Initiation à la programmation

Zhentao Li

22 février 2017

Les listes

Type `list`

- Similaire aux chaînes de caractères, mais chaque element peut être n'importe quoi (au lieu d'un seul caractère).
- Donc, une **liste** est une structure de données qui contient une série de valeurs.
- Création: `[element1, element2, element3]`

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille = [5.0, 1.0, 0.7, 2.0]
>>> mix = ['girafe', 5.0, 'dahu', 2]
>>> animaux
['girafe', 'hippopotame', 'singe', 'dahu']
>>> taille
[5.0, 1.0, 0.7, 2.0]
>>> mix
['girafe', 5.0, 'dahu', 2]
```

Création de `list` (2e méthode)

```
>>> animaux = []
>>> animaux.append('girafe')
>>> animaux.append('hippopotame')
>>> animaux.append('singe')
>>> animaux.append('dahu')
>>> animaux
['girafe', 'hippopotame', 'singe', 'dahu']
```

C'est plutôt une convention d'usage. C'est très pratique pour créer une liste en utilisant une boucle.

Indices

- On peut rappeler ses éléments par leur numéro de position **indice**
- Les indice d'une liste de n éléments commence à 0 et se termine à $n-1$

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> animaux[0]
'girafe'
>>> animaux[1]
'hippopotame'
>>> animaux[3]
'dahu'
```

Opérations + et *

- + concaténation
- * duplication.

```
>>> animaux = ['aigle', 'ecureuil']
>>> animaux + animaux
['aigle', 'ecureuil', 'aigle', 'ecureuil']
>>> animaux * 3
['aigle', 'ecureuil', 'aigle', 'ecureuil', 'aigle',
'ecureuil']
```

Indiçage négatif

```
liste     : ['girafe', 'hippopotame', 'singe', 'dahu']
index pos :        0              1         2        3
index nég :       -4             -3        -2       -1
```

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singe'
>>> animaux[-1]
'dahu'
```

Tranches

Sélectionner une **partie** de liste avec `une_liste[m:n+1]`

```
>>> animaux = ['girafe', 'hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[0:2]
['girafe', 'hippo']
>>> animaux[0:3]
['girafe', 'hippo', 'singe']
>>> animaux[0:]
['girafe', 'hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[:]
['girafe', 'hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[1:]
['hippo', 'singe', 'dahu', 'ornithorynque']
>>> animaux[1:-1]
['hippo', 'singe', 'dahu']
```

L'instruction `range()`

L'instruction `range` vous permet de créer des listes d'entiers de manière simple et rapide.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 5)
[0, 1, 2, 3, 4]
>>> range(15, 21)
[15, 16, 17, 18, 19, 20]
>>> range(0, 1000, 100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

L'instruction `len()`

L'instruction `len` vous permet de connaître la longueur d'une liste.

```
>>> animaux = ['girafe', 'hippo', 'singe', 'dahu']
>>> len(animaux)
4
```

L'instruction `for`

Il arrive souvent que nous voulons une boucle pour traverser les éléments d'une liste ou chaîne. L'instruction `for` permet de simplifier

```
l = ['a', 'b', 'c', 'd']
indice = 0
while indice < len(l):
    element = l[indice]
    # Faire quelque chose avec l'element
    indice = indice + 1
```

à

```
l = ['a', 'b', 'c', 'd']
for element in l:
    # Faire quelque chose avec l'element
```

Autre exemple for

```
for lettre in "ciao":
    print lettre # c i a o

for x in [2, 'a', 3.14]:
    print x # 2 a 3.14

for i in range(5):
    print i # 0 1 2 3 4
```

Syntaxe

```
for variable in list:
    instruction1
    instruction2
```

L'instruction break

L'instruction `break` interrompt les boucles `for` comme pour les boucles `while`.

```
for x in range(1, 11):
    if x == 5:
        break
    print x,

print "\nBoucle interrompue pour x =", x
```

Résultat

```
1 2 3 4
Boucle interrompue pour x = 5
```

L'instruction continue

L'instruction `continue` fait passer à l'itération suivante les instructions `while` ou `for`.

```
for x in range(1, 11):
    if x == 5:
        continue
    print x,

print "\nLa boucle a sauté la valeur 5"
```

Résultat

```
1 2 3 4 6 7 8 9 10
La boucle a sauté la valeur 5
```

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

`liste2` a été réassignée mais pas `liste1` alors que sa valeur a changée! Essayez le même exercice avec d'autre types, `str` par exemple.

Mutation

Les `lists` sont *mutable*: la valeur d'une variable peut changer sans réassignation.

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste2[2] = 0
>>> liste1
[1, 2, 0]
```

`liste2` a été réassignée mais pas `liste1` alors que sa valeur a changée! Essayez le même exercice avec d'autre types, `str` par exemple.

Attention: certaines opérations renvoient des nouvelles listes alors que d'autre modifient la variable.

Type `tuple`

- Un `tuple` est une liste non-mutable.
- Création: Un `tuple` est créé par des parenthèses au lieu de crochets.

```
>>> t = (1, 'a', 2)
>>> t
(1, 'a', 2)
>>> t[1]
'a'
>>> t2 = (1,)
>>> t2
(1,)
```

Il faut au moins une virgule pour créer un `tuple`.

Unpacking

Pour récupérer tous les entrées d'une `list`, au lieu de

```
>>> animaux = ['girafe', 'hippopotame', 'singe']
>>> a = animaux[0]
>>> b = animaux[1]
>>> c = animaux[2]
```

nous pouvons utiliser la syntaxe suivante.

```
>>> animaux = ['girafe', 'hippopotame', 'singe']
>>> a, b, c = animaux
```

La syntaxe est donc

```
variable1, variable2, ... = la_liste
```

Opérations sur `list` et `tuple` (avancé)

D'autre opérations qui sont des mutations les `lists`: `append`, `extend`, `insert`, `pop`, `remove`, `reverse` et `sort`.

```
l = range(5)
l.append(10)
l2 = ('a', 1)
l.extend(l2)
l.remove(1)
element = l.pop()
premier_element = l.pop(0)
l.reverse()
l.sort()
```

Opérations sur `list` et `tuple` (avancé)

D'autre opérations qui sont des mutations les `lists`: `append`, `extend`, `insert`, `pop`, `remove`, `reverse` et `sort`.

```
l = range(5)
l.append(10)
l2 = ('a', 1)
l.extend(l2)
l.remove(1)
element = l.pop()
premier_element = l.pop(0)
l.reverse()
l.sort()
```

Attention: Une erreur récurrente est d'appeler `l2 = l.append(10)`. Cela met `l2` à `None`.
D'autre opérations sur les `lists` et `tuples`: `index` et `count`.

Compréhension de liste (très? avancé)

Un motif récurrent d'initialisation d'une liste est sa création par une boucle.

```
carres = []
for i in range(10):
    carres.append(i*i)
```

Dans l'intérêt de la lisibilité du programme, une nouvelle syntaxe a été ajoutée.

```
carres = [i*i for i in range(10)]
carres_impairs = [i*i for i in range(10) if i%2 == 1]
```

La syntaxe est

```
[expression for variable in une_liste]
[expression for variable in une_liste if condition]
```

Cette écriture ressemble à la notation d'ensemble en mathématiques.

Les fonctions

Définition d'une fonction

Rappel: Deux formes d'appel

- `nom_de_fonction(parametre1, parametre2, ...)`
- `variable.nom_de_fonction(parametre1, parametre2, ...)`

Nous allons maintenant voir comment définir de nouvelles fonctions. Les **fonctions** permettent de décomposer les programmes en sous-programmes et de réutiliser des morceaux de programmes.

Exemple de définition d'une fonction

```
def compter_lettre(lettre, texte):
    n = 0
    for c in texte :
        if c == lettre :
            n += 1
    return "nombre d'occurences de la lettre %s : %s" \
               % (lettre, n)

print compter_lettre('e', 'je reviens')
```

Résultat

```
nombre d'occurrences de la lettre e : 3
```

Syntaxe

- Une fonction renvoie une valeur de sortie à partir de paramètres d'entrées.
- Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    instruction1
    instruction2
    ...
```

- `return valeur` dans le bloc d'instructions: fin d'exécution et retour de la valeur.

Syntaxe

- Une fonction renvoie une valeur de sortie à partir de paramètres d'entrées.
- Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    instruction1
    instruction2
    ...
```

- `return valeur` dans le bloc d'instructions: fin d'exécution et retour de la valeur.

Remarque (avancée): Une fois définie, une fonction peut être manipulée comme tout autre variable!

```
>>> print compter_lettre
<function compter_lettre at 0x7f485e9d45f0>
>>> type(compter_lettre)
<type 'function'>
```

Appel d'une fonction

- Une fois qu'une fonction `f` a été définie, elle peut être utilisée dans une expression particulière qu'on nomme un appel de fonction et qui a la forme `f(v1,v2,...,vn)`, où `v1`, `v2`, ..., `vn` sont des expressions dont la valeur est transmise au paramètres.
- Cela veut dire que d'abord `parametre1` est assigné la valeur `v1`, `parametre2` est assigné la valeur `v2`, et ainsi de suite. Ensuite le bloc d'instruction est exécuté ligne par ligne, jusqu'à la fin du bloc ou une instruction `return`.
- Si la fin du bloc est atteint, la valeur `None` est retournée.

Paramètres par défaut

Python offre un mécanisme d'instanciation des paramètres par défaut. On peut écrire la liste des paramètres en entête d'une définition de fonction. Par exemple

```
>>> def pluriel(mot, famille = 'standard'):
...     if famille == 'standard':
...         return mot + 's'
...     if famille == 's':
...         return mot
...     if famille == 'oux':
...         return mot + 'x'
...     if famille == 'al':
...         return mot[:-1] +'ux'
...
>>> print pluriel('maison')
'maisons'
>>> print pluriel('souris', 's')
'souris'
>>> print pluriel('chou', 'oux')
'choux'
>>> print pluriel('cheval', 'al')
'chevaux'
```

Paramètres par défaut

- Syntaxe

```
def nom_de_la_fonction(p_1, ..., p_k, d_1=expr_1,..., d_n=expr_n):
    instruction1
    instruction2
    ...
```

- Les k premiers paramètres doivent obligatoirement être précisés à l'appel de fonction mais pas les n derniers. L'appel de fonction se fait donc avec k arguments au minimum et $k+n$ arguments au maximum. Si un paramètre `d_i` n'est pas instancié explicitement, il prend la valeur par défaut de `expr_i`.
- Il est possible d'instancier certain paramètre et pas d'autre. Par exemple, avec un appel `pluriel(p_1, ..., p_k, d_4 = 10)`. `d_4` prends la valeur `10` au lieu de `expr_4`.

Variables locales et variables globales

- Les variables qui sont introduites dans la définition d'une fonction peuvent être utilisées dans la suite de la définition mais **pas à l'extérieur de la fonction**.
- Ces variables sont dites **locales** par opposition aux variables **globales** qui sont introduites à l'extérieur de la définition d'une fonction et qui peuvent être utilisées à l'intérieur comme à l'extérieur de cette définition.
- Lorsque le même nom est utilisé pour introduire une variable locale et une variable globale, Python distingue bien deux variables différentes mais à l'intérieur de la définition de la fonction, c'est à la variable locale auquel le nom réfère.

Variables locales et variables globales

```
>>> def f(x):
...     y=2
...     return x + y
>>> print f(3)
5
>>> print y
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <mo
    print y
NameError: name 'y' is not defined
>>> u = 7
>>> def g(v):
...     return u * v
>>> print g(2)
14
```

```
>>> def h(u):
...     return u
>>> print h(3)
3
>>> print u
7
>>> def k(w) :
...     u = 5
...     return w+u
>>> print k(3)
8
>>> print u
7
```