Les Données sur Python

Auteur: Mirko Pirona, Gymnase du Bugnon, 3MOCINFO

Date: Mai 2019

Python Logo

Listes

Une liste en python est une séquence de valeurs. Les valeurs dans la liste sont appelées **éléments**. Les valeurs dans une liste peuvent être de n'importe quel type (chaîne de caractère, flottant, entier et même une autre liste). Il existe plusieurs façons de créer une nouvelle liste dont la plus commune consiste à entourer les éléments par des crochets ([ ]):

In [1]:
```
vb = [5, 10, 15, 20, 25]

vb1 = ['Macdonald', 'Burger King', 'Holly Cow']

vb2 = [10, 'Gymnase du Bugnon', [10, 20, 30]]

print(vb,vb1,vb2)
```
```
[5, 10, 15, 20, 25] ['Macdonald', 'Burger King', 'Holly Cow'] [10, 'Gymnase du Bugnon', [10, 20, 30]]
```

Comme on peut le constater dans la dernière ligne de l'exemple au-dessus, une liste dans une liste est dite **imbriquée**. Il est possible de créer une liste qui ne contient aucun élément, cette liste est dite vide. Il est également possible d'affecter des valeurs aux variables de la liste :

In [2]:
```
Devises = ['Dollar', 'Euro', 'Franc']

Nombres = [5, 10, 15]

Liste_Vide = []

print(Devises, Nombres, Liste_Vide)
```
```
['Dollar', 'Euro', 'Franc'] [5, 10, 15] []
```

Les listes sont modifiables mais avant de pouvoir les modifier il faut accéder aux éléments de la liste. Pour ce faire il suffit d'utiliser l'opérateur [ ] d'indexation. Il ne faut pas oublier que l'expression placée entre crochets spécifie l'indice (le premier indice est 0 et non pas 1). Lorsque l'opérateur d'indexation apparaît du côté gauche de l'affectation, il identifie l'élément de la liste auquel la valeur sera effectée.

In [3]:
```
Devises = ['Dollar', 'Euro', 'Franc']

Devises[1]
```
Out[3]:
```
'Euro'
```

In [4]:
```
Nombres = [5, 10, 15]

Nombres[1] = 100

print(Nombres)
```
```
[5, 100, 15]
```

Le **deuxième élément** de ***Nombres***, qui avait la valeur 10, a maintenant la valeur 100.

Il est possible de parcourir une liste en utilisant une boucle ***for***. Mais si nous souhaitons écrire ou mettre à jour une liste il faut utiliser des indices. Une façon usuelle d'effectuer cela est de combiner les fonctions internes ***len*** et ***range***

In [5]:
```
Fruits = ['pommes', 'poire', 'banane', 'pêche', 'cerise']
Chiffres = [1, 5, 10, 100, 1000]

for Fruit in Fruits:
    print(Fruit)

for i in range (len(Chiffres)):
    Chiffres[i] = Chiffres[i] * 2
    print(Chiffres)
```
```
pommes
poire
banane
pêche
cerise
[2, 5, 10, 100, 1000]
[2, 10, 10, 100, 1000]
[2, 10, 20, 100, 1000]
[2, 10, 20, 200, 1000]
[2, 10, 20, 200, 2000]
```

La deuxième boucle de l'exemple au-dessus parcourt la liste et met à jour chaque élément. ***Len*** renvoie le nombre d'éléments dans la liste. ***Range*** renvoie une liste d'indices allant de 0 à n-1, où n est la longueur de la liste. A chaque passage dans la boucle, i prend la valeur de l'indice de l'élément suivant. L'instruction d'affectation dans le corps utilise i pour lire l'ancienne valeur de l'élément et pour lui attribuer la nouvelle valeur.

Il est possible d'effectuer des opérations sur les listes:

1. L'opérateur + concatène des listes
2. L'opérateur * répète une liste un nombre donné de fois

In [6]:
```
a = ['pomme', 'poire', 'cerise']
b = ['gateau', 'cake']
c = a + b

print(c)

d = ['Ha!']
e = d * 5

print(e)
```
```
['pomme', 'poire', 'cerise', 'gateau', 'cake']
['Ha!', 'Ha!', 'Ha!', 'Ha!', 'Ha!']
```

L'opérateur de tranche [M:N] peut être utilisé sur les listes. Si on omet le premier indice en utilisant l'opérateur alors la liste débutera avec le premier élément de la liste. La même chose se produit si on omet de deuxième indice mais cette fois la liste se terminera avec le dernier élément:

In [7]:
```
t = ['I', 'II', 'III', 'IV', 'V', 'VI']
s = [1, 2, 3, 4, 5, 6, 7, 8]
x = ['a', 'b', 'c', 'd', 'e', 'f']


print(t[1:3])


print(s[:4])


print(x[5:])
```
```
['II', 'III']
[1, 2, 3, 4]
['f']
```

Il existe des méthodes qui agissent sur les listes:

1. ***append*** qui ajoute un nouvel élément à la fin d'une liste
2. ***extend*** qui prend comme argument une liste et ajoute tous les élément de celle-ci
3. ***sort*** qui arrange les élément d'une liste en ordre croissant

In [8]:
```
w = ['a', 'b', 'c', 'd']
w.append('e')
print(w)

w1 = [1, 2, 3, 4]
w2 = [5, 6, 7]
w1.extend(w2)
print(w1)

w3 = [4, 7, 1, 2, 9, 19]
w3.sort()
print(w3)
```
```
['a', 'b', 'c', 'd', 'e']
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 4, 7, 9, 19]
```

Dans Pyhon il est possible de créer des fonctions qui :

1. mappent une liste (appliquent une fonction sur chacun des éléments d'une séquence)
2. filtrent une liste (selectionent certains éléments et filtrent les autres)
3. réduisent une liste (combinent une séquence d'éléments en une seule valeur)

In [9]:
```
Q = [1, 2, 3, 4, 5]
test = sum(Q)
print(test)


Q1 = ['a', 'b', 'c', 'd', 'e']
def mettre_tout_en_majuscule(t):
    res = []
    for s in t:
        res.append(s.capitalize ())
    return res
test1 = mettre_tout_en_majuscule(Q1)
print(test1)

Q2 = ['a', 'B', 'd', 'E', 'F']
def seulement_majuscules(v):
    vor = []
    for s in v:
        if s.isupper():
            print('true')
        else:
            print ('false')
test2 = seulement_majuscules(Q2)
```
```
15
['A', 'B', 'C', 'D', 'E']
false
true
false
true
true
```

Le premier exemple au-dessus montre une fonction qui additionne tous les éléments d'une liste. Le deuxième exemple montre une fonction qui transforme toutes les lettres minuscules d'une liste en majuscules. Le troisième exemple montre une fonction qui renvoie **true** quand l'élément de la liste est en majuscule, dans tou autre cas **false** sera renvoyé.

Il existe plusieurs façon de **supprimer** des éléments d'une liste:

1. La première méthode est ***pop*** (cette méthode modifie la liste et renvoie l'élément qui a été supprimé)
2. La deuxième méthode est ***del*** (cette méthode modifie la liste sans renvoyer l'élément qui a été supprimé)
3. La troisième méthode est ***remove*** (cette méthode permet de modifier une liste sans connaître l'indice de l'élément)

In [10]:
```
g = ['a', 'b', 'c', 'd', 'e']
h = g.pop(1)
print (g)
print (h)

g1 = ['x', 'y', 'z']
del g1[1]
print (g1)


g2 = ['pomme', 'poire', 'cerise']
g2.remove('pomme')
print (g2)
```
```
['a', 'c', 'd', 'e']
b
['x', 'z']
['poire', 'cerise']
```

Une chaîne est une séquence de caractères et une liste est une séquence de valeurs, mais une liste de caractères n'est pas la même chose qu'une chaîne de caractères. Pour convertir une chaîne en une liste de caractères il existe deux fonctions:

1. La première fonction est ***list*** (cette fonction décompose une chaîne de caractère en lettres individuelles)
2. La deuxième fonction est ***split*** (cette fonction décompose une chaîne en mots)

Pour la deuxième fonction il est possible d'utiliser un délimiteur qui spécifie quels caractères utiliser comme limites de mots

In [11]:
```
ab = 'Bugnon'
cd = list (ab)
print (cd)

ab1 = "L'informatique c'est top !"
cd1 = ab1.split()
print (cd1)

ab2 = 'Bug-Bug-Bug'
delimiteur = '-'
cd2 = ab2.split(delimiteur)
print (cd2)
```
```
['B', 'u', 'g', 'n', 'o', 'n']
["L'informatique", "c'est", 'top', '!']
['Bug', 'Bug', 'Bug']
```

Join est l'inverse de ***split***. Cette fonction prend une liste de chaîne de caractère et concatène les éléments. Pour l'utiliser il faut également invoquer un délimiteur.

In [12]:
```
ab3 = ['Le', 'Bugnon', 'est', 'beau', '!']
delimiteur1 = ' '
cd3 = delimiteur1.join(ab3)
print (cd3)
```
```
Le Bugnon est beau !
```

Il est important de différencier la notion **d'objet** et de **valeur**. Un objet dans python pourrait être imagé par une pomme. Tandis que la valeur pourrait être son prix. Lors d'une attribution de variable nous séléctionons un objet et donc la pomme dans notre exemple. Si nous attribuons une autre variable à une pomme il pourrait s'agir de la même pomme (même prix) ou alors d'une autre. Dans python deux objets identiques peuvent avoir la même valeur comme dans le cas d'une chaîne de caractère ou bien une valeur différente comme avec les listes. Pour savoir si deux objets ont la même valeur il faut utiliser l'opérateur ***is***.

In [13]:
```
ef = 'chaise'
fe = 'chaise'
gh = ef is fe
print (gh)

ef1 = ['a', 'b', 'c']
fe1 = ['a', 'b', 'c']
gh1 = ef1 is fe1
print (gh1)
```
```
True
False
```

Dans le deuxième exemple au-dessus les deux listes sont dites **équivalentes**, parce qu'elles ont les mêmes éléments, mais pas **identiques**, car il ne s'agit pas d'un même objet.

L'aliasing est une circonstance où deux ou plusieurs variables font références au même objet. Il est normalement conseillé de l'éviter car cela peut être source d'erreur.

In [14]:
```
sd = ds = [1, 2, 3, 4, 5, 6]
ds[1] = 37
print (sd)
```
```
[1, 37, 3, 4, 5, 6]
```

Dictionnaires

Un dictionnaire ressemble à une liste, mais il est plus général. Dans une liste, les indices doivent être des nombres entiers; dans un dictionnaire, ils peuvent être de (presque) n'importe quel type.

Un dictionnaire contient une collection d'indices, qui sont appelés **clés**, et une collection de valeurs. Chaque clé est associée à une valeur unique. L'association entre une clé et une valeur est appelée un **item**.

La fonction ***dict*** crée un nouveau dictionnaire sans aucun élément.

In [15]:
```
it_vers_fr = dict ()
print (it_vers_fr)
```
```
{}
```

Pour aujouter des éléments au dictionnaire on doit utiliser des crochets :

In [16]:
```
it_vers_fr2 = dict ()
it_vers_fr2 ['albero'] = 'arbre'
print (it_vers_fr2)
```
```
{'albero': 'arbre'}
```

Cette exemple au dessus crée un élément qui fait correspondre la clé 'albero' à la valeur 'arbre'. Cela s'appelle un **item**.

Le nombre d'éléments dans un dictionnaire est illimité. La fonction ***len*** peut être utilisée sur les dictionnaires ; elle renvoie le nombre de paires clé-valeur. L'opérateur ***in*** fonctionne également sur les dictionnaire ; il nous indique si quelque chose apparaît comme une clé dans le dictionnaire. Enfin, pour savoir si quelque chose apparaît comme une valeur dans un dictionnaire, il faut utiliser la méthode ***values*** combinée avec l'opérateur ***in*** :

In [17]:
```
it_vers_fr3 = {'albero' : 'arbre', 'salame' : 'salami', 'pasta' : 'pâte'}

fg = len(it_vers_fr3)
print(fg)

fg1 = 'albero' in it_vers_fr3
print(fg1)

valeurs = it_vers_fr3.values()
fg2 = 'arbre' in valeurs
print(fg2)
```
```
3
True
True
```

Il est possible d'utiliser les dictionnaires pour créer **un compteur** de la fréquence d'apparition des lettres dans un mot :

In [18]:
```
def histogramme (er):
    re = dict()
    for c in er:
        if c not in re:
            re[c] = 1
        else:
            re[c] += 1
    return re

test20 = histogramme ('enveloppe')
print (test20)
```
```
{'e': 3, 'n': 1, 'v': 1, 'l': 1, 'o': 1, 'p': 2}
```

Tuples

Un tuple est une séquence de valeurs. Les valeurs peuvent être de tout type et elles sont indexées par des entiers ; à cet égard, les tuples ressemblent donc beaucoup aux listes. La différence est que les tuples sont **immuables**. Syntaxiquement, un tuple est une liste de valeurs séparées pas des virgules.

In [19]:
```
hu = 'a', 'b', 'c', 'd', 'e', 'f'
type (hu)
```
Out[19]:
```
tuple
```

Un autre façon de créer un tuple est d'utiliser la fonction interne ***tuple*** :

In [20]:
```
uh = tuple ()
we = type (uh)
print (we)

uh1 = tuple ('mirko')
ew = type (uh1)
print (ew)
```
```
<class 'tuple'>
<class 'tuple'>
```

La majorité des opérateurs de liste tel que l'opérateur ***[ ]*** d'indexation ou encore de celui de tranche fonctionne sur les tuples. Cependant, il n'est pas possible de modifier un des éléments du tuple car les tuples sont **immuable**.

In [21]:
```
jk = ('a', 'b', 'c', 'd', 'e')
kj = jk[0]
print (kj)

kj1 = jk[1:3]
print (kj1)

kj2 = jk[0] = 'qq'
print(kj2)
```
```
a
('b', 'c')

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-e4a6b2e95784> in <module>
      6 print (kj1)
      7 
----> 8 kj2 = jk[0] = 'qq'
      9 print(kj2)

TypeError: 'tuple' object does not support item assignment
```