Tuples

Types construits

Qu'est-ce qu'un tuple ?

Un **tuple** est une suite ordonnée de valeurs, écrites entre parenthèses et séparées par des virgules. Comme une liste, il regroupe plusieurs valeurs dans une seule variable ; mais, contrairement à une liste, un tuple est **immuable** : on ne peut plus le modifier après sa création.

Voici trois exemples : un point, avec ses deux coordonnées ; une couleur, avec ses trois composantes (la part de rouge, la part de vert et la part de bleu) ; et un élève, avec son prénom, son âge et sa classe.

```
point = (3, 5)
couleur = (255, 128, 0)  # rouge, vert, bleu
eleve = ("Lina", 16, "1re")
print(point)
print(eleve)
```

```
(3, 5)
('Lina', 16, '1re')
```

Les valeurs d'un même tuple peuvent être de types différents : ici, des chaînes de caractères et des entiers.

En mathématiques, on parle de **couple** pour deux valeurs, de **triplet** pour trois, de quadruplet pour quatre, et plus généralement de **p-uplet** : un tuple de `p` valeurs est un p-uplet. C'est le mot qu'emploie le programme de NSI. Quand les valeurs sont repérées par un nom plutôt que par leur position, on parle de **p-uplet nommé** ; en Python, on le représente par un dictionnaire (chapitre 12).

Créer un tuple

On crée un tuple en plaçant ses éléments entre parenthèses, séparés par des virgules :

```
T = ('a', 'b', 'c')
print(f"{T} est de type {type(T)}")
```

```
('a', 'b', 'c') est de type <class 'tuple'>
```

Les parenthèses sont en réalité facultatives : c'est la virgule qui fait le tuple.

```
T = 'a', 'b', 'c'
print(f"{T} est de type {type(T)}")
```

```
('a', 'b', 'c') est de type <class 'tuple'>
```

On peut aussi créer un tuple à partir d'une autre séquence grâce à la fonction `tuple()` :

```python
T = tuple(['a', 'b', 'c'])  # à partir d'une liste
print(f"{T} est de type {type(T)}")
```

```
('a', 'b', 'c') est de type <class 'tuple'>
```

Le tuple d'une seule valeur

Un détail d'écriture piège souvent les débutants.

```python
a = (5)
print(type(a))
```

Question.
La variable `a` contient-elle un tuple ?

Réponse.

```
<class 'int'>
```

Elle ne contient pas de tuple. Entre parenthèses, `5` reste simplement le nombre 5 : les parenthèses ne servent ici qu'à grouper un calcul, comme dans `(2 + 3) * 4`.

Pour créer un tuple d'une seule valeur, il faut ajouter une **virgule** après la valeur. C'est vrai pour un nombre comme pour une chaîne de caractères :

```python
b = (5,)
print(type(b))

T = ('a')   # ceci n'est PAS un tuple, mais une chaîne
print(f"{T} est de type {type(T)}")

T = ('a',)  # ceci EST un tuple (remarquez la virgule)
print(f"{T} est de type {type(T)}")
```

```
<class 'tuple'>
a est de type <class 'str'>
('a',) est de type <class 'tuple'>
```

À retenir. C'est la virgule qui fait le tuple, pas les parenthèses. On écrit cependant toujours les parenthèses, pour la lisibilité.

Le tuple vide

On crée un tuple vide avec des parenthèses vides ou avec `tuple()` sans argument :

```
T_vide = ()
print(f"{T_vide} est de type {type(T_vide)} et de longueur {len(T_vide)}")
```

```
() est de type <class 'tuple'> et de longueur 0
```

Lire une valeur avec son indice

On lit une valeur d'un tuple exactement comme un élément d'une liste : par son **indice**, en commençant à 0. Les indices négatifs comptent à partir de la fin.

	"Lina"	16	"1re"
indice	0	1	2
indice négatif	−3	−2	−1

```
eleve = ("Lina", 16, "1re")
print(eleve[0])    # le prénom
print(eleve[2])    # la classe
print(eleve[-1])   # la dernière valeur : la classe, là encore
```

```
Lina
1re
1re
```

Question.
Que vaut `eleve[1]` ?

Réponse.
`16` : l'indice `1` désigne la deuxième valeur du tuple, l'âge de l'élève.

Un tuple ne se modifie pas

Voici la grande différence avec une liste : une fois créé, un tuple garde ses valeurs. On dit qu'il est **immuable**. À gauche, on remplace la première valeur d'une liste ; à droite, on essaie la même chose avec un tuple.

```
liste = [3, 5]
liste[0] = 7
print(liste)
```

```
[7, 5]
```

```
point = (3, 5)
point[0] = 7
```

Question.
Que se passe-t-il pour le tuple ?

Réponse.

```
TypeError: 'tuple' object does not support item assignment
```

Python refuse la modification et affiche une erreur de type. On peut lire une valeur d'un tuple, mais pas la remplacer.

Pourquoi est-ce utile ? L'immuabilité offre plusieurs avantages :

— **sécurité** : les données d'un tuple ne peuvent pas être modifiées par erreur ;

— **clé de dictionnaire** : les tuples peuvent servir de clés dans un dictionnaire (structure présentée au chapitre 12), contrairement aux listes, car les clés doivent être immuables.

Créer un nouveau tuple

Un tuple ne se modifie pas, mais on peut toujours en **créer un nouveau** à partir d'autres tuples. Le signe `+` colle deux tuples l'un après l'autre (c'est la **concaténation**), et le signe `*` répète un tuple :

```
T1 = ('a', 'b', 'c')
T2 = ('d', 'e')
print(f"T1 + T2 = {T1 + T2}")
print(f"T2 * 3 = {T2 * 3}")
print(f"T1 = {T1}, T2 = {T2}")
```

```
T1 + T2 = ('a', 'b', 'c', 'd', 'e')
T2 * 3 = ('d', 'e', 'd', 'e', 'd', 'e')
T1 = ('a', 'b', 'c'), T2 = ('d', 'e')
```

Dans les deux cas, Python crée un nouveau tuple : les tuples de départ restent intacts.

Passer d'une liste à un tuple

On passe facilement d'un type à l'autre grâce aux fonctions `tuple()` et `list()` :

```
L = [1, 2, 3, 'toto']
T = tuple(L)
print(f"Liste : {L} -> Tuple : {T}")

chaine = 'azerty'
T2 = tuple(chaine)
```

```
print(f"Chaîne : {chaine} -> Tuple : {T2}")

T3 = (10, 20, 30)
L2 = list(T3)
print(f"Tuple : {T3} -> Liste : {L2}")
```

```
Liste : [1, 2, 3, 'toto'] -> Tuple : (1, 2, 3, 'toto')
Chaîne : azerty -> Tuple : ('a', 'z', 'e', 'r', 't', 'y')
Tuple : (10, 20, 30) -> Liste : [10, 20, 30]
```

C'est ainsi qu'on fait évoluer un tuple : on le transforme en liste, on modifie la liste, puis on la transforme de nouveau en tuple.

```
notes = (12, 15, 8)
L = list(notes)
L.append(17)
notes = tuple(L)
print(notes)
```

```
(12, 15, 8, 17)
```

Longueur, appartenance et calculs

Comme pour une liste, la fonction `len()` donne le nombre de valeurs d'un tuple, et le mot-clé `in` teste si une valeur est présente :

```
T = ('pomme', 'banane', 'cerise')
print('banane' in T)
print('kiwi' in T)
print(f"Longueur : {len(T)}")
```

```
True
False
Longueur : 3
```

Les fonctions `min()`, `max()` et `sum()` fonctionnent également sur les tuples, à condition que les valeurs soient comparables ou numériques :

```
notes = (12, 15, 8, 17, 14)
print(f"Min : {min(notes)}")
print(f"Max : {max(notes)}")
print(f"Somme : {sum(notes)}")
print(f"Moyenne : {sum(notes) / len(notes)}")
```

```
Min : 8
Max : 17
```

```
Somme : 66
Moyenne : 13.2
```

Stocker les valeurs dans des variables : le déballage

Le **déballage** (en anglais *unpacking*) stocke chaque valeur d'un tuple dans une variable distincte, en une seule instruction. La première variable reçoit la première valeur, la deuxième variable la deuxième valeur, et ainsi de suite.

```python
coordonnees = (3, 7)
x, y = coordonnees
print(f"x = {x}, y = {y}")
```

```
x = 3, y = 7
```

⚠ **Attention.** il faut autant de variables à gauche que de valeurs dans le tuple, sinon Python affiche une erreur.

```python
T = (1, 2, 3)
a, b = T  # erreur : 3 valeurs mais seulement 2 variables
```

```
ValueError: too many values to unpack (expected 2)
```

Échanger deux variables

Le déballage permet aussi d'échanger les valeurs de deux variables. Sans tuple, il faut une troisième variable, appelée **variable auxiliaire**, pour garder une valeur de côté :

```python
a, b = 1, 2
aux = a
a = b
b = aux
print(f"a = {a}, b = {b}")
```

```
a = 2, b = 1
```

Ligne exécutée	a	b	aux
`a, b = 1, 2`	1	2	
`aux = a`	1	2	1
`a = b`	2	2	1
`b = aux`	2	1	1

Il a fallu trois lignes. Avec un tuple, une seule ligne suffit :

```python
a, b = 1, 2
a, b = b, a
print(f"a = {a}, b = {b}")
```

Question.
Que valent `a` et `b` après la ligne `a, b = b, a` ?

Réponse.

```
a = 2, b = 1
```

Python forme d'abord le tuple des valeurs de droite, `(b, a)`, c'est-à-dire `(2, 1)`. Il stocke ensuite ces deux valeurs dans les variables `a` et `b`, dans cet ordre.

Une fonction qui renvoie plusieurs valeurs

Grâce aux tuples, une fonction peut **renvoyer plusieurs valeurs** à la fois : il suffit de les séparer par une virgule après `return`.

```python
def min_max(L):
    """Renvoie le minimum et le maximum d'une liste."""
    return min(L), max(L)

resultat = min_max([4, 7, 2, 9, 1])
print(resultat)
```

Question.
Que contient la variable `resultat` ?

Réponse.

```
(1, 9)
```

Elle contient un tuple : la virgule du `return` a créé ce tuple de deux valeurs.

Le plus souvent, on stocke aussitôt ces valeurs dans deux variables, grâce au déballage :

```python
petit, grand = min_max([4, 7, 2, 9, 1])
print(f"Le plus petit est {petit}, le plus grand est {grand}")
```

```
Le plus petit est 1, le plus grand est 9
```

Parcourir un tuple

On parcourt un tuple exactement comme une liste, avec une boucle `for`. Première façon : le parcours par les **valeurs**.

```
jours = ("lundi", "mardi", "mercredi")

for jour in jours:
    print(f"Cours le {jour}")
```

```
Cours le lundi
Cours le mardi
Cours le mercredi
```

Tour	Valeur de jour	Affichage
1	`"lundi"`	`Cours le lundi`
2	`"mardi"`	`Cours le mardi`
3	`"mercredi"`	`Cours le mercredi`

Deuxième façon : le parcours par les **indices**, avec `range(len(jours))`, quand on a besoin de la position de chaque valeur.

```
for i in range(len(jours)):
    print(f"{i} : {jours[i]}")
```

```
0 : lundi
1 : mardi
2 : mercredi
```

Une liste de tuples

Dans la pratique, on regroupe souvent des tuples dans une liste. Ici, chaque tuple décrit un élève : son prénom et sa note. La boucle parcourt la liste, et le déballage stocke chaque tuple dans deux variables, `prenom` et `note`.

```
eleves = [("Lina", 15), ("Tom", 12), ("Sami", 17)]

for prenom, note in eleves:
    print(f"{prenom} : {note}")
```

Question.
Qu'affiche ce programme ?

Réponse.

```
Lina : 15
Tom : 12
Sami : 17
```

Tour	Tuple lu	Valeur de prenom	Valeur de note
1	`("Lina", 15)`	`"Lina"`	15
2	`("Tom", 12)`	`"Tom"`	12
3	`("Sami", 17)`	`"Sami"`	17

La liste peut s'allonger, avec `append()`, tandis que chaque tuple reste fixe :

```
eleves.append(("Inès", 14))
print(len(eleves))
```

```
4
```

Tuple ou liste ?

On choisit une **liste** pour une collection qui évolue : on lui ajoute des éléments, on en retire, on la trie. On choisit un **tuple** pour des données fixes, qui forment un tout : les coordonnées d'un point, la fiche d'un élève, les deux valeurs renvoyées par une fonction.

Tuple nommé (pour aller plus loin)

Le module `collections` propose les **namedtuple**, qui permettent d'accéder aux éléments par un nom plutôt que par un indice :

```
from collections import namedtuple

Eleve = namedtuple('Eleve', ['nom', 'prenom', 'classe'])
e = Eleve('Dupont', 'Marie', '1NSI')
print(f"{e.nom} {e.prenom} {e.classe}")
print(e[0])  # l'accès par indice fonctionne toujours
```

```
Dupont Marie 1NSI
Dupont
```

À retenir

1. Un tuple regroupe plusieurs valeurs, entre parenthèses : `point = (3, 5)`.
2. On lit une valeur par son indice, en commençant à `0`.
3. Un tuple est immuable : on ne peut pas le modifier.

4. C'est la virgule qui fait le tuple : pour une seule valeur, elle est obligatoire, `(5,)`.
5. Les fonctions `tuple()` et `list()` permettent de passer d'un type à l'autre.
6. Le déballage stocke chaque valeur dans une variable, `x, y = point`, et permet d'échanger deux variables en une ligne, `a, b = b, a`.
7. Une fonction peut renvoyer plusieurs valeurs grâce à un tuple : `return mini, maxi`.
8. On parcourt un tuple comme une liste, et une liste de tuples avec le déballage : `for prenom, note in eleves`.

Résumé : tuples vs listes

Caractéristique	Liste	Tuple
Syntaxe	`[1, 2, 3]`	`(1, 2, 3)`
Modifiable	Oui (mutable)	Non (immuable)
Clé de dictionnaire	Non	Oui
Cas d'usage	Collection évolutive	Données fixes

Exercices

Exercice 1 Parmi les instructions suivantes, lesquelles créent un tuple ? Vérifier en utilisant `type()`.

```
a = (1, 2, 3)
b = 1, 2, 3
c = (1)
d = (1,)
e = tuple([1, 2])
f = ()
```

Exercice 2 Écrire une fonction `milieu(A, B)` qui prend en paramètres deux points A et B représentés par des tuples de coordonnées `(x, y)` et qui renvoie le tuple des coordonnées du milieu du segment [AB].

Rappel. Le milieu de [AB] a pour coordonnées $\left(\frac{x_A + x_B}{2} \ ; \ \frac{y_A + y_B}{2}\right)$.

```
# Exemple d'appel attendu :
print(milieu((1, 2), (5, 8)))  # doit afficher (3.0, 5.0)
```

Exercice 3 Écrire une fonction `statistiques(notes)` qui prend un tuple de notes et renvoie un tuple contenant dans cet ordre : la note minimale, la note maximale et la moyenne.

```
# Exemple d'appel attendu :
print(statistiques((12, 15, 8, 17, 14)))  # doit afficher (8, 17, 13.2)
```

Exercice 4 On dispose d'une liste de tuples représentant des élèves sous la forme `(nom, moyenne)`. Écrire une fonction `meilleur_eleve(eleves)` qui renvoie le nom de l'élève ayant la meilleure moyenne.

```python
classe = [('Alice', 14.5), ('Bob', 12.0), ('Charlie', 16.5), ('Diana', 15.0)]
print(meilleur_eleve(classe))  # doit afficher Charlie
```

Exercice 5 Écrire une fonction `frequences(texte)` qui prend une chaîne de caractères et renvoie une liste de tuples `(caractère, nombre_d_occurrences)`, dans l'ordre de première apparition des caractères. On ignorera les espaces.

```python
# Exemple d'appel attendu :
print(frequences("abracadabra"))
# doit afficher [('a', 5), ('b', 2), ('r', 2), ('c', 1), ('d', 1)]
```