Structures de données

Table des matières

1 Méthodes

Python est un langage *orienté objet*. Dans la programmation orientée objet, on manipule des objets à l'aide de *méthodes*.

Définition 1.1

Une méthode est une fonction associée à une classe d'objets.

La syntaxe est :

Python
```
objet.methode(arg1, arg2, ...)
```

On peut voir une méthode comme une fonction prenant l'objet comme argument supplémentaire :

Python
```
methode(objet, arg1, arg2, ...)
```

Illustrons cela par quelques exemples de méthodes associées aux chaînes de caractères. Nous en verrons d'autres plus loin.

Python
```
In [1]: nom = "Pécheux"

In [2]: nom.endswith('x')
Out[2]: True

In [3]: nom.startswith('A')
Out[3]: False
```

Python
```
In [4]: nom.upper()
Out[4]: 'PÉCHEUX'

In [5]: nom.lower()
Out[5]: 'pécheux'

In [6]: nom.swapcase()
Out[6]: 'pÉCHEUX'
```

2 Collections

En informatique, on a très souvent besoin de regrouper des objets, et donc de manipuler des *collections* d'objets. En organisant d'une certaine manière les données, on permet un traitement automatique de ces dernières plus efficace et rapide.

Une collection d'objets (ou *éléments*) peut être, ou non :

- *ordonnée* : les éléments sont rangés dans la collection dans un ordre bien défini ;
- *indexable* : on a un moyen d'accéder directement à un élément, en général, si la collection est ordonnée, via un *indice* ;
- *modifiable* : on peut ajouter, supprimer ou modifier des éléments ;
- *itérable* : on peut parcourir les éléments de la collection.

En fonction des besoins, une ou plusieurs de ces propriétés peuvent être importantes. Il y a donc de nombreuses manières de créer des collections en fonction des propriétés voulues. Python propose de très nombreuses collections [1]. Dans ce cours, nous allons présenter les structures de données :

- *tuple* (type `tuple`), que nous avons déjà rencontré, une collection ordonnée et non modifiable d'objets ;
- *chaîne de caractères* (type `str`), une collection ordonnée et non modifiable de caractères ;
- *liste* (type `list`), une collection ordonnée mais modifiable d'objets ;
- *ensemble* (type `set`), une collection non ordonnée, modifiable, qui correspond à la notion mathématique de même nom.

Seules les chaînes de caractères et les listes sont explicitement au programme, mais on ne saurait se passer des tuples et les ensembles sont souvent très utiles pour les projets et les TIPE. Il existe aussi les dictionnaires (type `dict`) que nous étudierons peut-être plus tard.

À tous ces types est associée une fonction qui porte le même nom que le type et qui permet de convertir vers ce type (cf. section 6). En particulier, on peut créer un tuple vide avec l'appel `tuple()`, une chaîne de caractères vide avec `str()`, une liste vide avec `list()` et un ensemble vide avec `set()`.

3 Tuples, chaînes de caractères, listes et ensembles

3.1 Tuples

Définition 3.1

Un tuple est une collection ordonnée et immuable (non modifiable) d'éléments.

1. Il y a même un module qui porte ce nom.

Nous avons déjà vu comment construire et déconstruire des tuples en Python.

Python
```
In [7]: t = ("MPSI", 3)

In [8]: t
Out[8]: ("MPSI", 3)

In [9]: filiere, classe = t
```

3.1.1 Cas particulier : $n = 0$ ou 1

Il n'existe qu'un 0-uplet, c'est celui qui ne contient rien [2], il s'écrit `()`. On peut aussi l'obtenir avec `tuple()`. C'est une valeur un peu étrange dont l'utilité est limitée.

Un 1-uplet ne contenant que la valeur `v` s'écrit `(v,)`. Ici, la virgule finale est essentielle car c'est elle qui distingue le 1-uplet `(v,)` de l'expression `(v)` dont la valeur est `v`.

Python
```
In [10]: () == tuple()
Out[10]: True

In [11]: type(())
Out[11]: tuple
```

Python
```
In [12]: type((3))
Out[12]: int

In [13]: type((3,))
Out[13]: tuple
```

3.2 Chaîne de caractères

Définition 3.2

Une chaîne de caractères (*string* en anglais) est une collection ordonnée et modifiable d'objets particuliers : les caractères.

Les caractères peuvent désigner des lettres de l'alphabet, des symboles comme les signes de ponctuation ou des caractères spéciaux comme le caractère `'\n '` (retour à la ligne) ou le caractère `'\t '` (tabulation).

2. Vraiment rien, `(None,)` est un 1-uplet.

Une chaîne de caractères est une suite finie de caractères consécutifs et se note indifféremment[3] entre apostrophes « ' » ou guillemets « " ». La chaîne de caractères vide s'obtient en écrivant `''`, `""` ou `str()`. En Python, un caractère n'est rien d'autre qu'une chaîne de caractères de longueur unitaire.

Python
```
In [14]: 'a'
Out[14]: 'a'

In [15]: '\n'
Out[15]: '\n'
```

Python
```
In [16]: 'MPSI-3'
Out[16]: 'MPSI-3'

In [17]: "Aujourd'hui"
Out[17]: "Aujourd'hui"
```

☞ **Remarque 1**

On pourrait représenter une chaîne de caractères par un tuple de caractères, par exemple utiliser `('B', 'o', 'n', 'j', 'o', 'u', 'r')` au lieu de `"Bonjour"`. S'il existe un type particulier pour les chaînes de caractères, c'est que leur usage est constant et que de nombreuses opérations spécifiques sont disponibles, comme nous le verrons.

3.2.1 Conversions vers les types simples

On peut convertir une valeur d'un type simple vers une chaîne de caractères à l'aide de la construction `str(v)`. La chaîne obtenue est la même que celle que Python affiche par évaluation de `v`. Inversement, on peut utiliser les fonctions de conversion qui portent le nom du type.

Python
```
In [18]: str(42)
Out[18]: '42'

In [19]: str(False)
Out[19]: 'False'

In [20]: str(4.2)
Out[20]: '4.2'
```

Python
```
In [21]: int("42")
Out[21]: 42

In [22]: float("4.2")
Out[22]: 4.2

In [23]: bool("True")
Out[23]: True
```

3. Le choix peut être imposé par le contenu : une chaîne contenant un apostrophe sera encadrée par des guillemets doubles, et inversement.

3.2.2 Formatage d'une chaîne de caractères

La méthode `format` permet de créer lisiblement des chaînes de caractères qui dépendent de valeurs. Il suffit de mettre des accolades « `{}` » dans la chaîne de caractères qui seront automatiquement remplacées par les valeurs données en arguments.

Python
```
In [24]: "Bonjour aux {}-{} !".format("MPSI", 3)
Out[24]: 'Bonjour aux MPSI-3 !'

In [25]: "La réponse est {0}, mais {1} n'est pas {0}".format(
    ...:   42, 43)
Out[25]: 'La réponse est 42 et non 43. Répondez 42 !'
```

Il y a d'innombrables possibilités[4] de formatage des arguments, que l'on place dans les accolades après un deux-points. Le format `:xf` avec x un flottant et `f` pour flottant, permet de spécifier la précision souhaitée de l'affichage.

Python
```
In [26]: "Pi n'est ni {0:.0f}, ni {0:.20f}".format(math.pi)
Out[26]: "Pi n'est ni 3, ni 3.14159265358979311600"
```

3.3 Listes

Définition 3.3

Une liste est une collection ordonnée et modifiable d'éléments.

On construit une liste comme un tuple, mais avec des crochets au lieu des parenthèses. Une liste vide s'obtient avec `[]` ou `list()`.

Python
```
In [27]: []
Out[27]: []

In [28]: list()
Out[28]: []
```

Python
```
In [29]: [0, 1, 2]
Out[29]: [0, 1, 2]

In [30]: ["foo", "bar"]
Out[30]: ['foo', 'bar']
```

4. Ne pas hésiter à aller se document sur le web pour les besoins de vos TIPE.

On peut ajouter ou supprimer un élément à la fin de la liste avec les méthodes `append` et `pop`.

PYTHON
```
In [31]: L = [0, 1, 2]

In [32]: L.append(3)

In [33]: L
Out[33]: [0, 1, 2, 3]
```

PYTHON
```
In [34]: L.pop()
Out[34]: 3

In [35]: L
Out[35]: [0, 1, 2]
```

Nous étudierons d'autres opérations sur les listes dans les sections suivantes et la semaine prochaine.

3.4 Ensembles

Définition 3.4

Un ensemble est une collection non ordonnée modifiable d'éléments, sans répétitions.

On utilise les accolades au lieu des crochets ou des parenthèses, comme en mathématiques.

PYTHON
```
In [36]: {0, 1, 2}
Out[36]: {0, 1, 2}

In [37]: {"a", 0, 0, "a"}
Out[37]: {0, 'a'}
```

PYTHON
```
In [38]: {0, 1} == {1, 0}
Out[38]: True

In [39]: {0, 1} == [0, 1]
Out[39]: False
```

⚠ Un ensemble vide s'obtient avec `set()` mais pas avec `{}` qui désigne en fait un dictionnaire vide.

PYTHON
```
In [40]: type(set())
Out[40]: set
```

PYTHON
```
In [41]: type({})
Out[41]: dict
```

On peut ajouter ou supprimer un élément avec les méthodes `add` et `remove`.

PYTHON
```
In [42]: ens = {0, "a", 3.2}

In [43]: ens = {0, 1, 2}

In [44]: ens.add(0)

In [45]: ens.add(3)
```

PYTHON
```
In [46]: ens
Out[46]: {0, 3, 1, 2}

In [47]: ens.remove(2)

In [48]: ens
Out[48]: {0, 1, 3}
```

4 Opérations sur les collections

Dans toute cette section, on suppose définies les variables suivantes qui nous serviront d'exemples.

PYTHON
```
point = (1., 2.)
nom = "Pécheux"
notes = [12, 14, 8, 19, 11]
fruits = {'pêche', 'pomme', 'poire'}
```

4.1 Longueur

On obtient la longueur, c'est-à-dire le nombre d'éléments, d'une collection à l'aide de la fonction `len` :

PYTHON
```
In [49]: len(point)
Out[49]: 2

In [50]: len(())
Out[50]: 0

In [51]: len((42,))
Out[51]: 1
```

PYTHON
```
In [52]: len(notes)
Out[52]: 5

In [53]: len([])
Out[53]: 0

In [54]: len([[]])
Out[54]: 1
```

PYTHON
```
In [55]: len(nom)
Out[55]: 7

In [56]: len("")
Out[56]: 0

In [57]: len("a")
Out[57]: 1
```

PYTHON
```
In [58]: len(fruits)
Out[58]: 3

In [59]: len(set())
Out[59]: 0

In [60]: len({()})
Out[60]: 1
```

4.2 Test d'appartenance

Il est possible de tester si une valeur appartient à une collection à l'aide de l'opérateur `in` :

PYTHON
```
In [61]: 1. in point
Out[61]: True

In [62]: 'a' in nom
Out[62]: False
```

PYTHON
```
In [63]: 8 in notes
Out[63]: True

In [64]: "banane" in fruits
Out[64]: False
```

Dans le cas des chaînes de caractères on peut même tester directement l'appartenance d'une sous-chaîne avec cette même construction :

PYTHON
```
In [65]: "la MP" in "Après la MPSI"
Out[65]: True

In [66]: "la PC" in "Après la MPSI"
Out[66]: False

In [67]: "si" in "Après la MPSI".lower()
Out[67]: True
```

4.3 Concaténation

Pour les collections ordonnées, il est possible de « coller » deux collections à la suite pour obtenir la collections comportant tous les éléments dans cet ordre. On parle de *concaténation*. L'opérateur correspondant en PYTHON est l'opérateur « + », le même symbole que pour l'addition.

PYTHON
```
In [68]: (0, 1) + (2, 3, 4)
Out[68]: (0, 1, 2, 3, 4)

In [69]: "MPSI" + "-" + "3"
Out[69]: 'MPSI-3'

In [70]: notes + [0]
Out[70]: [12, 14, 8, 19, 11, 0]
```

Si on veut ajouter un seul élément à un tuple, il faut faire attention à bien considérer un 1-uplet.

PYTHON
```
In [71]: (0, 1) + 3
TypeError: can only concatenate tuple (not "int") to tuple

In [72]: (0, 1) + (3,)
Out[72]: (0, 1, 3)
```

Mais... on avait dit que les tuples n'étaient pas modifiables. Comment a-t-on pu alors ajouter un élément à un tuple ? En fait, on a créé un *nouveau* tuple, résultant de la concaténation de deux tuples.

✔ Lorsque l'on concatène deux collections, on créé une *nouvelle* collection.

Puisque l'on peut ajouter, on peut multiplier, avec la règle :

$$\text{obj} \times n = \overbrace{\text{obj} + \text{obj} + \cdots + \text{obj}}^{n \text{ fois}}$$

Python
```
In [73]: point * 2
Out[73]: (1.0, 2.0, 1.0, 2.0)

In [74]: nom * 1 + nom * 0
Out[74]: 'Pécheux'

In [75]: [0] * 10
Out[75]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Les ensembles ne sont pas ordonnés et ne peuvent pas se concaténer

Python
```
In [76]: fruits + {"banane"}
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

En revanche, on dispose des opérateurs d'union, d'intersection, de différence, de différence symétrique et bien d'autres.

Python
```
In [77]: fruits | {"banane"}
Out[77]: {'banane', 'poire', 'pomme', 'pêche'}

In [78]: fruits & {"poire", "fraise"}
Out[78]: {'poire'}

In [79]: fruits - {"pomme", "poire"}
Out[79]: {'pêche'}
```

4.4 Accès aux composantes

Pour les collections indexables, on peut directement accéder à une composante d'une collection `foo` en utilisant la syntaxe `foo[i]` où `i` est l'*indice* de la composante dans la collection `foo`.

⚠ En informatique, les indices commencent à partir de 0 et non à partir de 1.

Python
```
In [80]: point[0]
Out[80]: 1.0

In [81]: point[1]
Out[81]: 2.0

In [82]: point[2]
IndexError: tuple index out of range

In [83]: nom[0]
Out[83]: 'P'

In [84]: notes[2]  # 3ième note
Out[84]: 8

In [85]: fruits[1]
TypeError: 'set' object does not support indexing
```

Les listes sont modifiables, on peut changer le contenu de la case d'indice *i* en lui affectant une autre valeur :

Python
```
In [86]: notes
Out[86]: [12, 14, 8, 19, 11]

In [87]: notes[2] = 20

In [88]: notes
Out[88]: [12, 14, 20, 19, 11]
```

En revanche, les tuples et les chaînes de caractères sont *immuables*. Il n'est pas possible d'affecter de nouvelles valeurs aux composantes.

Python
```
In [89]: point[0] = -2.
TypeError: 'tuple' object does not support item assignment

In [90]: nom[1] = 'e'
TypeError: 'str' object does not support item assignment
```

5 Itération

5.1 Itérables et `for`

Définition 5.1

Un *itérable* est un objet dont on peut parcourir les valeurs. Les collections sont itérables.

On peut parcourir les éléments d'un itérable avec la construction :

Python
```
for element in iterable:
    # bloc dans lequel
    # on dispose de
    # 'element'
```

Comme pour les fonctions et les tests, on remarque que le corps de l'itération est introduit par un deux-points et est convenablement indenté.

Python
```
In [91]: for coordonnee in point:
    ...:     print(coordonnee)
1.0
2.0

In [92]: for note in notes:
    ...:     print("{:02}/20".format(note), end=" ")
12/20 14/20 20/20 19/20 11/20

In [93]: for lettre in nom:
    ...:     if lettre.isupper():
    ...:         print(lettre)
P

In [94]: for fruit in fruits:
    ...:     print(fruit, end=" ; ")
pomme ; poire ; pêche ;
```

⚠

Dans le cas d'un ensemble, *on ne sait pas* dans quel ordre vont être itérés les éléments. C'est souvent le même, mais cela peut changer à tout moment.

5.2 Exercices classiques ♥

Exercice 1 (*Somme des éléments d'une liste* ♥) Écrire une fonction calculant la somme des éléments d'une liste d'entiers.

La fonction suivante calcule la somme des éléments de n'importe quel itérable de nombres.

Python
```
def somme(iterable):
    """Somme des éléments de l'itérable."""
    # Somme jusqu'à présent
    somme_partielle = 0
    for element in iterable:
        # Mise à jour de la somme en prenant en compte l'élément
        somme_partielle += element
    return somme_partielle
```

☞ **Remarque 2**

Cette fonction existe en Python et s'appelle `sum`. Essayer `help(sum)`. Vous pouvez l'utiliser, sauf s'il vous semble que l'on demande de l'implémenter soi-même.

Exercice 2 (*Somme des éléments vérifiant un prédicat* ♥) Écrire une fonction calculant la somme des éléments (strictement) positifs d'une liste d'entiers.

Python
```
def somme_elements_positifs(iterable):
    """Somme des éléments positifs de l'itérable."""
    somme_partielle = 0
    for element in iterable:
        if element > 0:
            # Ne prendre en compte que les éléments qui satisfont
            # le prédicat demandé.
            somme_partielle += element
    return somme_partielle
```

Exercice 3 (*Nombre d'éléments vérifiant un prédicat* ♥) Écrire une fonction calculant le nombre d'espaces d'une chaîne de caractères.

Python
```python
def nombre_espaces(chaine):
    """Nombre d'espaces dans une chaîne de caratères."""
    # On définit un compteur
    compteur = 0
    for lettre in chaine:
        if lettre == ' ':
            # Ne compter que les espaces
            compteur += 1
    return compteur
```

5.3 L'itérateur `range` ♥

La fonction `range` renvoie un itérable (de type `range`) un peu particulier qui permet de définir rapidement une progression arithmétique.

La fonction `range` prend trois arguments entiers : `range(debut, fin, pas)` énumère les entiers de `debut` (inclus) à `fin` (exclus) avec un pas de `pas`. Sans le troisième argument, `range(debut, fin)`, le pas est de 1 par défaut. Avec un seul argument, `range(fin)`, le début est 0 par défaut.

Python
```
In [95]: for i in range(10):
    ...:     print(i, end=' ')
0 1 2 3 4 5 6 7 8 9

In [96]: for i in range(5, 11):
    ...:     print(i, end=' ')
5 6 7 8 9 10

In [97]: for i in range(1, 10, 3):
    ...:     print(i, end=' ')
1 4 7
```

Exercice 4 (*Somme des entiers consécutifs* ♥) Écrire une fonction calculant $S_n = \sum_{k=0}^{n} k$.

Python
```python
def somme_entiers_consecutifs(n):
    """Somme des entiers de `0` à `n` (inclus)."""
    somme = 0
    for i in range(1, n + 1):
        somme += i
    return somme
```

⚠
Il faut bien faire attention aux bornes dans le `range`. Ici, on veut aller jusqu'à n inclus, donc il faut une fin (exclue) à $n+1$.

☞ **Remarque 3**

Pour les collections itérables, on en déduit une autre manière d'énumérer la collection. Comparer les deux fonctions suivantes, qui produisent le même résultat.

Python
```python
def epeler(mot):
    """Imprime les lettres d'un mot."""
    for lettre in mot:
        print(lettre, end=" ")
```

Python
```python
def epeler(mot):
    """Imprime les lettres d'un mot."""
    for i in range(len(mot)):
        print(mot[i], end=" ")
```

✔
Il faut savoir jongler avec les deux versions.

Exercice 5 (*Liste des carrés* ♥) Écrire une fonction qui renvoie la liste des carrés des entiers entre 1 et n.

Python
```
def liste_carres(n):
    """Liste des carrés de ‘1‘ à ‘n‘ (inclus)."""
    # On initialise une liste vide qui va contenir les carrés
    liste = []
    for i in range(1, n + 1):
        # On met à jour la liste avec le carré du nombre en cours
        liste.append(i**2)
    return liste
```

5.4 Compléments : autres itérateurs

Il est parfois utile de connaître à la fois l'indice et la valeur d'un élément d'une collection indexable.

Python
```
def affiche_notes(notes):
    """Affiche les notes d'une classe."""
    for i in range(len(notes)):
        print("L'élève {} a eu {:02}/20.".format(i, notes[i]))
```

Python
```
In [98]: affiche_notes(notes)
L'élève 0 a eu 12/20.
L'élève 1 a eu 14/20.
L'élève 2 a eu 20/20.
L'élève 3 a eu 19/20.
L'élève 4 a eu 11/20.
```

On dispose en Python d'une construction élégante avec l'itérateur `enumerate` qui énumère les couples (indice, élément) d'une collection indexable.

Python
```
def affiche_notes(notes):
    """Affiche les notes d'une classe."""
    for i, note in enumerate(notes):
        print("L'élève {} a eu {:02}/20.".format(i, note))
```

Enfin, il est possible d'énumérer deux itérables en parallèle à l'aide de l'itérateur `zip` (penser à une fermeture éclair qui « zippe » les deux collections).

Python
```
In [99]: prenoms = ("Nicolas", "Jérémy", "Simon")

In [100]: noms = ("Pécheux", "Larochette", "Rognerud")

In [101]: for prenom, nom in zip(prenoms, noms):
    ...:     print(prenom, nom)
Nicolas Pécheux
Jérémy Larochette
Simon Rognerud
```

6 Conversions

Les fonctions `tuple`, `list` et `set` peuvent prendre en argument un itérable et créer une *nouvelle* collection de ce type à partir des éléments de l'itérable.

Python
```
In [102]: list(point)
Out[102]: [1.0, 2.0]

In [103]: list(fruits)
Out[103]: ['poire', 'pêche']
```

Python
```
In [104]: tuple("foo")
Out[104]: ('f', 'o', 'o')

In [105]: set("foo")
Out[105]: {'f', 'o'}
```

6.1 Copie d'une collection

On en déduit un moyen de copier une collection, puisque ces fonctions renvoient de *nouvelles* collections.

Python
```
In [106]: notes_copy = list(notes)

In [107]: notes_copy
Out[107]: [12, 14, 20, 19, 11]

In [108]: notes[0] = 1

In [109]: notes
Out[109]: [1, 14, 20, 19, 11]

In [110]: notes_copy
Out[110]: [12, 14, 20, 19, 11]
```

⚠ L'affectation `notes_copy = notes` *ne copierait pas* la liste. On en reparlera prochainement.

6.2 Supprimer les doublons d'une liste

On en déduit un moyen très « pythonique » de supprimer les doublons d'une liste.

Python
```
In [111]: valeurs = [0, 1, 1, 0, 2, 1, 0, 1, 1]

In [112]: list(set(valeurs))
Out[112]: [0, 1, 2]
```

Cette astuce est très utile pour les TIPE, mais dans un sujet de concours, si c'est demandé, il faudra très probablement le faire « à la main ».

6.3 Liste des entiers de 0 à $n-1$ ♥

On en déduit aussi un moyen simple de créer une liste d'entiers en progression arithmétique, en transformant en liste un `range`.

Python
```
In [113]: list(range(10))
Out[113]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.4 `split` et `join`

Deux méthodes très utiles sur les chaînes de caractères sont les méthodes `split` et `join`.

La méthode `split` (« éclater » en anglais) permet de découper une chaîne de caractères selon un caractère donné (ou une sous-chaîne) et de renvoyer la liste des sous-chaînes ainsi obtenues. Sans arguments, elle découpe suivant les espaces, tabulations et retours à la ligne.

Python
```
In [114]: "La réponse est 42".split()
Out[114]: ['La', 'réponse', 'est', '42']

In [115]: "La réponse est 42".split('e')
Out[115]: ['La répons', ' ', 'st 42']
```

Inversement, la méthode `join` (« joindre » en anglais) permet d'insérer une sous-chaîne entre tous les éléments d'un itérable de chaînes de caractères donné en argument et de produire la chaîne de caractères résultante.

Python
```
In [116]: " ".join("toto")
Out[116]: 't o t o'

In [117]: "-".join("Pécheux")
Out[117]: 'P-é-c-h-e-u-x'

In [118]: " ".join(("Nicolas", "Pécheux"))
Out[118]: 'Nicolas Pécheux'

In [119]: " ".join("La réponse est 42.".split())
Out[119]: 'La réponse est 42.'
```