Introduction à la Programmation des Algorithmes

5.2. Python – Structures de données

François Fleuret
https://fleuret.org/11x001/

Certains des exemples que nous verrons dans les slides sont obtenus à l'aide d'une session interactive. Les lignes commençant par >>> sont celles tapées par l'utilisateur et les autres sont les affichages de l'interpréteur.

```
>>> a = 3
>>> a += 5
>>> a
8
>>> print(a * 10)
80
```

Python est un langage **orienté objets**, ce qui veut dire entre autres que:

- chaque type dispose de fonctions qui lui sont propres que l'on appelle des **méthodes**,
- ces méthodes prennent toute un premier argument du type donné, auquel elles sont "appliquées."

Nous n'irons pas plus dans les détails de la programmation orientée objets. Pour nos usages, cela veut simplement dire que chaque type a des fonctions qui peuvent être appelées avec l'opérateur `.`

Par exemple la méthode `str.lower` permet de convertir une chaîne de caractères en minuscules.

Elle peut être appelée de la manière classique:

```
>>> s = 'Une Grande Maison'
>>> str.lower(s)
'une grande maison'
```

ou de façon équivalente avec:

```
>>> s = 'Une Grande Maison'
>>> s.lower()
'une grande maison'
```

La terminologie de la programmation orientée objet est spécifique pour des concepts très proches de ceux de la programmation traditionnelle.

En particulier:

- Une **classe** est un type,
- Un **objet** ou **instance d'une classe** est une variable,
- Une **méthode** est une des fonctions associées à une classe.

Structure de données et mutabilité

Python offre différents types de données qui permettent de regrouper plusieurs valeurs. Les principaux sont:

- les listes,
- les tuples,
- les ensembles, et
- les dictionnaires.

Une liste est une séquence de valeurs qui peuvent être de types différents.

On peut définir explicitement une liste à l'aide d'expressions séparées par des virgules et placées entre crochets, et on peut accéder à des éléments avec l'opérateur crochet de manière similaire aux chaînes de caractères.

```
>>> a = [ 'un', 2, 3, 'quatre', 5, 6, 'sept' ]
>>> a
['un', 2, 3, 'quatre', 5, 6, 'sept']
>>> a[0]
'un'
>>> a[1:]
[2, 3, 'quatre', 5, 6, 'sept']
>>> a[2:-1]
[3, 'quatre', 5, 6]
```

Contrairement à ce que nous avons vu en C, toutes les variables Python sont en réalité des références vers des structures en mémoire.

Python est assez bien fait pour que les types standards que nous avons vus jusque là se comportent comme attendu. On dit qu'ils sont **immutables**: si vous copiez A dans B et modifiez ensuite A, cela ne change pas B.

Mais la plupart des types complexes en Python sont **mutables**, c'est à dire qu'un tel objet peut être modifié. Dans ce cas toutes les variables qui y font référence subiront cette modification: si vous copiez A dans B et modifiez ensuite A, **cela change B.**

Bien que les chaînes de caractères et les listes aient des similarités, les premières sont immutables alors que les secondes sont mutables.

```
>>> a = 'toto'
>>> b = a
>>> a += ' et titi'
>>> a
'toto et titi'
>>> b
'toto'
```

```
>>> a = [ 1, 2, 3 ]
>>> b = a
>>> a += [ 4, 5 ]
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
```

Donc comme les chaînes de caractères sont immutables Python comprend

```
a += 'blah'
```

comme “crée un nouvelle chaîne de caractères avec le contenu de celle référencée par a suivi de 'blah', et change a pour qu'elle fasse référence à cette nouvelle chaîne”, alors que comme les listes sont mutables

```
l += [ 4, 40, 10 ]
```

est compris comme “modifie la liste référencée par `l` en lui rajoutant `[ 4, 40, 10 ]` à la fin.”

Il peut arriver que l'on veuille avoir un nouvel objet mutable ayant le même contenu qu'un objet existant. Il suffit de faire une copie:

```
>>> a = [ 'truc' ]
>>> b = a
>>> c = a.copy()
>>> a += [ 'machin' ]
>>> a
['truc', 'machin']
>>> b
['truc', 'machin']
>>> c
['truc']
```

Attention au cas où un objet contient lui même des références vers d'autres objets mutables. La copie que nous venons de voir ne fait pas de "copie profonde".

```
>>> l = [ [ 1, 2 ], [ 3 ] ]
>>> l
[[1, 2], [3]]
>>> m = l.copy()
>>> m
[[1, 2], [3]]
>>> l[0] = 'truc'
>>> l
['truc', [3]]
>>> m
[[1, 2], [3]]
>>> l[1] += [ 'un' ]
>>> l
['truc', [3, 'un']]
>>> m
[[1, 2], [3, 'un']]
```


Le nombre d'éléments d'une liste peut être obtenu avec `len`, et les listes disposent de nombreuses méthodes, dont quelques unes sont:

- `list.append(x)`
  Ajoute un élément à la fin de la liste.
- `list.insert(i, x)`
  Insère l'élément `x` à la position `i`.
- `list.remove(x)`
  Supprime de la liste le premier élément dont la valeur est égale à `x`.
- `list.clear()`
  Supprime tous les éléments de la liste.
- `list.reverse()`
  Inverse l'ordre des éléments dans la liste.

De plus, une sous-partie de liste spécifiée avec les opérateurs `[]` et `:` peut être l'opérande de gauche d'une affectation.

```
>>> l = [ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
>>> len(l)
9
>>> l.append(9)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l.insert(4, 40)
>>> l
[0, 1, 2, 3, 40, 4, 5, 6, 7, 8, 9]
>>> l.remove(40)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l.reverse()
>>> l
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> l[5] = 111
>>> l
[9, 8, 7, 6, 5, 111, 3, 2, 1, 0]
>>> l[1:4] = [ 'a', 'b' ]
>>> l
[9, 'a', 'b', 5, 111, 3, 2, 1, 0]
```


L'instruction `del` permet de supprimer des éléments d'une liste:

```
>>> a = [ 0, 1, 2, 3, 4, 5 ]
>>> a
[0, 1, 2, 3, 4, 5]
>>> del a[2]
>>> a
[0, 1, 3, 4, 5]
>>> del a[1:3]
>>> a
[0, 4, 5]
```

Une liste est itérable élément par élément:

```
>>> for m in [ 'premier', 'second', 'troisieme' ]:
...     print(m)
...
premier
second
troisieme
```

Et l'instruction in permet de tester si une valeur est dans une liste:

```
>>> a = [ 2, 3, 4, 3, 2 ]
>>> 2 in a
True
>>> 1 in a
False
```


Une liste peut être utilisée comme une pile grâce à append qui rajoute un élément à la fin et pop qui enlève le dernier élément de la liste et le retourne:

```
>>> p = [ ]
>>> p.append('Paris')
>>> p
['Paris']
>>> p.append('Londres')
>>> p.append('Tokyo')
>>> p
['Paris', 'Londres', 'Tokyo']
>>> p.pop()
'Tokyo'
>>> p
['Paris', 'Londres']
>>> p.append('Berlin')
>>> p
['Paris', 'Londres', 'Berlin']
>>> p.pop()
'Berlin'
>>> p.pop()
'Londres'
>>> p
['Paris']
```

Tuples

Un **tuple** est un n-uplet de valeurs. Il est **immutable** et généralement assez court.

Il se note avec des valeurs séparées par des virgules, optionnellement entre parenthèses, et les opérateurs `[:]` fonctionnent comme sur les listes.

```
>>> a = ( 'un', 2, 3.0 )
>>> a
('un', 2, 3.0)
>>> len(a)
3
>>> b = 'un', 2, 3.0
>>> b
('un', 2, 3.0)
>>> s = 150,
>>> s
(150,)
>>> a[0]
'un'
>>> b[1:3]
(2, 3.0)
```

Comme indiqué, et contrairement aux listes, les tuples ne sont pas mutables.

```
>>> l = [ 'un', 2, 3.0 ]
>>> l
['un', 2, 3.0]
>>> l[2] = 'deux'
>>> l
['un', 2, 'deux']
>>> t = 'un', 2, 3.0
>>> t
('un', 2, 3.0)
>>> t[2] = 'deux'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```


Une spécificité très pratique du langage Python est que son opérateur d'affectation accepte comme opérande gauche une séquence d'identifiants séparés par des virgules si l'opérande droite est un objet itérable (chaîne de caractères, liste, tuple, etc.) avec un nombre d'éléments correspondant.

```
>>> a, b = ( 1, 2 )
>>> a
1
>>> b
2
>>> x = [ 'Bob', 35, 'Berlin' ]
>>> nom, age, ville = x
>>> nom
'Bob'
>>> age
35
>>> ville
'Berlin'
>>> a, b, c = ( 1, 2, 3, 4, 5 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
```

Et on peut faire de même avec les variables d'une boucle for:

```
>>> for a, b in [ (1, -1), [3, 4], ["pin", "pon" ] ]: print(a + b)
...
0
7
pinpon
```


Comme nous l'avons vu, les conversions explicites de types se font avec des fonctions associées à chaque type. Cela vaut pour les séquences.

```
>>> a = '123'
>>> list(a)
['1', '2', '3']
>>> tuple(a)
('1', '2', '3')
>>> l = [ 'le', 'petit', 'chat' ]
>>> tuple(l)
('le', 'petit', 'chat')
```

Ensembles

Les **ensembles** permettent de regrouper des valeurs de manière non ordonnée, donc sans élément dupliqué. On peut noter explicitement un ensemble avec {} ou en créer un avec `set`.

```
>>> { 2, 3, 5, 7 }
{2, 3, 5, 7}
>>> set('abracadabra')
{'d', 'b', 'r', 'a', 'c'}
>>> a = set([ 12, 35, 56, 17, 12, 12, 56, 35, 17, 12 ])
>>> a
{56, 17, 35, 12}
>>> 35 in a
True
```

```
>>> f = { 'Genève', 'Lausanne', 'Neuchâtel', 'Berne', 'Zurich' }
>>> g = { 'Berne', 'Zurich', 'Bâle', 'Fribourg' }
>>> f - g
{'Lausanne', 'Genève', 'Neuchâtel'}
>>> f | g
{'Fribourg', 'Genève', 'Neuchâtel', 'Berne', 'Zurich', 'Bâle', 'Lausanne'}
>>> f & g
{'Zurich', 'Berne'}
>>> 'Berne' in g
True
```


Dictionnaires

La dernière structure de donnée native en Python, et la plus puissante, est le **dictionnaire** qui permet d'associer des **valeurs** à des **clés**.

On peut noter explicitement un dictionnaire avec `{}` en indiquant les paires clé / valeur avec `:`. Les clés doivent être de types immutables.

```
>>> prix = { 'pomme': 1.25, 'orange': 1.8 }
>>> prix['pomme']
1.25
>>> prix['poire'] = 1.3
>>> prix
{'pomme': 1.25, 'orange': 1.8, 'poire': 1.3}
>>> del prix['pomme']
>>> prix
{'orange': 1.8, 'poire': 1.3}
>>> prix['orange'] = 1.75
>>> prix
{'orange': 1.75, 'poire': 1.3}
```


Un dictionnaire peut être créé à partir d'une liste de paires clé / valeur avec `dict`. Et il est possible de faire une boucle sur les paires clé / valeur à l'aide de la méthode `items`.

```
>>> d = dict([ (1, 'un'), (2, 'deux'), (3, 'trois') ])
>>> d
{1: 'un', 2: 'deux', 3: 'trois'}
>>> for k, v in d.items():
...     print(k, v)
...
1 un
2 deux
3 trois
```

Finalement il est possible de récupérer la liste des clés et des valeurs à l'aide des méthodes `keys` et `values`.

Les types de retour de ces méthodes ne sont pas des listes à proprement parler, mais ils peuvent être convertis explicitement en séquences:

```
1  >>> prix = { 'pomme': 1.25, 'orange': 1.8, 'poire': 1.3 }
2  >>> list(prix.keys())
3  ['pomme', 'orange', 'poire']
4  >>> list(prix.values())
5  [1.25, 1.8, 1.3]
```