éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

TYPES CONSTRUITS EN PYTHON

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. Introduction

En classe de seconde, des types de données simples sont présentés dans le cours de mathématiques. Ce sont les type int (nombres entiers), float (nombres flottants), bool (booléens). Le type str est aussi utilisé. Il est un peu moins simple. Un objet de type str est une chaîne de caractères qui s'écrit entre des guillemets ou des apostrophes. Un caractère est, pour simplifier, ce que l'on obtient par exemple avec les touches d'un clavier. Dans une chaîne, chaque caractère est repéré par un indice qui commence à 0. Avec la chaîne `ch ="exemple"`, `ch[0]` est le caractère "e", `ch[1]` est le caractère "x", et ainsi de suite.

Ces types simples ne sont plus suffisants si nous avons besoin de garder en mémoire un grand nombre de valeurs comme dans le cas d'un traitement de données statistiques. Il en est de même si l'on souhaite regrouper des valeurs, par exemple afin d'avoir une variable représentant les coordonnées d'un point.

L'objectif est donc de construire un type de variable capable de contenir plusieurs valeurs. Nous pouvons nous inspirer du type str et utiliser des indices pour repérer les éléments. Ceci amène à la construction des p-uplets, type tuple, et des listes, type list. Comme pour le type str, un objet `t` de type tuple n'est pas modifiable par une affectation `t[i]=valeur`. Un objet de type list est lui modifiable par une affectation ce qui autorise de nombreuses méthodes applicables à ces objets mais en contrepartie une grande vigilance sur leur utilisation. Un troisième type est présenté, le type dict pour les dictionnaires. La principale différence avec les listes est qu'un dictionnaire n'est pas ordonné. Un élément n'est pas repéré par un indice entier mais par une "clé".

2. P-uplets

2.1. Définition

Un objet de type tuple, un p-uplet, est une suite ordonnée d'éléments qui peuvent être chacun de n'importe quel type. On parlera indifféremment de p-uplet ou de tuple.

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, 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.

Opérations

Nous avons deux opérateurs de concaténation qui s'utilisent comme avec les chaînes de caractères, ce sont les opérateurs + et *. De nouveaux p-uplets sont créés.

```
>>> 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
```

2.2. Utilisation des indices

Les indices permettent d'accéder aux différents éléments d'un tuple. Pour accéder à un élément d'indice `i` d'un tuple `t`, la syntaxe est `t[i]`. L'indice `i` peut prendre les valeurs entières de 0 à $n-1$ où n est la longueur du tuple. Cette longueur s'obtient avec la fonction **`len`**. Exemple :

```
>>> t = "a", 1, "b", 2, "c", 3
>>> len(t)
6
>>> t[2]
'b'
```

La notation est celle utilisée avec les suites en mathématiques : $u_0, u_1, u_2, \ldots$: les indices commencent à 0 et par exemple le troisième élément a pour indice 2. Le dernier élément d'un tuple `t` a pour indice **`len`**`(t)-1`. On accède ainsi au dernier élément avec `t[len(t)-1]` qui peut s'abréger en `t[-1]`.

```
>>> t = "a", 1, "b", 2, "c", 3
>>> t[-1]
3
>>> t[-2]
'c'
```

Exemple avec des tuples emboîtés (un tuple contenant des tuples) :

```
>>> t = ("a", "b"), ("c", "d")
>>> t[1][0]
'c'
```

Explication : `t[1]` est le tuple `("c", "d")` et 'c' est l'élément d'indice 0 de ce tuple.

Rappelons ce qui a été annoncé plus haut : les éléments d'un tuple ne sont pas modifiables par une affectation de la forme `t[i]= valeur` qui provoque une erreur et arrête le programme.

2.3. 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.

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

Cette syntaxe s'utilise souvent avec une fonction qui renvoie un tuple.
Voici un exemple avec une fonction qui calcule et renvoie les longueurs des trois côtés d'un triangle ABC. La fonction prend en paramètres trois p-uplets représentant les coordonnées des trois points. On importe au préalable la fonction racine carrée `sqrt` du module `math`.

```
from math import sqrt

def longueurs(A, B, C):
    xA, yA = A
    xB, yB = B
    xC, yC = C
    dAB = sqrt((xB - xA) ** 2 + (yB - yA) ** 2)
    dBC = sqrt((xC - xB) ** 2 + (yC - yB) ** 2)
    dAC = sqrt((xC - xA) ** 2 + (yC - yA) ** 2)
    return dAB, dBC, dAC
```

La fonction étant définie, nous l'utilisons dans l'interpréteur :

```
>>> M = (3.4, 7.8)
>>> N = (5, 1.6)
>>> P = (-3.8, 4.3)
>>> dMN, dNP, dMP = longueurs(M, N, P)
>>> dMN
6.4031242374328485
```

3. Listes

3.1. Définition

Un objet de type list, que nous appelons une liste, ressemble à un p-uplet : un ensemble ordonné d'éléments avec des indices pour les repérer. Les éléments d'une liste sont séparés par des virgules et entourés de crochets.
Exemples :

```
>>> liste1 = ["a", "b", "c"] # une liste à 3 éléments
>>> liste2 = [1]  # une liste contenant un seul élément
>>> liste3 = [[1, 2], [3, 4]] # une liste de listes
>>> liste_vide = []  # une liste vide
```

Création d'une liste

Pour créer une liste d'entiers, nous pouvons utiliser les fonctions `list` et `range`.

```
liste = list(range(2, 10, 3)) # [2, 5, 8]
```

Pour ajouter les éléments un par un en fin de liste, nous utilisons une boucle et la méthode `append` :

```
multiples_de_3 = []
for i in range(100):
    multiples_de_3.append(3 * i)
```

3.2. Construction par compréhension

L'instruction s'écrit sous la forme `[expression(i) for i in objet]`. Ce type de construction est très spécifique au langage Python. En voici deux exemples :

```
multiples_de_3 =  [3 * i for i in range(100)]
multiples_de_6 =  [2 * n for n in multiples_de_3]
```

Si on dispose d'une fonction `f` et d'une liste d'abscisses :

```
images = [f(x) for x in abscisses]
```

Un exemple avec des listes emboîtées :

```
>>> liste = [[[i, j] for i in range(3)] for j in range(2)]
>>> liste
[[[0, 0], [1, 0], [2, 0]], [[0, 1], [1, 1], [2, 1]]]
```

3.3. Utilisation

La fonction **`len`** renvoie la longueur d'une liste, le nombre d'éléments de la liste.

Accès aux éléments

On accède aux différents éléments d'une liste avec les indices comme pour les p-uplets.

```
>>> liste = ["a", "b", "c"]
>>> liste[1]
'b'
>>> liste =  [["a", "b"], ["c", "d"]]
>>> liste[1][0]
'c'
```

Méthodes

Le type d'une variable définit les valeurs qui peuvent être affectées à cette variable ainsi que les opérateurs et les fonctions utilisables. Les fonctions propres à un type donné sont appelées des méthodes. La fonction **`len`** par exemple, s'applique aux chaînes de caractères, aux p-uplets, aux listes. La méthode `append`, présentée plus haut, est par contre propre aux listes.
Attention à la syntaxe : on écrit **`len`**`(liste)`, mais `liste.append(...)`. Le nom de la variable est suivi d'un point puis du nom de la méthode.

Voici quelques méthodes :

```
>>> liste = ["a", "b", "c"]
>>> liste.insert(1, "d")  # insertion à l'indice 1 de l'élément "d"
>>> liste
['a', 'd', 'b', 'c']
>>> liste.remove("b")
>>> liste
['a', 'd', 'c']
>>> x = liste.pop()
```

```
>>> x
'c'
>>> liste
['a', 'd']
>>> liste.reverse()
>>> liste
['d', 'a']
>>> liste.sort()
>>> liste
['a', 'd']
```

Toutes ces méthodes modifient la liste initiale contrairement aux opérateurs de concaténation + et * avec lesquels une nouvelle liste est créée. Ces deux opérateurs s'utilisent comme avec les p-uplets.

Notons qu'il est aussi possible de trier une liste sans la modifier avec la fonction **sorted** qui crée une nouvelle liste.

```
>>> liste = [5, 2, 7, 4]
>>> tri = sorted(liste)
>>> liste
[5, 2, 7, 4]
>>> tri
[2, 4, 5, 7]
```

Copie

Une liste peut donc être modifiée par une méthode. On peut aussi modifier l'un de ses éléments par affectation.

```
>>> liste = ["a", "b", "c"]
>>> liste[1] = "d"
>>> liste
["a", "d", "c"]
```

Ceci oblige à une grande attention en particulier dans la création de copie d'une liste. Observons le code suivant :

```
>>> liste1 = ["a", "b", "c"]
>>> liste2 = liste1
>>> liste1[1] = "d"
>>> liste2
["a", "d", "c"]
```

Dans cet exemple, une même liste a deux noms et `liste2` n'est pas une copie de `liste1`. Pour obtenir une copie, il faut créer une nouvelle liste.
Par exemple :

```
>>> liste2 = list(liste1)
```

Il s'agit d'une copie superficielle, disons de niveau 1. Pour comprendre le fonctionnement, il faut considérer qu'une liste est juste une adresse (en mémoire) qui contient les adresses de ses éléments. Avec cette copie superficielle, une nouvelle liste est créée avec une nouvelle adresse. Les éléments gardent eux la même adresse.

Examinons un autre exemple avec `liste = 2 * [["a", "b"]]`. Cela revient à écrire `liste1 = ["a", "b"]`, puis `liste = 2 * [liste1]`. Le résultat est `[['a', 'b'], ['a', 'b']]`. Les deux éléments ont ici la même adresse, celle de `liste1`. Donc une modification d'un élément de `liste1` concerne les deux éléments de `liste`. Que ce soit `liste1[1] = "c"` ou `liste[0][1] = "c"`, le résultat est le même, `[['a', 'c'], ['a', 'c']]`.

Par contre avec l'instruction `liste = [["a", "b"] for i in range(2)]`, une première liste `["a", "b"]` est créée quand `i` prend la valeur 0, puis une seconde liste quand `i` prend la valeur 1. Ces deux listes sont distinctes, (les adresses sont différentes), donc modifier l'une n'a aucune conséquence sur l'autre.

Considérons le cas d'une liste dont les éléments sont des listes. Une copie superficielle comme ci-dessus crée une nouvelle liste avec une nouvelle adresse et les éléments de cette copie ont eux la même adresse que les éléments de la liste initiale. Par exemple avec `liste1 = [['a', 'b'], ['c', 'd']]` puis `liste2 = list(liste1)`, les adresses de `liste1` et `liste2` sont distinctes mais les adresses de `liste1[0]` et `liste2[0]` sont identiques. Donc la modification d'un élément de `liste1[0]` se répercute sur `liste2[0]`. Pour obtenir une copie "en profondeur", passer au niveau 2, il faut donner de nouvelles adresses pour les listes éléments d'une liste.

Des fonctions de copie sont disponibles à partir du module `copy`. En particulier, dans le cas d'une liste de listes, pour effectuer une copie en profondeur, nous disposons de la fonction `deepcopy`.

```
>>> from copy import deepcopy
>>> liste1 = [["a", "b"], ["c", "d"]]
>>> liste2 = deepcopy(liste1)
>>> liste2[1][0] = "e"
>>> liste2
[["a", "b"], ["e", "d"]]
>>> liste1
[["a", "b"], ["c", "d"]]
```

3.4. Applications

Calcul d'une moyenne

```
def moyenne(liste):
    s = 0
    for u in liste:
        s += u
    return s/len(liste)
```

Représentation graphique d'une fonction

```
import matplotlib.pyplot as plt

def f(x):
    return x ** 2 + x - 4

liste_x = [0.1 * n for n in range(-30, 31)]
liste_y = [f(x) for x in liste_x]
plt.plot(liste_x, liste_y)
plt.show()
```

Calculs des termes d'une suite

```
def suite(n, u0):
    u, termes = u0, [u0]
    for i in range(n):
        u = f(u)  # f est la fonction définie dans le programme ci-dessus
        termes.append(u)
    return termes
```

4. Dictionnaires

4.1. Définition

Les éléments d'une liste sont repérés par des indices 0, 1, 2, ... Une différence essentielle avec un dictionnaire, objet de type dict, est que les indices sont remplacés par des objets du type str, float, tuple (si les n_uplets ne contiennent que des entiers, des flottants ou des p-uplets). On les appelle des clés et à chaque clé correspond une valeur. Ces clés ne sont pas ordonnées.

4.2. Création

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}
```

4.3. Utilisation

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
```

On en déduit trois manières de parcourir un dictionnaire. Il est possible d'utiliser les clés, les valeurs, ou les couples clés-valeurs. Voici un exemple avec les clés :

```
>>> for key in d:
        print(key)

A
B
C
```

L'accès à une valeur particulière s'obtient en précisant la clé. Par exemple, `d["A"]` a la valeur 0.

Nous pouvons modifier une valeur par affectation comme `d["A"] = 1`. Et l'instruction `d["D"] = 3` ne provoque pas d'erreur. Une nouvelle clé est créée.

Attention : une instruction comme `v = d["E"]` provoque une erreur car la clé "E" n"existe pas. La méthode `get` permet de gérer ce genre de problème.

```
>>> v = d.get("A")
>>> v
0
>>> v = d.get("E")
>>> print(v)
None
```

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'}
```

L'implémentation d'un dictionnaire optimise le coût en temps de la recherche d'un élément.

4.4. Application

Les photos prises avec un appareil numérique contiennent de nombreuses informations. Dans le fichier image, par exemple au format jpeg, sont stockées des données non seulement sur l'image elle-même mais aussi sur l'appareil, le logiciel utilisé, et en particulier des données EXIF (Exchangeable Image File Format). Une partie est accessible dans les propriétés de fichier ou avec un logiciel de traitement d'images.

Les spécifications sont gérées par un organisme japonais, le JEITA, qui définit différents dictionnaires de référence permettant d'accéder à ces données. Les clés (les tags) et les valeurs sont des nombres écrits dans l'entête du fichier. Les dictionnaires donnent l'interprétation de ces clés. Par exemple la clé 256 a pour valeur `'Width'`, la largeur de l'image en pixel, la clé 257 a pour valeur `'Length'`, la hauteur de l'image en pixel.