Initiation à la programmation: Dictionnaires et Classes

Zhentao Li

8 mars 2017

Dictionnaires

Dictionnaires

Les listes nous permettent d'accéder aux éléments de la liste par indices.

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

Dictionnaires

Les listes nous permettent d'accéder aux éléments de la liste par indices.

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

Mais parfois, on aimerais que les indices soient autre chose que des numéros. Des chaînes de caractères, par exemple.

```
>>> nombre["giraffe"]
3
>>> nombre["singe"]
5
>>> emplacement["giraffe"]
"Cage nord-ouest"
```

Dictionnaires

Les listes nous permettent d'accéder aux éléments de la liste par indices.

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

Mais parfois, on aimerais que les indices soient autre chose que des numéros. Des chaînes de caractères, par exemple.

```
>>> nombre["giraffe"]
3
>>> nombre["singe"]
5
>>> emplacement["giraffe"]
"Cage nord-ouest"
```

Une telle représentation de l'information est possible avec le dictionnaire.

```
>>> nombre = {"giraffe":3, "singe":5}
>>> emplacement = {"giraffe":"Cage nord-ouest"}
```

Dictionnaire

- La *liste* associe à chaque *indice* une valeur.
  Le *dictionnaire* associe à chaque *clé* une valeur.
- Une clé de dictionnaire peut être un *chiffre*, une *chaîne de caractères*, un *n-tuplet*, etc. Mais pas une *liste* ou un autre *dictionnaire*.
- Un dictionnaire est créé en utilisant les accolades `{}` avec la syntaxe
  `{cle1: valeur1, cle2: valeur2, ...}`

Dictionnaire

- La *liste* associe à chaque *indice* une valeur.
  Le *dictionnaire* associe à chaque *clé* une valeur.
- Une clé de dictionnaire peut être un *chiffre*, une *chaîne de caractères*, un *n-tuplet*, etc. Mais pas une *liste* ou un autre *dictionnaire*.
- Un dictionnaire est créé en utilisant les accolades `{}` avec la syntaxe
  `{cle1: valeur1, cle2: valeur2, ...}`

Remarque: Les dictionnaires sont un type de base en Python mais est inhabituelle pour d'autre langages de programmation.
Dans ce sens, ils ne sont pas strictement nécessaires. De plus il est possible de simuler un dictionnaire avec des listes (c'est-à-dire écrire un ensemble de fonctions qui remplissent les mêmes fonctions que celle d'un dictionnaire).

Opération sur les dictionnaires

- Ajout ou mise à jour de valeur.

```
>>> nombre['dahu'] = 1
>>> nombre["singe"] = 6
>>> nombre
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

Opération sur les dictionnaires

- Ajout ou mise à jour de valeur.

```
>>> nombre['dahu'] = 1
>>> nombre["singe"] = 6
>>> nombre
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

- Déterminer si une clé existe dans un dictionnaire.

```
>>> 'singe' in nombre
True
>>> 'ornithorynque' in nombre
False
```

Opération sur les dictionnaires

- Ajout ou mise à jour de valeur.

```
>>> nombre['dahu'] = 1
>>> nombre["singe"] = 6
>>> nombre
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

- Déterminer si une clé existe dans un dictionnaire.

```
>>> 'singe' in nombre
True
>>> 'ornithorynque' in nombre
False
```

- Obtenir les clés, valeurs ou paires clé-valeur comme liste.

```
>>> nombre.keys()
['giraffe', 'dahu', 'singe']
>>> nombre.values()
[3, 1, 6]
>>> nombre.items()
[('giraffe', 3), ('dahu', 1), ('singe', 6)]
```

Opération sur les dictionnaires

- Conversion de liste de paires en dictionnaire.

```
>>> liste_de_paires = nombre.items()
>>> dict(liste_de_paires)
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

Opération sur les dictionnaires

- Conversion de liste de paires en dictionnaire.

```
>>> liste_de_paires = nombre.items()
>>> dict(liste_de_paires)
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

- Itération sur les clés

```
>>> for cle in nombre:
...     print cle
...
giraffe
dahu
singe
```

Opération sur les dictionnaires

- Conversion de liste de paires en dictionnaire.

```
>>> liste_de_paires = nombre.items()
>>> dict(liste_de_paires)
{'giraffe': 3, 'dahu': 1, 'singe': 6}
```

- Itération sur les clés

```
>>> for cle in nombre:
...     print cle
...
giraffe
dahu
singe
```

- Enlever une paire clé-valeur.

```
>>> del nombre['giraffe']
>>> nombre
{'dahu': 1, 'singe': 6}
```

Compréhension de dictionnaire (très avancé)

Comme pour les listes, une notation « mathématique » compacte existe pour la création de dictionnaires.

```
# Pour les listes
carres_impairs = [i*i for i in range(10) if i%2 == 1]
# Pour les dictionnaires
ordre = {cara: ord(cara) for cara in 'abcde' if cara != 'd'}
>>> print ordre
{'a': 97, 'c': 99, 'b': 98, 'e': 101}
```

Classes

Définition de nouveaux types avec `class`

Une nouvelle classe (nouveau « type ») peut être créé avec

```
class MaClasse:
    pass

instance = MaClasse()
```

Définition de nouveaux types avec `class`

Une nouvelle classe (nouveau « type ») peut être créé avec

```
class MaClasse:
    pass

instance = MaClasse()
```

Comme la définition de fonctions, aucun effet d'exécution immédiate.
Chaque copie est une *instance* de la classe `MaClasse`.
Voir `help("class")`.

Attributs et méthodes

Une classe est constituée de

- **attributs**: un dictionnaire
- **méthodes**: un regroupement de fonction qui seront notamment utiliser pour modifier ou obtenir de l'information sur des instances de la classe.

Attributs et méthodes

Une classe est constituée de

- **attributs**: un dictionnaire
- **méthodes**: un regroupement de fonction qui seront notamment utiliser pour modifier ou obtenir de l'information sur des instances de la classe.

La notion de classes fait partie de la "programmation orienté objet". L'intuition est que les instances sont des objets que nous manipulons. Soit en les examinant (attributs), soit en les modifiants (méthodes).

Exemple attribut

Exemple: les nombres complexes qui ont deux attributs: `real` et `imag`.

```
>>> x = complex(1, 2)
>>> x.real
1.0
>>> x.imag
2.0
>>> x = x + 3
>>> x.real
4.0
```

Les attributs sont accédés et assignés avec la syntaxe `nom_de_variable.nom_de_l_attribut`.

Remarque: Les fonctions de conversion de types du premier cours étaient des initialisations de variables.

Exemple de méthode

```
>>> x.conjugate()
(1-2j)
```

Nous avons déjà vu beaucoup d'exemples de méthodes pour `str`, `list`, `dict` comme `list.append` et `dict.keys`.
Rappel: La deuxième formes d'appel d'une fonction `variable.nom_de_fonction(parametre1, parametre2, ...)` est un appel à une méthode.
La définition de méthode nous permet de créer de nouvelle fonctions qui sont appelé de cette façon.

Déclaration d'une classe: un exemple

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def distance(self, autre_point):
    distance_carre = (self.x - autre_point.x)**2 + (self.y - autre_point.y)**2
    return distance_carre**0.5

point = Point(10, 20)
print point.x, point.y
point.x += 3
print point.x, point.y
origine = Point(0, 0)
print point.distance(origine)
```

Déclaration d'une classe: syntaxe

La forme générale que prends la déclaration d'une classe est la suivante.

```
class MaClasse:
    def __init__(self, parametre1, parametre2, ...):
        # Blocs d'instructions
        # Inclut typiquement l'initialisation
        # des attributs de l'instance.

    def methode1(self, parametre1, ...):
        # Bloc d'instructions

    def methode2(self, parametre1, ...):
        # Bloc d'instructions

instance = MaClasse()
print instance.attribut1
```

Déclaration d'une classe: syntaxe

La forme générale que prends la déclaration d'une classe est la suivante.

```
class MaClasse:
    def __init__(self, parametre1, parametre2, ...):
        # Blocs d'instructions
        # Inclut typiquement l'initialisation
        # des attributs de l'instance.

    def methode1(self, parametre1, ...):
        # Bloc d'instructions

    def methode2(self, parametre1, ...):
        # Bloc d'instructions

instance = MaClasse()
print instance.attribut1
```

La méthode `__init__` est spéciale. Elle est appelée à la création d'une instance.

Déclaration d'une classe: syntaxe

La forme générale que prends la déclaration d'une classe est la suivante.

```
class MaClasse:
    def __init__(self, parametre1, parametre2, ...):
        # Blocs d'instructions
        # Inclut typiquement l'initialisation
        # des attributs de l'instance.

    def methode1(self, parametre1, ...):
        # Bloc d'instructions

    def methode2(self, parametre1, ...):
        # Bloc d'instructions

instance = MaClasse()
print instance.attribut1
```

La méthode `__init__` est spéciale. Elle est appelée à la création d'une instance.
Autres exemples: `__add__`, `__sub__`, `__str__`, `__getitem__`

Similarités entre méthodes et fonctions

Le premier paramètre `self` d'une méthode est toujours l'instance de la classe.
Comparer cette fonction avec la méthode `distance`.

```
def distance(premier_point, autre_point):
    distance_carre = (premier_point.x - autre_point.x)**2 + (
    return distance_carre**0.5

point = Point(10, 20)
origine = Point(0, 0)
print point.distance(origine)
print distance(point, origine)
```

Même résultat.

Attributs de classes

Vous pouvez aussi avoir des attributs de classes (plutôt que des attributs d'instances) mais c'est plus rarement utilisé.

```
class MaClasse:
    x = 10
    chaine = "abc"

print MaClasse.x # 10
MaClasse.x = 20
print MaClasse.x # 20
instance = MaClasse()
print instance.x # 20
instance.x = 30
print MaClasse.x # 20
print instance.x # 30
```

Héritage et polymorphisme (avancé)

Déclarer une nouvelle classe qui a presque les mêmes méthodes.
Exemple:

```
class Rectangle:
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur

    def volume(self):
        return self.longueur * self.largeur

class TriangleRectangle(Rectangle):
    def volume(self):
        return self.longueur * self.largeur / 2

rectangle = Rectangle(10, 20)
print rectangle.volume()  # 200
triangle = TriangleRectangle(10, 20)
print triangle.volume()  # 100
```

`__init__` non-redéfinie pour `TriangleRectangle`.

Héritage: syntaxe

La syntaxe est

```
class NouvelleClasse(AncienneClasse):
    # Definition des methodes de la nouvelle classe
```

Il est possible d'appeler les méthode de l'ancienne classe (que nous avons possiblement remplacé) avec `AncienneClasse.nom_de_la_methde(parametre1, ...)`

Une remarque sur l'abstraction et les boîtes noires

Abstraction

- La création et l'utilisation de classes illustre le principe de l'abstraction: avoir des objets qu'on utilise de façon abstraite sans connaître sont fonctionnement interne. On fait simplement des appels aux méthodes disponibles en supposant qu'elle font ce que leurs noms indiquent (par ex, `un_rectangle.volume()` nous donne bien le volume et pas quelque chose d'autre).
- Cette idée permet de décomposer un problème en plus petit problèmes: tant que chaque morceaux fonctionne comme prévu, le système en entier a le fonctionnement prévu.
- En fait (même sans les classes), cette idée de décomposer un problème en plus petits problèmes est toujours utilisée pour écrire des programmes. Il est trop difficile pour un human de garder le tout en tête d'un coup! Vaut mieux travailler sur un morceaux à la fois.

Boîtes noires

- L'envers de la medaille est que si on a trops d'abstraction entre notre programme et ce que l'on cherche à faire, nous aurons beaucoup de difficulté à écrire et modifier notre programme dû à la taille de l'édifice sur lequel il est bâti. Chaque ligne est aussi un endroit potentiel d'erreurs.
- C'est surtout le cas lorsque nous avons trops de couches d'abstraction dont plusieurs n'ajoute pas grand chose (chaque couche accède uniquement aux fonctions de la couche précédente).