Python - Tuples

[Mise à jour le : 31/1/2022]

- **Source**
  - **Documentation** sur Python.org : référence du langage, tuples et séquences, types séquentiels list, tuple, range, fonctions natives (built-in)
- **Mots-clés** : immuable, unpacking, tuple

Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

and	continue	finally	is	raise
as	def	for	lambda	return
assert	del	from	None	True
async	elif	global	nonlocal	try
await	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield

- Fonctions natives **(built-in)**[1] utilisées dans les exemples : **list**(), **print**(), **range**(), **tuple**().

---

1. Introduction

Le tuple est une séquence. Comme la liste, il référence des objets hétérogène. On peut lui appliquer le test d'appartenance avec **in**, accéder aux différents éléments avec un crochet, faire du slicing etc. La raison d'être principale du type tuple est de construire des objets globalement immuables.

Un **tuple**, ou **P-uplet** est une liste de **constantes**. Il est **immuable** donc non modifiable. Les **parenthèses ( )** délimitent les **tuples**.

2. Création

Exemple

*.py

```
# Affectation simple
tp =  "a","b",4 # Création d'un tuple à 3 éléments
tp = () # Création d'un tuple vide (peu d'intérêt)
tp = 6, # un seul élément nécessite une virgule !!!

# Les parenthèses sont obligatoires dès que l'écriture d'un tuple est
contenu dans une expression plus longue.
tp = 3,(1,True,4.5,8, "a") # différent types
>>> tp # renvoie (3, (1, True, 4.5, 8, 'a'))

# Tuple contenant des tuples
t2d = (1,"a"), # équivalent à (1,"a"),(2,"b")
      (2,"b")

# Affectation  multiple
a,b,c = 1,2,3 # a=1, b=2, c=3
```

Un tuple avec **un seul élément** nécessite une **virgule**.

3. Accès aux éléments

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 ou n est la longueur du tuple.

*.py

```
# Accès à une valeur en lecture
tp = "a",2,8,7,"b"
>>> tp # renvoie ('a', 2, 8, 7, 'b')
>>> len(tp) # renvoie 5
>>> tp[2] # renvoie 8
# i peut être négatif !
>>> tp[-1] # renvoie "b" car on parcourt le tuple en partant de la fin
```

4. Construction

- **Issue d'une concatenation avec +**

Bien que le type tuple soit immuable, il est tout à fait légal d'“additionner” deux tuples, et l'“addition” va produire un **nouveau** tuple

Exemple

*.py

```
tuple1 = (1, 2,)
tuple2 = (3, 4,)
print('Resultat:', tuple1 + tuple2) # Résultat: (1, 2, 3, 4)
```

- **Issue d'une concaténation avec ***

Exemple

*.py

```
tuple3 = 3*tuple1 # (1,2,1,2,1,2)
```

- **Issue d'une liste**

Exemple

*.py

```
liste = list(range(10)) # résultat[0,1,2,3,4,5,6,7,8,9]
liste[9] = 'Inconnu' # résultat[0,1,2,3,4,5,6,7,8,'Inconnu']
del liste [2:5] # résultat : [0, 1, 5, 6, 7, 8, 'Inconnu']
mon_tuple = tuple(liste) # résultat : (0, 1, 5, 6, 7, 8, 'Inconnu')
```

5. Appartenance

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

*.py

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

6. Modification !

Comme l'objet est immuable, la modification est impossible. Il faut transformer le tuple en liste et créer un nouveau tuple à partir de la liste modifiée.

Exemple

*.py

```
tp = "a",2,8,7,"b"
>>> tp # renvoie ('a', 2, 8, 7, 'b')
>>> tp[2] = 3 # Renvoie une erreur car un tuple est immuable
              # Traceback (most recent call last):
              #       File "<stdin>", line 1, in <module>
              #  TypeError: 'tuple' object does not support item
assignment
lst = list(tp)
>>> lst # renvoie la liste ['a', 2, 8, 7, 'b']
lst[2] = 3 # modification de la liste
tp = tuple(lst) # conversion en un tuple
>>> tp # renvoie le tuple ('a', 2, 3, 7, 'b')
```

7. Fonction renvoyant un tuple

Les affectations multiples (§2) s'utilisent souvent dans les fonctions renvoyant un tuple.

Exemple : calcul des longueurs des trois côtés d'un triangle.

*.py

```
from math import sert

# Déclaration de la fonction
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

# Utilisation
M = (3.4,7.8)
N = (5,1.6)
P = (-3.8,4.3)
dMN, dNP, dMP = longueurs(M,N,P)
dMN # renvoie 6.4031...
```

8. tuple unpacking

L'affectation dans Python peut concerner plusieurs variables à la fois.

Exemple 1

*.py

```
couple = (100, 'spam')
gauche, droite = couple
print('gauche:', gauche, 'droite:', droite) # résultat gauche: 100
droite: spam
```

Exemple 2

*.py

```
# membre droit: une liste
liste = [1, 2, 3]
# membre gauche : un tuple
gauche, milieu, droit = liste
print('gauche:', gauche, 'milieu:', milieu, 'droit:', droit) # gauche:
1 milieu: 2 droit: 3
```

Les seules contraintes fixées par Python sont que :
- le terme à droite du signe = soit un itérable (tuple, liste, string, etc.) ;
- le terme à gauche soit écrit comme un tuple ou une liste - notons tout de même que l'utilisation d'une liste à gauche est rare et peu pythonique ;
- les deux termes aient la même longueur - en tout cas avec les concepts que l'on a vus jusqu'ici, mais voir aussi plus bas l'utilisation de *arg avec le extended unpacking.

Exemple 3

Il est possible de faire une boucle for qui itère sur une seule liste mais qui agit sur **plusieurs variables**.

*.py

```
entrees = [(1, 2), (3, 4), (5, 6)]
for a, b in entrees:
    print(f"a={a} b={b}")

'''
a=1 b=2
a=3 b=4
a=5 b=6
'''
```

9. Extended unpacking

Le mécanisme ici est une extension de sequence unpacking ; Python vous autorise à mentionner une seule fois, parmi les variables qui apparaissent à gauche de l'affectation, une variable précédée de *.

Exemple 1

*.py

```
reference = [1, 2, 3, 4, 5]
a, *b, c = reference
print(f"a={a} b={b} c={c}") # a=1 b=[2, 3, 4] c=5
```

Cela peut s'avérer pratique, lorsque par exemple on s'intéresse seulement aux premiers éléments d'une structure :

Exemple 2

*.py

```
# si on sait que data contient prenom, nom,
# et un nombre inconnu d'autres informations
data = [ 'Jean', 'Dupont', '061234567', '12', 'rue du four', '57000',
'METZ', ]

# on peut utiliser la variable _
# ce n'est pas une variable spéciale dans le langage,
# mais cela indique au lecteur que l'on ne va pas s'en servir
prenom, nom, *_ = data
print(f"prenom={prenom} nom={nom}") # résultat : prenom=Jean nom=Dupont
```

On peut utiliser plusieurs fois la même variable dans la partie gauche de l'affectation.

Cette technique n'est utilisée en pratique que pour les parties de la structure dont on n'a que faire dans le contexte. Dans ces cas-là, il arrive qu'on utilise le nom de variable _.

Exemple 3

*.py

```
entree = [1, 2, 3]
#
_, milieu, _ = entree
print('milieu', milieu) # résultat : milieu 2
```

```
ignored, ignored, right = entree
print('right', right)   # résultat :right 3
```

Quiz

- Python List and Tuples Quiz

Pour aller plus loin ...

- Write Pythonic and Clean Code With namedtuple

1)
Fonctions toujours disponibles.