Listes

S. Vialette

IGM-LabInfo, Université Paris-Est

21 janvier 2008

Un objet composite

- Phyton permet de représenter des listes *composites*.
- Une liste eut être écrite comme une liste de *valeurs* entre crochets et séparées par des virgules.
- La liste *vide* est le liste qui ne contient aucun élément.

```
>>> []
[]
>>> myList = []
>>> myList = [1, 2, "abc", 4+5j]
>>> myList
[1, 2, 'abc', (4+5j)]
>>>
```

Un objet composite

- La fonction `ln` permet de déterminer la taille d'une liste.

```
>>> myList = []
>>> len(myList)
0
>>>
>>> myList = [1, 2, "abc"]
>>> len(myList)
3
>>> len([1, 2, "abc"])
3
>>>
```

Accéder aux éléments d'une liste

- Le premier élément est à la position 0.
- Le dernier élément est à la position `len(`*myList*`)-1`.

```
>>>
>>> myList
[1, 2, 'abc', (4+5j)]
>>> myList[0]
1
>>> myList[len(myList)-1]
(4+5j)
>>>
```

Découpage en tranches

```
>>> myList = [1, 'a', 2, 'b', 3, 'c']
>>> myList[:]
[1, 'a', 2, 'b', 3, 'c']
>>> myList[2:]
[2, 'b', 3, 'c']
>>> myList[:4]
[1, 'a', 2, 'b']
>>> myList[2:4]
[2, 'b']
>>> myList[2:4] = '?'
>>> myList
[1, 'a', '?', 3, 'c']
>>> myList[1:] = myList[3:]
>>> myList
[1, 3, 'c']
>>> myList[:1] + myList[:1]
[1, 1]
>>> myList[1:] + myList[:1]
[3, 'c', 1]
>>>
```

Indices négatifs

```
>>> myList = [1, 'a', 2, 'b', 3, 'c']
>>> myList[-1]
'c'
>>> myList[-0]
1
>>> myList[2:-2]
[2, 'b']
>>> myList[3:-3]
[]
>>> myList[2:-2] + ['x', 'y']
[2, 'b', 'x', 'y']
>>>
>>> myList
[1, 'a', 2, 'b', 3, 'c']
>>> myList = myList[:-1] + myList
>>> myList
[1, 'a', 2, 'b', 3, 1, 'a', 2, 'b', 3, 'c']
>>>
```

Toujours plus de tranches

```
>>> myList = ['a', 'b', 'c', 'd']
>>> myList[0:2] = [1, 2, 3, 4]
>>> myList
[1, 2, 3, 4, 'c', 'd']
>>> myList[0:4] = []
>>> myList
['c', 'd']
>>> myList[1:1] = ['x', 'y']
>>> myList
['c', 'x', 'y', 'd']
>>> myList[:0] = myList
>>> myList
['c', 'x', 'y', 'd', 'c', 'x', 'y', 'd']
>>> myList[:3] = [myList[3:]]
>>> myList
[['d', 'c', 'x␣', 'y', 'd'], 'd', 'c', 'x␣', 'y', 'd']
>>> myList[0]
['d', 'c', 'x', 'y', 'd']
>>>
```

Quelques fonctions supplémentaires sur les listes

- *myList*.insert(*i*, *x*) : Insère l'élément *x* à la position *i* dans *myList*.
- *myList*.append(*x*) : Équivalent à *myList*.insert(len(*myList*), *x*).
- *myList*.extend(*otherList*) : Ajoute à la liste *myList* tous les éléments de la liste *otherList*. Équivalent à [len(*myList*):] = *otherList*.
- *myList*.remove(*x*) : Supprime la première occurrence de l'élément *x* dans *myList*.
- *myList*.pop(*[i]*) : Supprime et retourne l'élément en position *i* dans *myList*. Si aucun indice n'est spécifié *myList*.pop() supprime et retourne le dernier élément dans *myList*.

Quelques fonctions supplémentaires sur les listes

```
>>> myList = [1]
>>> myList
[1]
>>> myList.insert(0, 'a')
>>> myList
['a', 1]
>>> myList.insert(2, 'b')
>>> myList
['a', 1, 'b']
>>> myList.append('a')
>>> myList
['a', 1, 'b', 'a']
>>> myList.remove('a')
>>> myList
[1, 'b', 'a']
>>> myList.extend([3, 4, 5])
>>> myList
[1, 'b', 'a', 3, 4, 5]
```

Quelques fonctions supplémentaires sur les listes

```
>>> myList
[1, 'b', 'a', 3, 4, 5]
>>> myList.extend(myList)
>>> myList
[1, 'b', 'a', 3, 4, 5, 1, 'b', 'a', 3, 4, 5]
>>>
>>> myList.remove(myList.pop())
>>> myList
[1, 'b', 'a', 3, 4, 1, 'b', 'a', 3, 4]
>>>
>>> myList.extend([myList.pop(0), myList.pop(0)])
>>> myList
['a', 3, 4, 1, 'b', 'a', 3, 4, 1, 'b']
>>>
>>> myList.extend([myList.pop(1), myList.pop(0)])
>>> myList
[4, 1, 'b', 'a', 3, 4, 1, 'b', 3, 'a']
>>>
```

Quelques fonctions supplémentaires sur les listes

- *myList*.`index(`x`)` : Retourne la position de l'élément x dans *myList*. Cette fonction déclenche une erreur si x n'apparaît pas dans *myList*
- *myList*.`count(`x`)` : Retourne le nombre d'occurrences de l'élément x dans *myList*.
- *myList*.`sort()` : Tri les éléments de *myList*.
- *myList*.`reverse()` : Renverse l'ordre des éléments dans *myList*.

Quelques fonctions supplémentaires sur les listes

```
>>> myList = ['a', 'b', 'c'] * 3
>>> myList
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
>>> myList.index('b')
1
>>> myList[myList.index('b') + 1:].index('b')
2
>>> myList.count('a')
3
>>> myList.sort()
>>> myList
['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c']
>>> myList.reverse()
>>> myList
['c', 'c', 'c', 'b', 'b', 'b', 'a', 'a', 'a']
>>>
```

Utiliser les listes comme des piles

```
>>> # lists as stacks
...
>>> stack = ['a', 'b', 'c']
>>> stack.append('d')
>>> stack.append('e')
>>> stack
['a', 'b', 'c', 'd', 'e']
>>> stack.pop()
'e'
>>> stack
['a', 'b', 'c', 'd']
>>> stack.pop()
'd'
>>> stack.pop()
'c'
>>> stack
['a', 'b']
>>>
```

Utiliser les listes comme des files

```
>>> # lists as queues
...
>>> queue = ['a', 'b', 'c']
>>> queue.append('d')
>>> queue.append('e')
>>> queue.pop(0)
'a'
>>> queue
['b', 'c', 'd', 'e']
>>> queue.pop(0)
'b'
>>> queue.pop(0)
'c'
>>> queue
['d', 'e']
>>>
```

Programmation fonctionnelle

Il existe trois fonctions intégrées qui permettent de manipuler plus facilement les listes :

- `filter()`
- `map()`
- `reduce()`

L'utilisation de ces fonctions n'est néanmoins plus encourager

Programmation fonctionnelle : `filter()`

`filter(`*`fonction`*`,`*`séquence`*`)` : Retourne une liste contenant les éléments de *séquence* pour lesquels *fontion*`(`*élément*`)` retourne `vrai`.

```
>>> def isEven(e): return e % 2 == 0
...
>>> seq = range(1,20,3)
>>> seq
[1, 4, 7, 10, 13, 16, 19]
>>> filter(isEven, seq)
[4, 10, 16]
>>>
>>> def containsE(x): return 'e' in x
...
>>> seq = ["pomme", "ananas", "poire", "melon", "avocat"]
>>> seq
['pomme', 'ananas', 'poire', 'melon', 'avocat']
>>> filter(containsE, seq)
['pomme', 'poire', 'melon']
>>>
```

Programmation fonctionnelle : map()

map(*fonction*,*séquence*) : Retourne une liste contenant les éléments de *séquence* obtenus par application de *fontion*(*élément*).

```
>>> def mult2(e): return e*2
...
>>> seq = range(1,10,3)
>>> seq
[1, 4, 7]
>>> map(mult2, seq)
[2, 8, 14]
>>>
>>> seq = ["a", "bb", "ccc"]
>>> map(mult2, seq)
['aa', 'bbbb', 'cccccc']
>>>
```

Programmation fonctionnelle : `reduce()`

`reduce(`*fonction*`,`*séquence*`)` : Retourne une valeur unique construite par l'appel de la fonction (binaire) *fontion* sur les deux premiers éléments de *séquence*, puis sur lé résultat et l'élément suivant.

```
>>> def addition(x, y): return x + y
...
>>> seq = range(1, 6)
>>> seq
[1, 2, 3, 4, 5]
>>> reduce(addition, seq)
15
>>>
>>> seq = ["a", "bb", "ccc"]
>>> reduce(addition, seq)
'abbccc'
>>>
```

Programmation fonctionnelle et fonctions anonymes

Il ets bien sûr possible de combiner les fonctions `filter`, `map` et `reduce` avec des fonctions anonymes.

```
>>> seq = range(1,10)
>>> seq
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> filter(lambda x : x%2 == 0, seq)
[2, 4, 6, 8]
>>>
>>> filter(lambda x : 2*x, seq)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> reduce(lambda x,y : x+y, seq)
45
>>>
```