Cours sur les p-uplets

Thème types construits

Première NSI, Lycée du Parc

Table des matières

Crédits

Ce cours est inspiré du chapitre 14 du manuel NSI de la collection Tortue chez Ellipse, auteurs : Ballabonski, Conchon, Filliatre, N'Guyen. J'ai également consulté le prepabac Première NSI de Guillaume Connan chez Hatier, le document ressource eduscol sur les types construits et le livre ***Fluent Python*** *.*

1 p-uplets en Python

Définition 1

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**.

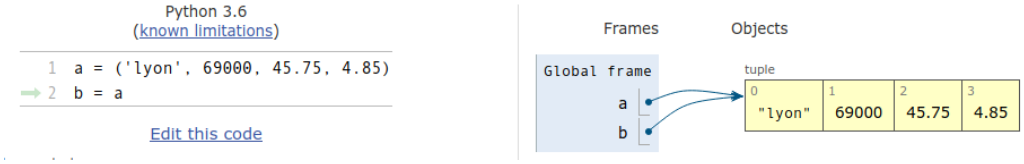
Un **p-uplet** est utilisé comme **enregistrement** de données hétérogènes qui sont liées entre elles, par exemple pour une ville : (nom, code postal, latitude, longitude).

```
>>> a = ('lyon', 69000, 45.75, 4.85)
```

Propriété 1

En Python, un **p-uplet** est un objet de type `tuple`. Pour le définir, on entoure de parenthèses la séquence ordonnée d'éléments qu'il contient.

Un objet de type `tuple` partage de nombreuses propriétés avec les objets de type `list` que nous avons utilisés pour implémenter les tableaux homogènes. En particulier, sa valeur est une **référence** vers la zone mémoire où est stockée la séquence d'objets (propriété d'aliasing).



Les objets de type `tuple` diffèrent de ceux de type `list` car ils ne peuvent être modifiés une fois qu'ils sont créés : on dit qu'ils sont **immutables**. Cette propriété, partagée avec les chaînes de caractères de type `str` est importante car elle facilite la gestion en mémoire des données qui ne doivent pas changer, la démonstration de propriétés des programmes et elle permet leur utilisation comme clef dans les tableaux associatifs ou tables de hachage. Ces derniers sont implémentés par un autre type construit très important en Python, celui des **dictionnaires**.

A part les propriétés de modification, les autres propriétés du type `list` sont disponibles pour le type `tuple`.

```
>>> a = ('lyon', 69000, 45.75, 4.85)
>>> type(a)
<class 'tuple'>
>>> a[0]
'lyon'
>>> len(a)
4
>>> a[0] = 'St-Etienne'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Méthode

Présentons les principales opérations sur les **tuples** en Python. Le type **tuple** implémente toutes les méthodes du type **list** sauf celles d'ajout ou de modification d'élément. Pour obtenir la liste de ces méthodes, on peut évaluer `dir(tuple)`dans une console.

- **Construction :**
  - *Par extension*, on définit un **tuple** en séparant ses éléments par une virgule. En particulier, s'il ne contient qu'un seul élément, il faut le faire suivre d'une virgule. Les parenthèses

ne sont pas obligatoires mais sont nécessaires si on imbrique des **tuples**. On peut aussi convertir en **tuple** un autre itérable (`list`, `str`, `range`) avec le constructeur `tuple` .

```
>>> notes = ('paul',10, 12,18)
>>> telephone = 'paul', '0606060606'
>>> telephone
('paul', '0606060606')
>>> singleton = 'Solo',
>>> singleton
('Solo',)
>>> s = ('solo')
>>> s
'solo'
>>> los_angeles = ('Lax aiport', (33.9425, -118.408056))
>>> tuple([1,2])
(1, 2)
>>> tuple(range(3))
(0, 1, 2)
```

– *Par compréhension*, la syntaxe est la même que pour les tableaux de type `list`, mais attention, il faut utiliser le constructeur `tuple`, sinon on construit un **générateur** (distributeur d'objets comme `range`). En général on construit plutôt une structure imbriquée avec un tableau de **tuples**.

```
>>> m = (k for k in range(10))
>>> m
<generator object <genexpr> at 0x7f26afa0ef90>
>>> tuple(m)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> famille = ['trefle', 'pique', 'carreau', 'coeur']
>>> hauteur = ['V', 'R', 'D']
>>> valets = [(h,f) for f in famille for h in hauteur if h == 'V']
>>> valets
[('V', 'trefle'), ('V', 'pique'), ('V', 'carreau'), ('V', 'coeur')]
```

- **Accès en lecture :** Seul l'accès en lecture est permis, on utilise les index (de 0 à `len(tuple)-1`) comme pour les objets de type `list`. Une propriété très utilisée est le *tuple unpacking* qui permet de déballer les éléments d'un **tuple** à droite d'un symbole d'affectation `=` pour les assigner à un **tuple** d'identifiants à gauche du `=`. On peut ainsi échanger des variables en une seule instruction. Cela fonctionne aussi sur les autres objets itérables comme ceux de type `list`.

```
>>> paul = ('identifiant', '01011970')
>>> len(paul)
2
>>> paul[1] = '31122000'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
>>> login, password = paul
>>> login
'identifiant'
>>> password
'01011970'
>>> login, password = password, login #échange de variables par tuple
    unpacking
>>> login, password
('01011970', 'identifiant')
>>> paul
('identifiant', '01011970')
```

- **Parcours :** Comme pour les objets de type `list` et les itérables en général, on peut parcourir un **tuple** par index ou par éléments. Pour un tableau de **tuple**, on peut déballer directement dans la boucle `for` avec le *tuple unpacking*.

```
>>> notes
('paul', 10, 12, 18)
>>> for k in range(len(notes)):
...     print('Index : ', k, 'Valeur : ', notes[k])
...
Index : 0 Valeur : paul
Index : 1 Valeur : 10
Index : 2 Valeur : 12
Index : 3 Valeur : 18
>>> for element in notes:
...     print(element)
...
paul
10
12
18
>>> passeports = [('USA', '31195855'), ('BRA', 'CE342567'), ('FRA', '
    XDA502856')]
>>> for pays, numero in passeports:
...     print(pays, numero)
...
USA 31195855
BRA CE342567
FRA XDA502856
```

- **Concaténation :** On peut concaténer deux **tuples** et on crée alors un nouveau **tuple**. On peut l'observer en affichant l'identité du **tuple** avec la fonction `id` : l'entier affiché représente l'identifiant mémoire de l'objet. On ne peut pas ajouter des éléments à un **tuple** comme pour un tableau avec la méthode `append`. On peut factoriser l'opération de concaténation avec

l'opérateur *.

```
>>> a = ('Limoges',)  #attention tuple avec un seul élément
>>> id(a)             #identifiant de a
139804156798144
>>> b = (45.85, 1.25) # latitude et longitude
>>> a = a + b
>>> a
('Limoges', 45.85, 1.25)
>>> id(a) #l'identifiant de a est changé, un nouveau tuple a été créé
139804132207488
>>> population = 132175
>>> a.append(population)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> d = ('paris',)
>>> 3 * d
('paris', 'paris', 'paris')
```

- **Test d'appartenance, comparaison :** On peut tester l'appartenance d'un élément à un **tuple** avec l'opérateur `in` et comparer deux **tuples** avec `==`, `<`, `>`. La comparaison de deux **tuples** s'effectue de gauche à droite selon *l'ordre lexicographique*.

```
>>> b = ('Lyon', 516092)
>>> a = ('Lyon', 516092)
>>> id(a), id(b)
(139804132208448, 139804132156672)
>>> a is b
False
>>> a == b
True
>>> 'Lyon' in a
True
>>> 'St-Etienne' not in a
True
>>> ('adama', 10, 6) < ('adam', 10)
False
>>> ('adama', 10, 6) > ('adama', 10)
True
```

- **Recherche, dénombrement :** On peut rechercher l'index de la première occurence d'un élément dans un **tuple** avec la méthode `index`, ou compter le nombre d'occurences d'une valeur avec `count`.

```
>>> from random import randint
>>> e = tuple([randint(1,6) for _ in range(10)])
```

```
>>> e
(2, 6, 1, 3, 5, 1, 5, 2, 5, 5)
>>> e.index(5)
4
>>> e.count(1)
2
```

Exercice 1

Dans le plan muni d'un repère orthonormé, chaque point est représenté par un **tuple** de coordonnées :

```
>>> M = (1, 2)
>>> N = (4,6)
>>> P = (9,8)
```

1. Compléter la fonction Python ci-dessous pour que `milieu(A, B)` retourne le **tuple** de coordonnées du milieu du segment reliant les points `A` et `B` passés en paramètres.

```
def milieu(A, B):
  xA, yA = A
  xB, yB = B
  # à compléter
```

2. Écrire une fonction `longueurs(A, B, C)` qui prend en paramètres trois points `A, B, C` et retourne le triplet de longueurs des côtés du triangle `ABC`. On rappelle la formule de la distance entre deux points $A(x_A, y_A)$, $B(x_B, y_B)$: $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$. On importera la fonction racine carrée depuis le module `math` avec `from math import sqrt` et on pourra écrire aussi une fonction de signature `distance(A, B)` qui renvoie la distance entre les points A et B.

Exercice 2

La fonction ci-dessous permet de générer un fichier texte au format CSV contenant `nb_triplets` lignes contenant chacune un `tuple` de trois entiers `a,b,c` choisis aléatoirement entre 1 et 100.

```
def generateur_triplets(path, nb_triplets):
    """
    Génère un fichier texte au format CSV où chaque ligne contient
    un tuple a,b,c constitué de trois entiers aléatoires entre 0 et 1000

    Parametres
    ----------
    path (str) : chemin vers un fichier texte de format csv
    nb_triplets (int) : nombre de triplets
```

```
    Retour
    -------
    None.

    """
    f = open(path, "w", encoding="utf-8")
    for _ in range(nb_triplets):
        triplet = tuple(str(random.randint(1, 100)) for _ in range(3))
        f.write(','.join(triplet) + '\n')
    f.close()
```

Le fichier `triplets.csv` contient 10000 triplets générés par cette fonction.

Compléter les fonctions ci-dessous pour satisfaire leurs spécifications et les deux tests unitaires fournis.

```
def compte_triangle(path):
    """
    Compte le nombre de triangles parmi les triplets
    a,b,c contenus dans le fichier de chemin path

    Parameters
    ----------
    path (str) : chemin vers un fichier texte au format CSV

    Returns
    -------
    nb_triangle (int)

    """
    f = open(path, encoding="utf-8")
    nb_triangle = 0
    for enreg in f:
        a, b, c = tuple(int(champ) for champ in enreg.split(','))
        # à compléter


    f.close()
    return nb_triangle


def compte_triplets_pythagoriciens(path):
    """
    Compte le nombre de triplets pythagoriciens (longueurs
    des cotes d'un triangle rectangle) parmi les triplets
    a,b,c contenus dans le fichier de chemin path
```

```
    Parameters
    ----------
    path (str) : chemin vers un fichier texte au format CSV

    Returns
    -------
    nb_triangle (int)

    """
    f = open(path, encoding="utf-8")
    #à  compléter




    f.close()
    return nb_pythagore

# generateur_triplets("triplets.csv", 10000)


def test_unitaire():
    assert compte_triangle("triplets.csv") == 5223
    assert compte_triplets_pythagoriciens("triplets.csv") == 5
```

2 QCM de type E3C2

Exercice 3

1. **Question 1 :** On définit :

```
tab = [ ('Léa', 14), ('Guillaume', 12), ('Anthony', 16),
('Anne', 15) ]
```

Quelle est la valeur de l'expression `[x[0] for x in tab if x[1] >= 15]` ?

Réponses

- A `[('Anthony', 16), ('Anne', 15)]`
- B `['Anthony', 'Anne']`
- C `[16, 15]`

- D TypeError : `'tuple'object is not callable`

2. **Question 2 :** Une table d'un fichier client contient le nom, le prénom et l'identifiant des clients sous la forme :

```
clients = [ ("Dupont", "Paul", 1),("Durand", "Jacques", 2),("Dutronc
     ", "Jean", 3),...]
```

En supposant que plusieurs clients se prénomment Jean, que vaut la variable x après l'exécution du code suivant ?

```
x = []
for i in range(len(clients)):
  if clients[i][1] == "Jean":
    x = clients[i]
```

Réponses

- A Une liste de tuples des noms, prénoms et numéros de tous les clients prénommés Jean
- B Une liste des numéros de tous les clients prénommés Jean
- C Un tuple avec le nom, prénom et numéro du premier client prénommé Jean
- D Un tuple avec le nom, prénom et numéro du dernier client prénommé Jean

3. **Question 3 :** Quel est le type de l'expression `f(4)` si la fonction `f`est définie par :

```
def f(x):
  return (x, x**2)
```

Réponses

- A un entier
- B un flottant
- C une liste
- D un tuple

3 Synthèse

À retenir

- Un **p-uplet** ou **tuple** est une séquence ordonnée d'éléments qui peuvent être de type hétérogènes.
- La séquence d'informations rassemblée dans un **tuple** est un **enregistrement**.
- En Python, les **tuples** sont de type `tuple` et sont **immutables**. Ils partagent les mêmes opérations que les tableaux de type `list` sauf les opérations de modification et d'ajout.