Chapitre 1
Typage en Python

1. Types de base

Python est un langage à typage dynamique, ce qui signifie qu'il n'est pas nécessaire de déclarer les variables avant de pouvoir leur affecter une valeur. La valeur que l'on affecte possède un **type** qui dépend de la nature des données (nombre entier, nombre à virgule, chaîne de caractères, etc.). Le type du contenu d'une variable peut donc changer si on change sa valeur par affectation directe.

On utilisera :

- **Le type int**
- **Le type float**
- **Le type str**
- **Le type bool**

Méthodes utiles :

Les méthodes `type()` et `id()`

2. Types construits

À partir des types de base, il est possible de façonner de nouveaux types de variables, appelés types construits. Ceux-ci ont chacun leurs particularités. Choisir le bon type de variable est essentiel pour implémenter un algorithme dans un langage de programmation. Les trois types p-uplet, tableau et dictionnaire présentés ici sont génériques et peuvent être mis en œuvre dans de nombreux langages de programmation. Ce chapitre utilise les appellations spécifiques du langage Python.

2.1 Le p-uplet, de type `tuple`

Définition

p-uplet : Un p-uplet (ou *tuple* en anglais) est une collection ordonnée d'éléments, appelés *composantes* ou *termes*. Chaque terme peut être de n'importe quel type.

Création d'un p-uplet

Pour créer un p-uplet non vide, on écrit n valeurs séparées par des virgules. Par exemple :

▷ t = "a", "b", "c", 3 pour un tuple à 4 éléments ;

▷ t = "a", pour un tuple à 1 éléments (attention à la virgule);

▷ t = ( ) pour un tuple à 0 éléments (ici, pas de virgule, mais des parenthèses).

Pour écrire un p-uplet qui contient un n-uplet et l'utilisation de parenthèses est nécessaire. Voici un exemple avec un tuple à 2 éléments dont le second est un tuple : t = 3, ("a", "b", "c"). En général, les parenthèses sont obligatoires dès que l'écriture d'un p-uplet est contenue dans une expression plus longue. Dans tous les cas, les parenthèses peuvent améliorer la lisibilité.

```
>>> t1 = "a", "b"
>>> t2 = "c", "d"
>>> t1 + t2
('a', 'b', 'c', 'd')
>>> 3 * t1
('a', 'b', 'a', 'b', 'a', 'b')
```

Appartenance

Pour tester l'appartenance d'un élément à un tuple, on utilise l'opérateur in :

```
>>> t = "a", "b", "c"
>>> "a" in t
True
>>> "d" in t
False
```

Affectation multiple

Prenons pour exemple l'affectation a, b, c = 1, 2, 3. Ceci signifie que le tuple (a, b, c) prend pour valeur le tuple (1, 2, 3) , autrement dit, les valeurs respectives des variables a , b et c sont 1 , 2 et 3 . En particulier, l'instruction a, b = b, a permet d'échanger les valeurs des deux variables a et b . Les valeurs des éléments d'un tuple peuvent ainsi être stockées dans des variables. Cette syntaxe s'utilise souvent avec une fonction qui renvoie un tuple.

```
>>> t = 1, 2, 3
>>> a, b, c = t
>>> b
2
```

2.2 La liste, de type `list`

Définition

Tableau : Un tableau est une collection ordonnée d'éléments de n'importe quel type, organisés séquentiellement (les uns à la suite des autres).

Les éléments d'une liste sont repérés par des indices 0, 1, 2, ...

La liste est la structure de donnée la plus utilisée en Python. Il faut en maîtriser les subtilités. En Python, une liste est une collection ordonnée d'objets séparés par des virgules, enfermés dans des crochets.

```
liste=['lundi',28,True,10.625]
>>>print(liste[1])          % On accède à un élément de la liste grâce à son indice
```

Méthodes utiles de la classe list

- len(L) renvoie l'entier 4 : longueur de la liste
- L.append('Samedi') ajoute l'objet 'Samedi' à la collection d'objets en dernière position.
- L.pop(i) : Enlève de la liste L l'élément situé à la position i.
- L.pop() enlève et retourne le dernier élément de la liste.

Parcours d'une liste

Python est un surdoué pour parcourir successivement tous les éléments d'une séquence donnée : ainsi l'instruction for est-elle l'instruction idéale pour parcourir une liste :

```
19 liste=['chien', 'chat','crocodile','poisson']
20 for animal in liste:
21     print('longueur de la chaîne', animal,'=', str(len(animal)))
```

renvoie :

longueur de la chaîne chien = 5

longueur de la chaîne chat = 4

longueur de la chaîne crocodile = 9

longueur de la chaîne poisson = 7

Python est le king of slice!

On peut accéder à une partie d'une liste avec la syntaxe suivante : **Liste[début : fin : pas]**

L'écriture d'un « : » peut signifier « on garde », on ne découpe pas. Par défaut, début et fin valent None et pas vaut 1. On garde toujours l'indice de début, on exclut l'indice de fin.

Soit une liste L constituée d'entiers :

```
>>> L
```

[0,1,2,3,4,5,6,7,8,9]

- On peut découper une partie au début de la liste :

  `>>>L[:2]` # on garde les éléments depuis le début jusqu'au deuxième indice, qui est exclu .

  [0, 1]

  Cette syntaxe est équivalente à L[None :2 :None]

- On peut découper la fin de la liste à partir de l'indice 2, que l'on garde :

  `>>>L[2 :]`

  [2, 3, 4, 5, 6, 7, 8, 9]

- On peut découper la liste dans le sens inverse, en utilisant le troisième paramètre fixé à −1. L'utilisation du troisième paramètre est permis depuis Python 1.4 et reste peu connu : c'est le découpage étendu ou extended slice.

  `>>> L[ : : -1]`

  [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

- On peut garder une partie de la liste parcourue à l'envers mais les indices sont comptés dans le sens normal :

  `>>> L[9 : 7 : -1]` # l'indice 9 est gardé , l'indice 7 est exclu .

  [9, 8]

- On peut parcourir une liste de 2 en 2, dans les deux sens :

  `>>> L[ : : 2]`

  [0, 2, 4, 6, 8]

  `>>> L[ : : -2]`

  [9, 7, 5, 3, 1]

Notons que les slices tolèrent le dépassement d'indice, à droite ou à gauche (avec un indice négatif) : les portions du slice qui « dépassent » à droite ou à gauche sont ignorées.

```
>>> L[2:100]
```

```
        [2, 3, 4, 5, 6, 7, 8, 9]
>>> L[-10 : 2 : 1]
        [0, 1]
```

Liste en compréhension (comprehension list)

En programmation informatique, la syntaxe de certains langages de programmation comme Python permet de définir des listes en compréhension, c'est-à-dire des listes dont le contenu est défini par filtrage du contenu d'une autre liste, selon un principe analogue à celui de la définition par compréhension de la théorie des ensembles.

Imaginons que l'on souhaite créer une liste contenant le carré des entiers de 0 à 10. Deux méthodes peuvent être proposées :

Voici la méthode classique :

```
1 carres=[]
2 for x in range(0,11):
3     carres.append(x**2)
4 print(carres)
```

qui renvoie [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Voici la méthode Pythonique : c'est très FOR!

```
6 carres=[x**2 for x in range(0,11)]
7 print(carres)
```

qui renvoie [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Filtrage de liste

C'est encore plus FOR avec un IF imbriqué (et ainsi de suite...). Imaginons que l'on souhaite former tous les couples différents de valeurs entières prises dans deux ensembles disjoints : par exemple (1,3) ou (1,4) ...

Voici la syntaxe du filtrage de liste :

[mapping-expression for element in liste if expression_de_Filtrage]

Voici la méthode classique :

```
 9 combinaisons = []
10 for x in [1,2,3]:
11     for y in [1,5,6]:
12         if x != y:
13             combinaisons.append((x, y))
14 print(combinaisons)
```

renvoie [(1, 5), (1, 6), (2, 1), (2, 5), (2, 6), (3, 1), (3, 5), (3, 6)]

Voici la méthode Pythonique, avec une liste en compréhension et un filtrage de liste : c'est super FOR !

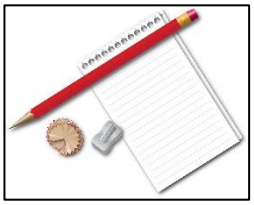
```
16 combinaisons = [(x,y) for x in[1,2,3] for y in [1,5,6] if x!=y]
17 print(combinaisons)
```

renvoie [(1, 5), (1, 6), (2, 1), (2, 5), (2, 6), (3, 1), (3, 5), (3, 6)]

Certains éléments ont été ajoutés alors que d'autres sont totalement ignorés. Cela revient à faire un filtrage sur une liste sans utiliser la méthode `filter( )`. Les listes en compréhension sont un outil très puissant de Python.

Ajoutons que la syntaxe est presque naturelle et que cela ressemble à une écriture mathématique :

$$D = \{ \ (x,y) \ | x \in \{1,2,3\}; y \in \{1,5,6\}; x \neq y; \}$$



Obtenir la liste des entiers compris entre 0 et 100, divisibles par 8 et non par 3.

Comment copier une liste ?

Pensez-vous qu'une instruction `copieListe=liste` permette de copier une liste ? Un test de comparaison avec l'opérateur `is` nous renseigne :

NB : `is` et `isnot` sont des opérateurs de type : `a is b` renvoie True si les variables a et b pointent vers le même objet.

```
23 liste=['chien','chat','crocodile','poisson']
24 copieListe=liste
25 print(copieListe is liste)
```

renvoie True, ce qui indique qu'on n'a rien copié du tout : liste et copie Liste font référence à un unique

objet ! Pouvez-vous proposer une manière efficace de copier une liste?

Indice : utiliser le slice.

Réponse : …

Il s'agit ici d'une *copie superficielle*. C'est-à-dire une copie au premier niveau. Si la liste copiée est une liste de liste, la copie ne sera pas complète (sous-entendu, la copie n'ira pas en profondeur comme pour une liste de liste de liste… etc.)

On peut alors utiliser la méthode *deepcopy* :

```
27 from copy import *
28 vraieCopie=copy.deepcopy(liste)
```

Cette copie profonde (appelée également *copie récursive*) est une opération très lourde, puisque l'on doit rentrer récursivement dans toutes les structures de données pour copier les valeurs individuelles successivement. Elle peut engendrer des appels de l'objet à lui-même et générer des erreurs ou ds boucles infinies.

2.3 Le dictionnaire, de type `dict`

Les éléments d'un dictionnaire sont des couples clé-valeur. Un dictionnaire est créé avec des accolades, les différents couples étant séparés par des virgules. La clé et la valeur correspondante d'un élément sont séparées par deux points.

Exemple : dico = {"A": 0, "B": 1, "C": 2, "D": 3}

On peut construire un dictionnaire en compréhension comme avec les listes :

```
>>> d = {chr(65+i): i for i in range(26)}
>>> d
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9, 'K':
   ↪ 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R': 17, 'S': 18, 'T':
   ↪  19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}
```

On peut convertir une liste de listes à deux éléments en dictionnaire avec la fonction dict .

```
>>> liste = [['A', 0], ['B', 1], ['C', 2]]
>>> d = dict(liste)
>>> d
{'A': 0, 'B': 1, 'C': 2}
```

Accès aux éléments

Pour accéder aux clés ou aux valeurs, nous avons les méthodes keys et values

```
>>> d = {'A': 0, 'B': 1, 'C': 2}
>>> d.keys()
dict_keys(['A', 'B', 'C'])
>>> d.values()
dict_values([0, 1, 2])
```

Pour l'accès à l'ensemble des couples clés-valeurs, nous utilisons la méthode items

```
>>> d.items()
dict_items([('A', 0), ('B', 1), ('C', 2)])
```

Remarque : les couples clés-valeurs obtenus sont du type tuple.

Nous pouvons tester l'appartenance à un dictionnaire avec le mot clé in

```
>>> "A" in d  # teste si "A" est une clé
True
>>> 3 in d.values()  # teste si 3 est une valeur
False
>>> ('C', 2) in d.items()  # teste si ('C', 2) est un couple clé-valeur
True
```

Nombre d'éléments :

La fonction len( ) renvoie le nombre d'éléments d'un dictionnaire, sa longueur.

Suppression d'un élément

Pour supprimer un élément, nous utilisons l'instruction del d[clé]

```
>>> d = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> del d["D"]
>>> d
{'A': 0, 'B': 1, 'C': 2}
```

Copie

Les comportements sont similaires à ceux rencontrés avec les listes en particulier si les valeurs sont des listes. Il est donc conseillé d'utiliser la fonction deepcopy du module copy pour être certain d'obtenir une "vraie" copie.

Les éléments d'un dictionnaire peuvent aussi être des dictionnaires. Voici un exemple :

```
vols = {'Lisbonne': {'heure': 21:10,
                     'num': 'EJU7674',
                     'compagnie': 'EASYJET'},
        'Vienne': {'heure': 21:25,
                   'num': 'OS430',
                   'compagnie': 'AUSTRIAN AIRLINES'},
        'Londres': {'heure': 21:55,
                    'num', 'BA357'
                    'compagnie': 'BRITISH AIRWAYS'}
        ...}
```

```
>>> vols['Lisbonne']
{'heure': 21:10, 'num': 'EJU7674', 'compagnie': 'EASYJET'}
```