p-uplets (type `tuple`)

Cours

Un **p-uplet** (ou n-uplet) est une suite **ordonnée** d'éléments qui ne peuvent **pas être modifiés** (immuables[3]).

En Python les p-uplets sont du type `tuple`.

Deux éléments ensembles forment un couple, trois éléments un triplet, quatre éléments un quadruplet, etc., et par extension *p* éléments ensembles forment un p-uplet.

Création

Un p-uplet est déclaré par une suite de valeurs, séparées par des virgules, et le tout encadré par des parenthèses " `( )` ". Il peut contenir des valeurs de types différents.

```
>>> p = (1, 2.5, 'hello', True)
```

Il est aussi possible de créer un p-uplet sans parenthèse :

```
>>> p = 1, 2.5, 'hello', True
>>> p
(1, 2.5, 'hello', True)
>>> type(p)
<class 'tuple'>
```

⚠️ Ne pas confondre la virgule qui sépare les différents éléments avec le point utilisé pour les nombres de type `float`, c'est souvent un risque d'erreur :

```
>>> p_2_elem = (1.2,3) # Creation d'un tuple avec 2 éléments : 1.2 (type float) et 3 (type int)
>>> p_3_elem = (1,2,3) # Creation d'un tuple avec trois éléments : 1 2 et 3 (type int)
```

Un p-uplet peut contenir des éléments de types différents, y compris d'autres p-uplets :

Le second p-uplet `(3, 4.0, 'bye', False)` doit obligatoirement être écrit entre parenthèse dans ce cas.

```
>>> p_de_p = p, (3, 4.0, 'bye', False)
>>> p_de_p
((1, 2.5, 'hello', True), (3, 4.0, 'bye', False))
```

D'autres exemples de p-uplets :

- p-uplet vide, les parenthèses sont obligatoires ici :

```
>>> p_vide = ()
```

- p-uplet avec un seul élément écrit avec une virgule à la fin :

```
>>> p_1_elem = 1,
>>> p_1_elem
(1,)
```

ou écrit avec des parenthèses :

```
>>> autre_p_1_elem = (1,)
>>> autre_p_1_elem
(1,)
```

- mais attention, c'est finalement la virgule plus que les parenthèses qui crée le p-uplet, ici `pas_p` n'est pas un p-uplet, c'est un entier !

```
>>> pas_p = (1)
>>> pas_p
1
>>> type(pas_p)
<class 'int'>
```

Fonction `len()`

La fonction `len()` renvoie la longueur d'un p-uplet, c'est-à-dire le nombre d'éléments qu'il contient.

```
>>> p = (1, 2.5, 'hello', True)
>>> len(p)
4
```

Accès aux éléments

Comme pour les chaines de caractères, la position de chaque élément d'un p-uplet `p` est indexée à partir **de `0` jusqu'à `len(p)` exclu**, c'est-à-dire le dernier élément est en position `len(p) - 1`. Il y a donc bien `len(p)` élements dans le p-uplet.

Il est possible d'accéder aux éléments par leur indice entre crochets.

```
>>> p = (1, 2.5, 'hello', True)
>>> p[1]
2.5
```

⚠️ Le premier élément est à l'indice 0.

Les positions des éléments d'un p-uplet peuvent également être indexées avec des nombres négatifs selon le modèle suivant :

`>>> p =`	`(`	`1,`	`2.5,`	`'hello,'`	`True`	`)`
indice positif		0	1	2	3	
indice négatif		-4	-3	-2	-1	

Les indices négatifs reviennent à compter à partir de la fin, `-1` est du raccourci syntaxique[1] pour `len(p) - 1`. Leur principal avantage est d'accéder au dernier élément d'un p-uplet à l'aide de l'indice `-1` sans pour autant connaître sa longueur. L'avant-dernier élément a lui l'indice `-2`, l'avant-avant dernier l'indice `-3`, etc.

```
>>> p[-1]
True
>>> p[-2]
'hello'
```

L'accès à une partie d'un p-uplet (une « tranche ») se fait sur le modèle `p[début:fin]`[2] pour récupérer tous les éléments, entre les positions `debut` (inclus) et `fin` (**exclu**).

```
>>> p[1:2]
(2.5,)
>>> p[1:3]
(2.5, 'hello')
>>> p[1:-1]
(2.5, 'hello')
```

Lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

```
>>> p[2:]
('hello', True)
>>> p[:2]
(1, 2.5)
>>> p[:]
(1, 2.5, 'hello', True)
```

Il est aussi possible de disperser, ou « déballer », un p-uplet en affectant tous ses éléments dans plusieurs variables :

```
>>> a, b, c, d = (1, 2.5, 'hello', True)
>>> b
2.5
```

Ce qui pouvait aussi s'écrire sans parenthèse :

```
>>> a, b, c, d = 1, 2.5, 'hello', True
```

Le mot clé `in` permet de vérifier si un élément `elem` est présent dans un p-uplet `p`, `elem in p` renvoie un booléen :

```
>>> p = (1, 2.5, 'hello', True)
>>> 'hello' in p
True
>>> 4 in p
False
```

Le mot clé `in` permet aussi d'écrire une boucle pour parcourir (ou «itérer sur») toutes les valeurs d'un p-uplet.

Comparons différentes façons pour parcourir un p-uplet `p` :

Avec une boucle non bornée `while`

Il faut gérer l'indice de boucle `i` pour qu'il parcourt toutes les positions des valeurs de `p`, c'est-à-dire l'initialiser à `0` puis l'incrémenter à chaque passage dans la boucle (`i = i + 1`) jusqu'à ce qu'il dépasse `len(p) - 1`. `p[i]` permet d'accéder à la valeur du p-uplet qui se trouve à la position `i`.

```
>>> p = (1, 2.5, 'hello', True)
>>> i = 0
>>> while i < len(p):
...     print(p[i])
...     i = i + 1
...
1
2.5
'hello'
True
```

Avec une boucle bornée `for`

Avec `for i in range(len(p)):`, l'indice de boucle `i` prend automatiquement les valeurs allant de `0` à `len(p) - 1`. `p[i]` permet d'accéder à la valeur du p-uplet qui se trouve à la position `i`.

```
>>> p = (1, 2.5, 'hello', True)
>>> for i in range(len(p)):
...     print(p[i])
...
1
2.5
'hello'
True
```

Avec une boucle bornée `for` et le mot clé `in`

`for elem in t` permet d'accéder directement à toutes les valeurs du p-uplet les unes après les autres, sans connaître leurs positions.

```
>>> p = (1, 2.5, 'hello', True)
>>> for elem in p:
...     print(elem)
...
 1
2.5
```

```
'hello'
True
```

La boucle `for elem in p` est plus simple pour parcourir les valeurs d'un p-uplet, par exemple pour rechercher la plus petite ou la plus grande valeur dans ce p-uplet, mais elle ne permet pas d'accéder à sa position. Pour accéder à la position d'une valeur que l'on recherche, il faut utiliser les boucles sur indices de position `while i < len(p):` ou `for i in range(len(p)):`.

⚠️ Un p-uplet est **immuable**, il est possible d'accéder à ses éléments, mais **pas de les modifier**.

```
>>> p = (1, 2.5, 'hello', True)
>>> p[1] = 3
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Opérations sur p-uplets

Deux opérations sont possibles, l'addition et la multiplication :

- L'opérateur d'addition « `+` » **concatène** (assemble) deux p-uplets.
- L'opérateur de multiplication « `*` » entre un nombre entier et un p-uplet **duplique** (répète) plusieurs fois les éléments dans un nouveau p-uplet.

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> 3 * (1, 2)
(1, 2, 1, 2, 1, 2)
```

Fonctions renvoyant un p-uplet

Les p-uplets sont très utiles pour écrire des fonctions renvoyant plusieurs valeurs en même temps :

```
1  from math import pi
2
3  def cercle_info(r):
4      """ (float) -> (float, float)
5      Renvoie le p-uplet (circonference, aire) d'un cercle de rayon r
6      """
7      c = 2 * pi * r
8      a = pi * r**2
9      return c, a
```

La dernière ligne `return c, a` peut tout aussi bien s'écrire `return (c, a)`, dans les deux cas la fonction renvoie exactement le même p-uplet.

Appelons maintenant la fonction `cercle_info()`, par exemple pour avoir la circonférence et l'aire d'un cercle de rayon 10 :

```
>>> cercle_info(10)
```

```
(62.83185307179586, 314.1592653589793)
```

La fonction renvoie un p-uplet de deux valeurs, la circonférence et l'aire du cercle, pour n'avoir que l'un des deux il faut accéder au premier et au deuxième élément du p-uplet :

```
>>> cercle_info(10)[0]            # circonférence d'un cercle de rayon 10
62.83185307179586
>>> cercle_info(10)[1]            # aire d'un cercle de rayon 10
314.1592653589793
```

Complétons le programme précédent pour demander à l'utilisateur de saisir le rayon du cercle :

```
10  rayon = float(input('Rayon du cercle ?'))
11  print('La circonférence du cercle est', cercle_info(rayon)[0])
12  print("L'aire du cercle est", cercle_info(rayon)[1])
```

Ici `cercle_info(rayon)[0])` et `cercle_info(rayon)[1])` permettent de récupérer la première et la seconde valeur du p-uplet renvoyé par l'appel de la fonction `cercle_info(rayon)`. Ce p-uplet peut aussi être dispersé dans deux variables, ce qui rend le code plus lisible :

```
10  rayon = float(input('Rayon du cercle ?'))
11  perimetre, aire = cercle_info(rayon)         # disperser le tuple renvoyé par cercle_info
12  print('La circonférence du cercle est', perimetre)
13  print("L'aire du cercle est", aire)
```

Conversion de type (cast)

La fonction `tuple()`, prend en argument un objet séquentiel (une chaine de caractère par exemple) et renvoie le p-uplet correspondant :

```
>>> tuple("ABCDEF")
('A', 'B', 'C', 'D', 'E', 'F')
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

---

1. appelé « sucre syntaxique » pour désigner un raccourci de syntaxe d'un langage de programmation facilitant sa lecture. ↩
2. Il est aussi possible de préciser un `pas` sur le modèle `p[début:fin:pas]`. ↩
3. ou *immutable* en anglais. ↩