Initiation au développement

6 — Ensembles et dictionnaires

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

1. Les ensembles
2. Les dictionnaires

Structures de données en Python

- On a déjà vu des itérables modifiables (`list`) ou non (`str`, `tuple`) ;
- On va voir des structures plus avancées aujourd'hui ;
- Le choix de la structure dépendra de vos besoins ;
  - l'ordre est-il important ?
  - veut-on autoriser les doublons ?
  - a-t-on besoin d'indices ?
  - a-t-on besoin de modifier les éléments ?
  - ...

- On verra les structures les plus utilisées aujourd'hui, mais il en existe beaucoup d'autres (voir par exemple le module **`collections`**) ;

Python pas à pas

Les ensembles

Les `set`

- Les `set` sont des itérables modifiables représentant des ensembles ; donc :
  1. chaque élément ne peut y être **qu'une seule fois** ;
  2. **il n'y a pas d'ordre**, donc il n'y a pas de positions . . .
  3. . . . et donc l'opérateur `[]` ne fonctionne pas ;

Les ensembles sont modifiables, mais ne peuvent contenir que des **éléments non modifiables**.

Utilisation des `set`

- Les `set` se déclarent comme les listes, mais avec des accolades :

```
>>> S = {3, 1, 4, 6, 2}
```

- Ou à l'aide de la fonction `set(iterable)`, aussi utile pour le transtypage :

```
>>> S = set([3, 1, 4, 6, 2])
>>> ensemble_vide = set()
```

- Comme ce sont des itérables, on peut utiliser `in` et `len` exactement de la même manière que pour les chaînes, les tuples, les listes, ...

Opérations ensemblistes

- Les opérations que vous connaissez sur les ensembles sont disponibles :

math	python	Méthode
$s \cup t$	`s	t`	`s.union(t)`
$s \cap t$	`s & t`	`s.intersection(t)`
$s \setminus t$	`s - t`	`s.difference(t)`
$s \Delta t$	`s ^ t`	`s.symmetric_difference(t)`

- Ces méthodes ne modifient ni s, ni t ; si c'est votre but :

Opération	Méthode
`s = s	t`	`s.update(t)`
`s = s & t`	`s.intersection_update(t)`
`s = s - t`	`s.difference_update(t)`
`s = s ^ t`	`s.symmetric_difference_update(t)`

Autres méthodes modifiant les ensembles

Les autres méthodes suivantes permettent de modifier les ensembles :

- `s.add(x)` ajoute `x` à `s` ;
- `s.remove(x)` retire `x` de `s` si `s` le contient — sinon, elle provoque une erreur ;
- `s.discard(x)` retire `x` de `s` sans erreur si `s` ne le contient pas.
- enfin, `s.pop()` (sans paramètre) retire et renvoie un élément aléatoire de `s` s'il n'est pas vide — sinon, elle provoque une erreur.

Comparaison d'ensembles

On peut aussi comparer les ensembles à l'aide des opérateurs suivants :

math	python
$s = t$	`s == t`
$s \neq t$	`s != t`
$s \subset t$	`s < t`
$s \subseteq t$	`s <= t`
$s \supset t$	`s > t`
$s \supseteq t$	`s >= t`

frozenset

- Les `frozenset` sont des ensembles non-modifiables ;
- Ils sont principalement utiles quand on veut représenter des collections d'ensembles, ou dans le contexte des dictionnaires (voir plus loin) :

Exemple (sous-ensembles de {1, 2, 3} de taille 2)

```
>>> S = set([{1, 2}, {1, 3}, {2, 3}])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> S = set([frozenset([1, 2]), frozenset([1, 3]),
             frozenset([2, 3])])
>>> S
{frozenset({1, 3}), frozenset({1, 2}), frozenset({2, 3})}
```

Test d'efficacité

Exercice (`rechercheiterables.py`)

Pour chacun des itérables vus jusqu'ici (`set`, `list`, `tuple`, `frozenset`), créez un itérable de ce type contenant $\{0, 1, 2, \ldots, n-1\}$ et écrivez une fonction vérifiant si chacun de ces entiers est dans l'itérable. Comparez le temps d'exécution avec le module **`time`**.

Python pas à pas

Les dictionnaires

Dictionnaires

- Les **dictionnaires** sont comme les listes, mais :
  1. les indices peuvent être autre chose que des naturels ;
  2. il n'y a pas d'ordre ;
- Ces indices sont appelés des **clés**, et on leur associe des **valeurs** ;
- On peut les déclarer avec la syntaxe suivante :

```
clients = {
    "Jean Dupont": "19/01/1985",
    "Anne Dubois": "27/04/1966"
}
```

clés

valeurs

Dictionnaires : exemple d'utilisation

Exemple

```
>>> D = dict() # dictionnaire vide
>>> D['toto'] = 4 # associe 4 a la cle 'toto'
>>> D['titi'] = 6
>>> D['toto']
4
>>> D['toto'] = 'bonjour' # remplace 4
>>> print(D)
{'titi': 6, 'toto': 'bonjour'}
```

La clé doit être un élément **non modifiable**

Dictionnaires : accès aux éléments

- Les méthodes suivantes du dictionnaire `dico` permettent d'accéder :
  1. aux clés : `dico.keys()` ;
  2. aux valeurs : `dico.values()` ;
  3. aux deux : `dico.items()` ;

Exemple

```
>>> dico = {"une clé": "une valeur",
            "une autre clé": "une autre valeur"}
>>> dico.keys()
dict_keys(['une clé', 'une autre clé'])
>>> dico.values()
dict_values(['une valeur', 'une autre valeur'])
>>> dico.items()
dict_items([('une clé', 'une valeur'),
            ('une autre clé', 'une autre valeur')])
```

- Comme toujours, on peut transtyper le résultat ;

Opérations sur les dictionnaires

Soit `D` un dictionnaire. On peut demander si `x` est une :

- clé, avec `x in D` : $O(1)$
- valeur, avec `x in D.values()` ; $O(n)$

On peut supprimer des éléments de deux façons :

1. `D.pop(x)` : renvoie `D[x]` et supprime `(x, D[x])` ;
2. `D.popitem()` : renvoie et supprime le dernier couple `(clé, valeur)` ajouté à `D` ;

Modifiables et non-modifiables

On a vu jusqu'ici des objets de deux catégories :

Types non-modifiables

`bool`, `int`, `float`, `str`, `tuple`, `frozenset`

Types modifiables

`list`, `set`, `dict`

Pour passer d'un type à l'autre, on peut par exemple utiliser la fonction `tuple()` pour transformer une liste (**modifiable**) en un tuple (**non-modifiable**)

Pour les dictionnaires comme pour les listes : on a besoin des `[]` pour modifier les éléments dans un **`for`**.

Modifiables et non-modifiables

- Quand utiliser une structure modifiable, et quand utiliser une structure non-modifiable ?
- Ça dépend de vos besoins et de vos contraintes ;
- Il arrive qu'on n'ait pas le choix (éléments d'un `set`, clés d'un dictionnaire, . . .) ;
- **Attention :** la discussion de la fois passée concernant l'effet d'une fonction sur un paramètre modifiable s'applique aux itérables modifiables vus aujourd'hui !

Itérer sur des structures plus complexes

- Tirez parti de la syntaxe agréable du **`for`** pour rendre votre code plus lisible ;
- Par exemple, si vous encodez un polygone à l'aide d'une liste de points, n'écrivez pas :

Exemple

```
print("Mon polygone contient les points suivants: ")
for point in liste_points:
    print("x =", point[0], ", y =", point[1])
```

- Écrivez plutôt :

Exemple

```
print("Mon polygone contient les points suivants: ")
for x, y in liste_points:
    print("x =", x, ", y =", y)
```

Itérer sur des dictionnaires

- On pourra ainsi itérer sur les dictionnaires de plusieurs manières :

Exemple

```
mon_dico = ... # (initialisation)
for cle in mon_dico:  # ou mon_dico.keys()
    print("cle =", cle, ", valeur =", mon_dico[cle])
for cle, valeur in mon_dico.items():
    print("cle =", cle, ", valeur =", valeur)
```

- La règle est la même que pour les listes : on n'a besoin des `[]` que pour modifier les éléments ;

Attention aux dictionnaires vides

- Python 3 nous autorise à créer des dictionnaires vides avec `{}` ;
- **Ceci est déconseillé** : en effet, `{1}` est un ensemble contenant 1, tandis que `{}` n'est pas un ensemble vide mais un dictionnaire vide ;
- Dès lors, le test `ensemble == {}` ne marchera pas pour vérifier si un ensemble donné est vide ;
- Cette notation étant source de confusion, il est donc préférable de l'éviter ;