof (as if it needed more).

Or, in other words, "obvious" is in part a learned, cultural judgment. There's really nothing universally obvious about any computer language, deluded proponents notwithstanding. Nevertheless, most of Python is obvious to the Dutch. Others sometimes have to work a bit at *learning* the one obvious way in Python, just as they have to work a bit at learning to appreciate tulips, and Woody Woodpecker impersonations.

What's the meaning of Dutch in "The Zen of Python" par Tim Peters