slicing

Meilland

November 12, 2014

1 2 Le slicing et les structures de liste

1.1 2.1 Complément pour la chaîne de caractères.

Une chaîne de caractères est une liste particulière ne contenant que des caractères.
Comme pour une liste, on peut accéder à ses éléments (les caractères) en spécifiant un indice :

```
In []: chaine= "Lycée Pablo Neruda"
       chaine[0]
```

Par contre, il est impossible de modifier une chaîne de caractères ! On dit alors qu'il s'agit d'une liste non modifiable :

```
In []: chaine[1]= "a"
```

Si vous décidiez de lui ajouter des caractères en fin de chaîne à l'aide d'une concaténation du type suivant :

```
In []: chaine=chaine+"!"
       chaine
```

Il existe de nombreuse méthode pour les chaînes de caratères. Ouvrez ipython3 puis tapez

```
In []: x='Pablo Neruda'
```

tapez alors juste ***x. (n'oubliez pas le point) puis appuyez sur la tabulation***, vous devez voir toutes les méthodes.

a	b	c	d
x.capitalize	x.isalnum	x.join	x.rsplit
x.casefold	x.isalpha	x.ljust	x.rstrip
x.center	x.isdecimal	x.lower	x.split
x.count	x.isdigit	x.lstrip	x.splitlines
x.encode	x.isidentifier	x.maketrans	x.startswith
x.endswith	x.islower	x.partition	x.strip
x.expandtabs	x.isnumeric	x.replace	x.swapcase
x.find	x.isprintable	x.rfind	x.title
x.format	x.isspace	x.rindex	x.translate
x.format_map	x.istitle	x.rjust	x.upper
x.index	x.isupper	x.rpartition	x.zfill

Inutile de toutes les connaître. Je vous propose de voir ici les fonctions les plus utiles.

1.1.1 a. Couper et joindre

La fonction ***split()*** permet de ***découper*** la chaîne de caractères qui lui est passée en paramètre suivant un ou des caractère(s) de séparation et renvoie une liste des chaînes découpées. Les caractères de séparation lui sont également passés en paramètre et, si ce n'est pas le cas, ce sera le caractère espace qui sera utilisé :

```
In []: str="Pablo Neruda Saint martin d'hère"
       str.split()
```

```
In []: str.split(' ',2)
```

L'opération inverse s'appelle ***join()***. Elle consiste a rendre une liste de chaînes de caractères pour former une chaîne en concaténant tous les éléments et en les assemblant à l'aide d'un caractère.

Cette méthode prend en paramètre une liste de caractères et s'applique à une chaîne de caractères désignant le ou les caractère(s) de liaison :

```
In []: l=str.split()
       l
```

```
In []: ' - '.join(l)
```

On peut faire aussi la transformation directement

```
In []: ' - '.join("Pablo Neruda Saint martin d'hère".split())
```

1.1.2 b. Majuscule et minuscule

Deux autres méthodes standards peuvent être utiles:

lower() et ***upper()*** permettant respectivement de convertir les caractères d'une chaîne en minuscules ou en majuscules.

Attention, bien que parlant de << conversion >>, ces méthodes ne modifient pas la chaîne de départ mais renvoient une nouvelle chaîne :

```
In []: 'Pablo Neruda'.lower()
```

```
In []: 'Pablo Neruda'.upper()
```

La fonction ***capitalize()*** permet de ne mettre en majuscule que la première lettre d'une chaîne :

```
In []: 'pablo neruda'.capitalize()
```

Si maintenant nous souhaitons que la première lettre de chaque mot soit une majuscule, que pouvons-nous faire?

En utilisant les quelques fonctions vues précédemment, cela est tout à fait réalisable :

```
In []: p='une petite phrase de test'
       l=p.split()
       l
```

```
In []: i=0
       for s in l:
           l[i] = s.capitalize()
           i += 1
       ' '.join(l)
```

On peut même le faire beaucoup plus << simplement >>, en utilisant la compréhension de listes

```
In []: ' '.join([s.capitalize() for s in p.split()])
```

On reverra les compréhensions de listes un peu plus tard

1.1.3 c. rechercher un caractère, une position etc... dans une chaîne.

- La fonction ***len()*** permet de compter le nombre de caractères.

len('Pablo Neruda')

- La méthode ***count()*** permet de compter le nombre d'occurrences d'une sous chaîne dans une chaîne de caractères.
  Le premier paramètre est la chaîne dans laquelle effectuer la recherche et le second paramètre est la sous chaîne :

```
In []: 'Pablo Neruda'.count('a')
```

- La fonction ***find()*** permet de trouver l'indice de la première occurrence d'une sous chaîne. Les paramètres sont les mêmes que pour la fonction ***count()***
  En cas d'échec, ***find()*** renvoie la valeur -1 (0 correspond à l'indice du premier caractère): On utilisera ***rfind()*** pour la dernière ocurrence.

```
In []: 'Pablo Neruda'.find('a')

In []: 'Pablo Neruda'.find('aa')

In []: 'Pablo Neruda'.rfind('a')
```

- ***index()*** est identique à ***find()*** mais retourne une erreur en cas d'échec

```
In []: 'Pablo Neruda'.index('a')

In []: 'Pablo Neruda'.index('aa')
```

Avec les méthode qui renvoie une ***ValueError*** on utlilise généralement la "condition" try except.

try: Quelquechose except: S'il y a une erreur alors autre chose

```
In []: try: print('Pablo Neruda'.index('aa'))
       except:  print("Il n'y a pas de aa")
```

- La fonction ***replace()*** permet, comme son nom l'indique, de remplacer une sous chaîne par une autre à l'intérieur d'une chaîne de caractères.
  Les paramètres sont, dans l'ordre : la chaîne de caractères à modifier, la sous chaîne à remplacer, la sous chaîne de remplacement,et, éventuellement, le nombre maximum d'occurrences à remplacer (si non spécifié, toutes les occurrences seront remplacées).

```
In []: 'Pablo Neruda'.replace('a','1')
```

Pour des recerches un peu plus complète on utilisera le module ***re*** (regex)

1.1.4 d. Convertion

str.encode(encoding="utf-8", errors="strict") Retourne une version codée de la chaîne comme un objet ***bytes***. L'encodage par défaut est "utf-8".

```
In []: 'Pablo Neruda'.encode('utf8')
```

*## À retenir:
Il faut savoir utiliser ***split()*** , ***join()*** , ***len()*** , ***count()*** , ***find()*** , ***index()*** , ***rfind()***, ***replace()***.

1.1.5 b. Exercice string

Exercice 1:
Écrire un programme qui dans une phrase compte:

- a) le nombre de voyelle.
  Entrée: 'Un matin'
  Sortie: 3
- b) Le nombre de caratères (on utilisera la méthode **isalnum()** qui retourne True si le caracère est une lettre ou un chiffres)
  Entrée: "l'avis n°1"
  Sortie: 7

Exercice 2:
Écrire un programme qui teste si la chaîne str* est une adresse mail. On va simplifier sur le fait que *str* doit se terminer par @****.**
- Un seul caractère @ - Un seul caractère .(point) après @.

Exercice 3:
Écrire un programme qui détermine si une chaîne de caractères est un palindrome (un mot qui pour lequel la signification est la même dans les deux sens de lecture, par exemple *ressasser*.

Exercice 4:
Demandez la saisie d'un message puis afficher ce dernier de façon indentée.

Exemple: si str='toto'
t
to
tot
toto

1.2 2.2 Les spécificités des tuples

Les tuples sont également des listes non modifiables (les chaînes de caractères sont donc en fait des tuples particuliers) :

```
In []: t=(1,2,3)
       t[0]

In []: t[0]=0
```

Attention de ne pas utiliser le mot-clé tuple comme nom de variable: ce dernier permet de créer un tuple de manière explicite. Il vous faudra utiliser en paramètre. . . une liste. Il permet donc d'effectuer une conversion :

```
In []: l=[1,2,3]
       t=tuple(l)
       t

In []: l

In []: t=tuple('1234')
       t
```

Tout comme avec les chaînes, il sera possible de concaténer des tuples et avoir la sensation d'avoir modifié un tuple alors que nous aurons simplement réaffecté une variable :

```
In []: t=(1,2,3)
       t=t+(4,5,6)
       t
```

Pour rappel, un tuple ne contenant qu'un seul élément est noté entre parenthèses, mais avec une virgule précédant la dernière parenthèse:
Quel peut être l'intérêt d'utiliser ce type plutôt qu'une liste ?

Tout d'abord, les données ne peuvent pas être modifiées par erreur mais surtout, leur implémentation en machine fait qu'elles occupent moins d'espace mémoire et que leur traitement par l'interpréteur est plus rapide que pour des valeurs identiques mais stockées sous forme de listes.

De plus, de par leur aspect non modifiable, les valeurs contenues dans un tuple peuvent être utilisées en tant que clé pour accéder à une valeur dans un dictionnaire (alors que c'est beaucoup plus dangereux avec les valeurs contenues dans une liste) :

```
In []: l=['cle1','cle2','cle3']
       t=('cle1','cle2','cle3')
       d={'cle1':1,'cle2':2,'cle3':3}
       d[l[0]]

In []: l[0]= 'cle_modifiee'
       d[l[0]]
```

Les tuples dispose de deux méthodes ***count()*** et ***t.index***. Leur utilisation est semblable aux chaînes de caractère.

1.2.1 À retenir

Les tuples s'écrivent avec des parenthèses et ne sont pas modifiables.

1.3 2.3 Les spécificités des listes

Nous avons vu que les listes étaient des éléments modifiables pouvant contenir différents types de données. Il est bien sûr possible de modifier les éléments d'une liste:

```
In []: l=[1,2,3,4]
       l

In []: l[0]=0
       l
```

Plusieurs **méthodes** peuvent être employées pour ajouter des éléments a une liste existante (sans réaffectation).
Voici la liste des méthodes pour les listes.

a	b	c	d	e	f
l.append	l.copy	l.extend	l.insert	l.remove	l.sort
l.clear	l.count	l.index	l.pop	l.reverse	

1.3.1 a. Ajouter des éléments

La méthode ***append(x)*** permet de rajouter l'élément x à la fin de la liste.

```
In []: l=[1,2,3,4]
       l.append(5)
       l
```

La méthode ***insert(n,x)*** permet d'insérer l'élément x dans la liste à l'indice n.

```
In []: l=[1,2,3,4]
       l.insert(1,8)
       l
```

La méthode ***extend(l2)*** permet de réaliser la concaténation de deux listes sans réaffectation. Cette opération est réalisable avec réaffectation en utilisant l'opérateur + :

```
In []: l1=[1,2,3]
       l2=[4,5,6]
       l1.extend(l2)
       l1
```

```
In []: l1=[1,2,3]
       l1=l1+l2  #On peur remplacer par l1+=l2
       l1
```

1.3.2 b. Supprimer des éléments

- La méthode ***clear()*** (depuis python3.3) permet d'effacer la liste

```
In []: l.clear()
       l
```

Si la version de python est inférieur à 3.2 alors on utilisera

```
In []: l=[]
```

- ***remove(x)*** supprime la première occurrence de x.

```
In []: l=[1,2,3,1]
       l.remove(1)
       l
```

```
In []: l.remove(1)
       l
```

```
In []: l.remove(1)
```

- La méthode ***l.pop([i])*** Retourne l'élément de la liste l d'indice i et supprime cet élément de la liste.

```
In []: l=[1,2,3,1]
       a=l.pop(2)
       print(a)
       print(l)
```

- La fonction ***del*** supprime aussi un élément précis en fonction de son indice :

```
In []: l=[1,2,3,1]
       del l[2]
       l
```

1.3.3 c. Autres méthodes

- Pour savoir si un élément appartient bien à une liste on utilise le mot-clé ***in***.
  Si l'élément testé est dans la liste, la valeur retournée sera ***True*** et sinon ce sera ***False***:

```
In []: distrib = ['debian','ubuntu','fedora']
       'debian' in distrib
```

```
In []: 'mandriva' in distrib
```

- Il est également possible d'obtenir l'indice de la première occurrence d'un élément par la méthode ***index()***:

```
In []: distrib.index('ubuntu')
```

- Pour connaître la taille d'une liste (nombre d'éléments qu'elle contient), on pourra utiliser comme pour les chaîne de caractère la fonction ***len()***:

Mais attention : cette fonction ne compte que les éléments de << *premier niveau* >> ! Si vous avez une structure de liste complexe contenant des sous-listes, chaque sous-liste ne comptera que comme un seul élément:

```
In []: l=[1,2,['a','b',['000','001','010']]]
       len(l)
```

```
In []: l[2]
```

```
In []: l[2][2]
```

```
In []: l[2][2][1]
```

1.3.4 d. Les compréhensions de listes

Nous souhaitons obtenir la liste des carrés pour i allant de 0 à 20.
Une première méthode peut être celle-ci

```
In []: carre=[]                 # On créer une liste vide
       for i in range(21):      # pour i allant de 0 à 20
           carre.append(i**2)   # On ajoute le carré
       print(carre)             # On affiche la liste
```

Python implémente un mécanisme appelé << ***compréhension de listes*** >>, permettant d'utiliser une fonction qui sera appliquée sur chacun des éléments d'une liste.

```
In []: l=[i**2 for i in range(21)]
       print(l)
```

À l'aide de l'instruction *for i in range(21)*, on récupère les entiers i de 0 à 20, et on place les carré (i**2) dans la liste.
Ce mécanisme peut produire des résultats plus complexes et on peut aussi appliquer une condition aux éléments *i* à utiliser.

```
In []: carre=[i**2 for i in range(21) if i%2==0 and i>2]  # si i est pair et strictement supérieur à 2
       carre
```

```
In []: import random
       l=[random.randint(1,6) for i in range(20)]
       l # liste de 20 nombres aléatoires entre 1 et 6
```

Il y a donc différentes façon de construire une liste
ou par compréhension
les compréhensions de listes peuvent également être utilisés avec les dictionnaires.

1.3.5 e. À retenir

Il faut savoir ajouter, rechercher , enlever un élément dans une liste Les méthodes à connaitre sont *append(), insert(), remove(), count(), index(), pop()* ou la fonction *del*

1.3.6 f. Exercices

Exercice 1: Soit (u_n) la suite définie par $\begin{cases} u_0 = 1,2 \\ u_{n+1} = -2u_n + 1 \end{cases}$

Demander à l'utilisateur de saisir un entier n puis afficher dans une liste tous les termes de la suite (u_n). Afficher enfin la somme avec la fonction *sum()*.

Exercice 2: On met un grain de riz sur la 1° case d'un échiquier, deux sur la deuxième, 4 sur la troisième, 8 sur la quatrième etc.....
Combien y a t il de grain de grain sur l'échiquier? On fera apparaitre dans une liste le nombre de nombre de grain de chaque case.
idem pour un damier

Exercice 3:
Écrire un programme qui inverse les mots (séparés par un espace) d'une phrase.

Entrée: 'Un matin nous partons, le cerveau plein de flamme'
sortie: 'flamme de plein cerveau le partons, nous matin un'

Exercice 4:
Soit (u_n) et (v_n) les suites définies par $u_0 = 2, \quad \forall n \in N \; v_{n+1} = \dfrac{2}{u_n}$ et $u_{n+1} = \dfrac{u_n + v_n}{2}$.

Déterminer dans les listes U et V les termes des suites (u_n) et (v_n)

Exercice 5: *Une marche aléatoire*

Un point M peut se déplacer sur un quadrillage d'un pas dans l'une des quatre directions. Les quatre déplacements possibles se font au hasard (ils sont donc équiprobables). La position de M est repérée par ses coordonnées $(x; y)$ entières dans le repère indiqué sur la figure. Au départ M est en $(0, 0)$.

On continue les déplacements jusqu'à ce que M sorte du cercle de centre O et de rayon 10.

On appelle N le nombre de pas duquel M sort du cercle pour la première fois *(N est donc une variable aléatoire)*.

Le but de l'exercice est d'essayer, en utilisant des simulations, d'avoir une idée des probabilités des évènements suivants:

- A: << M sort du cercle en moins de 20 pas >> (soit $N \leq 20$),
- B: << M sort du cercle après un nombre de pas compris entre 21 et 30 >> (soit $20 < N \leq 30$),
- C: << M sort du cercle après un nombre de pas compris entre 31 et 40 >> (soit $30 < N \leq 40$),
- D: << M sort du cercle en plus de 40 pas >> (soit $40 < N$),

Simuler un programme pour faire une simulation, puis pour faire 2000 simulations.
On Mettra les nombres N dans une liste L.

1.4 2.4 Les spécificités des dictionnaires

Comme nous l'avons vu, les dictionnaires sont des listes d'éléments indicés par des clés.
Comme pour les listes, la commande ***del*** permet de supprimer un élément et son mot clé , << ***in*** >> permet de vérifier l'existence d'une clé :

```
In []: d={'cle1':1, 'cle2':2, 'cle3':3}
       del d['cle2']
       d

In []: 'cle1' in d

In []: 'cle2' in d
```

Vous aurez pu remarquer lors de l'affichage du dictionnaire ***d***, que les couples *clé/valeur* n'étaient pas affichés dans l'ordre de création.
C'est tout à fait normal car les dictionnaires ne sont pas ordonnés !
Plusieurs méthodes permettent de récupérer des listes de clés, de valeurs et de couples *clé/valeur* :

```
In []: courses ={'pommes':3, 'poires':5 , 'kiwis':7}
       courses.keys()

In []: courses.values()

In []: courses.items()
```

Ces méthodes renvoient **des objets itérables**: ce ne sont pas des listes et ils ne consomment donc pas autant de place mémoire que pour stocker une liste, on ne stocke qu'un *pointeur* vers l'élément courant. Avec ces structures vous pourrez toujours utiliser les boucles ***for*** classiques, par contre vous n'aurez plus un accès direct aux valeurs en spécifiant leur indice.

Dans un dictionnaire, pour obtenir la valeur correspondant à une clé on peut bien entendu utiliser la notation classique ***dictionnaire[clé]***, mais on peut aussi utiliser la méthode ***get()*** qui renvoie la valeur associée a la clé passée en premier paramètre ou, si elle n'existe pas, la valeur passée en second paramétré :

```
In []: courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
       courses.get( 'pommes' , 'stock epuise' )

In []: courses.get('salades' , 'stock epuise')
```

On peut fusionner deux dictionnaires avec la méthode ***update()*** : Notez que si une clé existe déjà, la valeur stockée sera écrasée:

```
In []: courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
       courses2={ 'kiwis':3, 'salades':2}
       courses.update(courses2)
       courses
```

1.5 2.5 Le slicing

Le ***slicing*** est une méthode applicable à tous les objets de type liste ordonnée (donc pas aux dictionnaires). ll s'agit d'un << découpage en tranches >> des éléments d'une liste de manière à récupérer des objets respectant cette découpe.

Pour cela, nous devrons spécifier l'indice de l'élément de départ, l'indice de l'élément d'arrivée (qui ne sera pas compris dans la plage) et le pas de déplacement. Pour une variable v donnée, l'écriture se fera en utilisant la notation entre crochets et en séparant chacun des paramètres par le caractère deux-points: ***v[début:fin:pas]***. Cette écriture peut se traduire par : << les caractères de la variable ***v*** depuis l'indice ***début*** jusqu'à l'indice ***fin*** non compris avec un déplacement de ***pas*** caractère(s) >>.

Pour bien comprendre le fonctionnement du slicing, nous commencerons par l'appliquer aux chaînes de caractères avant de voir les listes et les tuples.

1.5.1 a) Le slicing sur une chaîne de caratère

Voici un premier exemple simple

```
In []: c='Pablo Neruda'
       c[:5]

In []: c[6:12]
```

Vous avez remarque que dans ce code aucune indication de pas n'a été donnée : c'est la valeur par défaut qui est alors utilisée, c'est-a-dire ***1***. De même, si la valeur de début est omise, la valeur par défaut utilisée sera ***0*** et si la valeur de fin est omise, la valeur par défaut utilisée sera la *taille de la chaîne+1*.

```
In []: c[:5] # équivaut à c[0:5], équivaut à c[0:5:1]

In []: c[6:] # équivaut à c[6:12], équivaut à c[6:12:1]
```

Du fait de ces valeurs par défaut, que pensez vous que l'on sélectionne en tapant: c[:] ou encore c[::]? La chaîne entière, bien sûr :

```
In []: c[:]
```

Il devient alors très simple d'inverser une chaîne en utilisant le pas :

```
In []: c[::-1]
```

Une première solution pour effectuer une copie peut être d'utiliser le slicing.
En effet, l'opération [:] renvoie une nouvelle liste, ce qui résout le problème des pointeurs vers la même zone mémoire, comme le montrent l'exemple suivant :

```
In []: listea =[1, 2, 3]
       listeb =listea[:]
       listea

In []: listeb

In []: listeb[0]=0
       listeb

In []: listea
```

Cette copie peut paraître satisfaisante... et elle l'est, mais à condition de ne manipuler que des listes de premier niveau ne comportant aucune sous-liste :

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

In []: listeb

In []: listeb[3][0]=0
       listeb

In []: listea
```

En effet, le slicing n'effectue pas de copie récursive: en cas de sous-liste, on retombe dans la problématique des pointeurs mémoire. La solution est alors d'utiliser un module spécifique, le module ***copy***, qui permet d'effectuer une copie récursive. Bien que n'ayant pas encore approfondi la manipulation des modules, voici comment réaliser une copie de liste par cette méthode:

```
In []: from copy   import *
       listea=[1,2,3,[4,5,6]]
       listeb=deepcopy(listea)
       listea

In []: listeb

In []: listeb[3][0]=0
       listeb

In []: listea
```

Le problème est exactement le même avec la copie de dictionnaires. Il faudra utiliser ici la méthode ***copy()***:

```
In []: dicoa={ 'cle1':1, 'cle2':2, 'cle3':3}
       dicob=dicoa.copy()
       dicoa

In []: dicob

In []: dicob['cle1']=0

In []: dicob

In []: dicoa
```

1.6 Conclusion

Les structures de liste en Python sont particulièrement fournies et il existe pour chacune d'elles de nombreuses méthodes permettant de manipuler leurs éléments.
Le slicing représente un raccourci efficace pour récupérer ou modifier des éléments d'une liste: il masque la complexité des opérations sous une syntaxe claire et lisible... pour peu que l'on ait pris le temps de se familiariser avec elle.