Informatique VII
Listes

Laurent Kaczmarek

PCSI^2 2013-2014
Lycée Louis Le Grand

Lundi 7 octobre 2013

Les listes sous Python

Les listes sous Python

- Le type `list` permet de regrouper dans une même variable plusieurs objets (ou *éléments*) rangés dans un ordre déterminé. Une liste peut être de longueur quelconque (dans la limite des capacités de l'ordinateur). Ses éléments peuvent être de types différents. Tous les objets manipulés par Python peuvent être éléments d'une liste.

Les listes sous Python

- Le type `list` permet de regrouper dans une même variable plusieurs objets (ou *éléments*) rangés dans un ordre déterminé. Une liste peut être de longueur quelconque (dans la limite des capacités de l'ordinateur). Ses éléments peuvent être de types différents. Tous les objets manipulés par Python peuvent être éléments d'une liste.
- Pour déclarer une liste, on énumère, entre crochets, des valeurs séparées par des virgules. La liste vide est notée `[]`.

Les listes sous Python

- Le type `list` permet de regrouper dans une même variable plusieurs objets (ou *éléments*) rangés dans un ordre déterminé. Une liste peut être de longueur quelconque (dans la limite des capacités de l'ordinateur). Ses éléments peuvent être de types différents. Tous les objets manipulés par Python peuvent être éléments d'une liste.
- Pour déclarer une liste, on énumère, entre crochets, des valeurs séparées par des virgules. La liste vide est notée `[]`.
- Exemple :

```
>>> T=[1,6,False,print,[True,2]]
>>> type(T),type(T[2]),type(T[4][1])
<class 'list'>, <class 'bool'>, <class 'int'>
```

La commande `range`

La commande range

- La commande `range` permet de créer un objet de type `range` listant une suite arithmétique d'entiers.

La commande range

- La commande `range` permet de créer un objet de type `range` listant une suite arithmétique d'entiers.
- Par exemple :

```
>>> list(range(2,6))
[2, 3, 4, 5]
```

```
>>> list(range(4))
[0,1, 2, 3]
```

La commande range

- La commande `range` permet de créer un objet de type `range` listant une suite arithmétique d'entiers.
- Par exemple :

```
>>> list(range(2,6))
[2, 3, 4, 5]
```

```
>>> list(range(4))
[0,1, 2, 3]
```

- Plus précisément, la commande `range(debut,fin,pas)` génère les entiers compris entre debut et `fin-1` avec un pas égal à pas.

La commande range

- La commande `range` permet de créer un objet de type `range` listant une suite arithmétique d'entiers.
- Par exemple :

```
>>> list(range(2,6))
[2, 3, 4, 5]
```

```
>>> list(range(4))
[0,1, 2, 3]
```

- Plus précisément, la commande `range(debut,fin,pas)` génère les entiers compris entre `debut` et `fin-1` avec un pas égal à `pas`.
- On prendra garde aux indices de début et de fin ! L'argument `fin` est par défaut égal à `0` et l'argument `pas` est par défaut égal à `1`.

Création d'une liste

Définition en extension

Création d'une liste

Définition en extension

- On peut générer une liste connaissant l'expression de son terme général en fonction d'un entier naturel au moyen de la commande `range` :

```
L=[expr for k in range(debut,fin,pas)]
```

où `expr` est une expression dépendant de `k`.

Création d'une liste

Définition en extension

- On peut générer une liste connaissant l'expression de son terme général en fonction d'un entier naturel au moyen de la commande `range` :

```
L=[expr for k in range(debut,fin,pas)]
```

où `expr` est une expression dépendant de `k`.

- Exemple :

```
>>> L=[k**2 for k in range(7)]
>>> L
[0,1,4,9,16,25,36]
```

Création d'une liste

Définition en extension

- On peut générer une liste connaissant l'expression de son terme général en fonction d'un entier naturel au moyen de la commande `range` :

```
L=[expr for k in range(debut,fin,pas)]
```

où `expr` est une expression dépendant de `k`.

- Exemple :

```
>>> L=[k**2 for k in range(7)]
>>> L
[0,1,4,9,16,25,36]
```

- Ce mode de définition est proche de la définition mathématique d'un ensemble en extension :

$$\left\{k^2; k \in [\![0, 6]\!]\right\}$$

La concaténation

La concaténation

- Les listes peuvent être *concaténées* (i.e. mises bout à bout) avec l'opérateur +.

La concaténation

- Les listes peuvent être *concaténées* (i.e. mises bout à bout) avec l'opérateur +.
- Exemple :

```
>>> entiers = [0,1,2,3]
>>> suivants = [4,5,6]
>>> tous = entiers + suivants
>>> tous
[0,1,2,3,4,5,6]
```

La concaténation

- Les listes peuvent être *concaténées* (i.e. mises bout à bout) avec l'opérateur +.
- Exemple :

```
>>> entiers = [0,1,2,3]
>>> suivants = [4,5,6]
>>> tous = entiers + suivants
>>> tous
[0,1,2,3,4,5,6]
```

Longueur d'une liste

La commande `len` permet de calculer la *longueur* d'une liste, i.e. son nombre d'éléments.

```
>>> len(tous)
7
```

Accès en lecture et en enregistrement

Accès en lecture et en enregistrement

- On peut accéder directement à un élément d'une liste `L` connaissant sa position `i` dans la liste par `L[i]`.

Accès en lecture et en enregistrement

- On peut accéder directement à un élément d'une liste `L` connaissant sa position `i` dans la liste par `L[i]`.
- Le premier élément d'une liste est d'indice 0.

Accès en lecture et en enregistrement

- On peut accéder directement à un élément d'une liste `L` connaissant sa position `i` dans la liste par `L[i]`.
- Le premier élément d'une liste est d'indice 0.
- Exemple :

```
>>> tous[len(tous)-1]
6
```

Accès en lecture et en enregistrement

- On peut accéder directement à un élément d'une liste `L` connaissant sa position `i` dans la liste par `L[i]`.
- Le premier élément d'une liste est d'indice 0.
- Exemple :

```
>>> tous[len(tous)-1]
6
```

- Les termes de la listes sont modifiables par une affectation.

Accès en lecture et en enregistrement

- On peut accéder directement à un élément d'une liste `L` connaissant sa position `i` dans la liste par `L[i]`.
- Le premier élément d'une liste est d'indice 0.
- Exemple :

```
>>> tous[len(tous)-1]
6
```

- Les termes de la listes sont modifiables par une affectation.
- Exemple :

```
>>> tous[3]=5
>>> tous
[0,1,2,5,4,5,6]
```

La technique de slicing (*tranchage*)

La technique de slicing (*tranchage*)

- On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est appelée tranchage (ou *slicing*) :

```
nom[debut:fin]
```

La technique de slicing (*tranchage*)

- On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est appelée tranchage (ou *slicing*) :

  `nom[debut:fin]`

- Exemple :

```
>>> tous[1:3]
[1,2]
```

La technique de slicing (*tranchage*)

- On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est appelée tranchage (ou *slicing*) :

  `nom[debut:fin]`

- Exemple :

```
>>> tous[1:3]
[1,2]
```

- On peut également donner un troisième indice lors de l'extraction d'une sous-liste, qui correspond au *pas* de l'extraction.

La technique de slicing (*tranchage*)

- On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est appelée tranchage (ou *slicing*) :

  `nom[debut:fin]`

- Exemple :

```
>>> tous[1:3]
[1,2]
```

- On peut également donner un troisième indice lors de l'extraction d'une sous-liste, qui correspond au *pas* de l'extraction.
- Exemple :

```
>>> entiers = [0,1,2,3,4,5,6,7,8]
>>> entiers[0:8:2]
[0,2,4,6]
```

La technique de slicing (*tranchage*)

La technique de slicing (*tranchage*)

- Considérons la commande `liste[i:j:k]`. Si `i` (resp. `j`) est négatif, alors `len(liste)+i` (resp. `len(liste)+j`) lui est substitué. Si `i` et `j` sont positifs, alors les indices considérés sont les `i+n*k`, où $0 \leqslant n < \frac{j-i}{k}$.

La technique de slicing (*tranchage*)

- Considérons la commande `liste[i:j:k]`. Si `i` (resp. `j`) est négatif, alors `len(liste)+i` (resp. `len(liste)+j`) lui est substitué. Si `i` et `j` sont positifs, alors les indices considérés sont les `i+n*k`, où $0 \leqslant n < \frac{j-i}{k}$.
- L'*affectation* dans des tranches ainsi que la *destruction* (commande `del`) d'éléments de la liste est possible.

La technique de slicing (*tranchage*)

- Considérons la commande `liste[i:j:k]`. Si `i` (resp. `j`) est négatif, alors `len(liste)+i` (resp. `len(liste)+j`) lui est substitué. Si `i` et `j` sont positifs, alors les indices considérés sont les `i+n*k`, où $0 \leqslant n < \frac{j-i}{k}$.
- L'*affectation* dans des tranches ainsi que la *destruction* (commande `del`) d'éléments de la liste est possible.
- Exemple :

```
>>> a=[4,True,1.,3.14]
>>> a[1:1]
[]
>>> a[0:3]=[1,12]; print(a) [1,12,3.14]
>>> a[1:1]=[3,True]; print(a)
[1,3,True,12,3.14]
```

Copies d'une liste

Copies vs. alias

Copies d'une liste

- Pour *copier* la liste `L` dans la liste `M`, on utilisera la commande `M = L[:]`.

Copies vs. alias

Copies d'une liste

- Pour *copier* la liste `L` dans la liste `M`, on utilisera la commande `M = L[:]`.
- La commande `M=L` crée un *alias* : les noms `M` et `L` pointent vers un même objet. Ainsi, la modification de `M` entraîne la modification de `L`.

Copies d'une liste

- Pour *copier* la liste `L` dans la liste `M`, on utilisera la commande `M = L[:]`.
- La commande `M=L` crée un *alias* : les noms `M` et `L` pointent vers un même objet. Ainsi, la modification de `M` entraîne la modification de `L`.
- Exemple :

```
>>> L=[3,True,2]
>>> L2=L
>>> L[1]=False
>>> L2
[3,False,2] >>>
```

```
L=[3,True,2]
>>> L3=L[:]
>>> L3[1]=[False]
>>> L3
[3,False,2]
>>> L
[3,True,2]
```

Copies vs. alias

Attention, cela est particulier aux listes...

Copies vs. alias

Attention, cela est particulier aux listes...

- En dehors du type liste, le signe d'affectation = a pour effet de copier une variable dans une autre.

Copies vs. alias

Attention, cela est particulier aux listes...

- En dehors du type liste, le signe d'affectation = a pour effet de copier une variable dans une autre.
- Exemple :

```
>>> a=2
>>> b=a
>>> a,b
(2,2)
```

```
>>> a=4
>>> a,b
(4,2)
```

Ajouter un élément à une liste

- Pour *ajouter* un élément à la fin d'une liste, plusieurs méthodes sont possibles.

Ajouter un élément à une liste

- Pour *ajouter* un élément à la fin d'une liste, plusieurs méthodes sont possibles.

  - ```
    >>> t=[6]
    >>> t[len(t):]=[1];
    >>> print(t)
    [6,1]
    ```
    ```

Ajouter un élément à une liste

- Pour *ajouter* un élément à la fin d'une liste, plusieurs méthodes sont possibles.
  -
```
>>> t=[6]
>>> t[len(t):]=[1];
>>> print(t)
[6,1]
```
  -
```
>>> t.append(2);
>>> print(t)
[6,1,2]
```

Ajouter un élément à une liste

- Pour *ajouter* un élément à la fin d'une liste, plusieurs méthodes sont possibles.

  - ```
    >>> t=[6]
    >>> t[len(t):]=[1];
    >>> print(t)
    [6,1]
    ```

  - ```
    >>> t.append(2);
    >>> print(t)
    [6,1,2]
    ```
  - ```
    >>> t=t+[5];
    >>> print(t)
    [6,1,2,5]
    ```
    ```

Ajouter un élément à une liste

- Pour *ajouter* un élément à la fin d'une liste, plusieurs méthodes sont possibles.
  - ```
    >>> t=[6]
    >>> t[len(t):]=[1];
    >>> print(t)
    [6,1]
    ```
  - ```
    >>> t.append(2);
    >>> print(t)
    [6,1,2]
    ```
  - ```
    >>> t=t+[5];
    >>> print(t)
    [6,1,2,5]
    ```
- La méthode par concaténation est de nature différente des précédentes. Elle n'utilise pas le caractère modifiable des listes mais crée une nouvelle liste. Cette méthode est donc moins efficace en termes de temps d'exécution et d'utilisation de l'espace mémoire, puisqu'elle nécessite de recopier toute la liste (qui peut être longue).
    ```

Supprimer un élément d'une liste

Supprimer un élément d'une liste

- Pour *supprimer* un élément à la fin d'une liste, plusieurs méthodes sont possibles.

Supprimer un élément d'une liste

- Pour *supprimer* un élément à la fin d'une liste, plusieurs méthodes sont possibles.

  - ```
    >>> L=[1,3,1,2]
    >>> L.remove(1)
    >>> L
    [3,1,2]
    ```
    ```

Supprimer un élément d'une liste

- Pour *supprimer* un élément à la fin d'une liste, plusieurs méthodes sont possibles.
  - ```
    >>> L=[1,3,1,2]
    >>> L.remove(1)
    >>> L
    [3,1,2]
    ```
  - ```
    >>> L=[1,3,1,2]
    >>> L.pop()
    2
    >>> L
    [1,3,1]
    ```

Supprimer un élément d'une liste

- Pour *supprimer* un élément à la fin d'une liste, plusieurs méthodes sont possibles.
  - ```
    >>> L=[1,3,1,2]
    >>> L.remove(1)
    >>> L
    [3,1,2]
    ```
  - ```
    >>> L=[1,3,1,2]
    >>> L.pop()
    2
    >>> L
    [1,3,1]
    ```
  - ```
    >>> N=L[:-1]
    >>> N
    [1,3,1]
    ```
    ```

Supprimer un élément d'une liste

- Pour *supprimer* un élément à la fin d'une liste, plusieurs méthodes sont possibles.
  - ```
    >>> L=[1,3,1,2]
    >>> L.remove(1)
    >>> L
    [3,1,2]
    ```
  - ```
    >>> L=[1,3,1,2]
    >>> L.pop()
    2
    >>> L
    [1,3,1]
    ```
  - ```
    >>> N=L[:-1]
    >>> N
    [1,3,1]
    ```
- La dernière méthode est à éviter car passe par une copie de la liste `L[:-1]`.
    ```

Itération sur une liste

Itération sur une liste

- Dans la boucle `for`, l'argument `list` n'est pas nécessairement de type `range` mais peut également être de type liste.

Itération sur une liste

- Dans la boucle `for`, l'argument `list` n'est pas nécessairement de type `range` mais peut également être de type liste.
- La fonction suivante renvoie le minimum d'une liste `L`.

```
def minimum1(L):
    m=L[0]
    for element in L:
        if element<m:
            m=element
    return m
```