NSI en première (2019-2020)
Types construits 1

1 P-uplets

1.1 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 $\star$. 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
```

1.2 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.

1.3 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
```

2 Listes

2.1 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)
```

2.2 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]]]
```

2.3 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

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. Avec cette copie superficielle, une nouvelle liste est créée avec une nouvelle adresse. Les éléments gardent eux la même adresse.

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"]]
```

2.4 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()
```