Séance 2: Données composites et mutabilité

Philipp Ahrendt

November 28, 2025

Contents

1 Données composites

A part les types basiques `int`, `float`, `bool` et `str`, python donne plusieurs manières d'en construire des structures de données plus complexes, comme par exemple des **listes**. En général, ces structures sont définies par des champs contenant des objets de type arbitraire.

Voici les types composites les plus importants:

Type Python	**Description**	Notation
`tuple`	n-uplets	`(a,b,..)`
`list`	listes ordonnées	`[a,b,..]`
`dict`	dictionnaires	`{cle1 : a, cle2 : b, cle3 : ..}`

Ici, `a,b,..` sont des objets de types arbitraires, et potentiellement différentes.

Pour chacun de ces types, il existe aussi un objet "vide", qui ne contient aucun élément. L'objet vide est noté, respectivement, `()`, `[]`, et `{}`.

1.1 Tuples

Les tuples sont souvent utilisés implicitement dans des constructions idiomatiques de python.

Exemple: Fonction avec plusieurs sorties

```
def quotient_et_reste(a,b):
    return a//b,a%b

paire = quotient_et_reste(11,3)

q,r = quotient_et_reste(9,4)

print(q,r,paire)
```

Exemple: Échanger deux variables

```
# Première tentative
a = 0
b = 1

a = b
b = a

print("a:",a)
print("b:",b)


# Deuxième tentative
a = 0
b = 1

x = a

a = b
b = x

print("a:",a)
print("b:",b)
```

```
# Troisième tentative
a = 0
b = 1

a,b = b,a

print("a:",a)
print("b:",b)
```

1.2 Listes

Les listes sont la structure principale pour opérer sur des séquences ordonnées en python.

```
# Syntaxe de base pour les listes
L = [1,2,3,4,5]

n = len(L)
print(n)

a = L[0]
b = L[1]
c = L[2]
d = L[3]
e = L[4]

print(a,b,c,d,e)

# Les indices sont "cycliques"
last = L[-1]
print(last)

# Opérations plus complexes
L2 = L[1:4]
L3 = L[1:5:2]

print(L2)
print(L3)
```

Cette syntaxe est celle que l'on utilise aussi pour les tuples (et les chaînes de caractères).

```
# Syntaxe de base pour les tuples
t = (1,2,3,4,5)

n = len(t)
print(n)

a = t[0]
b = t[1]
c = t[2]
d = t[3]
e = t[4]

print(a,b,c,d,e)

# Les indices sont "cycliques"
last = t[-1]
print(last)

# Opérations plus complexes
t2 = t[1:4]
t3 = t[1:5:2]

print(t2)
print(t3)


# Syntaxe de base pour les chaines de caracteres
s = "abcde"

n = len(s)
print(n)

a = s[0]
b = s[1]
c = s[2]
d = s[3]
```

```
e = s[4]

print(a,b,c,d,e)

# Les indices sont "cycliques"
last = s[-1]
print(last)

# Opérations plus complexes
s2 = s[1:4]
s3 = s[1:5:2]

print(s2)
print(s3)
```

On peut aussi tester si un élément est dans une liste, avec le mot-clé `in`. Cela marche aussi pour les chaînes de caractères et les tuples.

```
def fun(L1,L2):
   for e in L1:
      if e in L2:
         print(e,"est aussi dans L2")

fun([2,4,6,8,10],[1,4,5,6,9,10])


# fonction equivalente
def fun(L1,L2):
   for i in range(len(L1)):
      if L[i] in L2:
         print(L[i],"est aussi dans L2")

fun([2,4,6,8,10],[1,4,5,6,9,10])
```

Par rapport aux tuples, python fournit quelques opérations en plus sur les listes, permettant de les modifier (voir section sur la mutabilité).

Méthode	Description	Type de sortie
`L.append(a)`	Rajouter `a` à la fin de `L`	`None`
`L.pop()`	Enlever le dernier élément `t` de `L` et le donner en sortie	`type(t)`
`L.extend(L2)`	Concaténer `L2` à la fin de `L`	`None`
`L.remove(a)`	Si `a` est dans la liste, enlever sa première apparence	`None`
`del(L[i])`	Enlever le `i`-ème élément de `L`	`None`

Un exemple d'utilisation de listes est pour écrire des fonctions avec un nombre variable de sorties.

Exemple: Pour un entier n, trouver les nombres premiers plus petits que n.

```
def premiers(n):

   premiers = []
   candidat = [True for i in range(n+1)]

   for i in range(2,n+1):
      if candidat[i]:
         premiers.append(i)
         for m in range(2,n//i+1):
            candidat[m*i] = False

   return premiers

print(premiers(30))
```

1.3 Dictionnaires

Dans une liste, chaque entrée est associée à un nombre entre 0 et la longueur de la liste. Il y a des situations où on a intérêt à ordonner autrement nos données. Par exemple, après un examen, on peut vouloir associer à chaque étudiant une note.

Une première solution est d'utiliser plusieurs listes.

```
etudiants = ["Marc", "Alice", "Julien", "Grégoire"]
notes = [8,14,11,19]

def note(etudiant, liste_etudiants, liste_notes):
    i = liste_etudiants.index(etudiant)
    return liste_notes[i]
```

```
a = note("Alice", etudiants, notes)

print(a)
```

Cela fonctionne, mais ce n'est pas adapté à la situation. Pour trouver ou modifier la note d'un étudiant, il faut trouver sa place dans la liste, et ensuite chercher cette place dans une autre liste. Cela devient vite compliqué si on a beaucoup de données reliées.

Une manière plus efficace serait de relier directement un étudiant à sa note. On peut faire cela avec un **dictionnaire**.

Un dictionnaire est un collection de paires de **clés** et valeurs. Voici un dictionnaire associé au code précédent

Clé	**Valeur**
Marc	8
Alice	14
Julien	11
Grégoire	19

En python, on peut réécrire le code précédent avec un dictionnaire.

```
notes = {
   "Marc": 8,
   "Alice": 14,
   "Julien": 11,
   "Grégoire": 19,
}

a = notes["Alice"]
print(a)
```

En termes de manipulation, les dictionnaires sont similaires aux listes, mais avec des petites différences.

```
#Opérations de base sur un dictionnaire
notes = {
   "Marc": 8,
   "Alice": 14,
   "Julien": 11,
```

```
   "Grégoire": 19,
}

n = len(notes)
print(n)

notes["Grégoire"] = 17
notes["Caroline"] = 20

del(notes["Julien"])
print(notes)

# On peut itérer sur les clés du dictionnaire, avec dict.keys()
def admis(notes):
   liste_admis = []
   for etudiant in notes.keys():
      if notes[etudiant] >= 10:
         liste_admis.append(etudiant)

   return liste_admis

liste_admis = admis(notes)
print(liste_admis)

# On peut aussi accéder aux valeurs, avec dict.values()

def ratrappages(notes):
   for i in range(10):
      if i in notes.values():
         print("Il faut une session de ratrappage")
         return

   print("Tout le monde a validé!")

ratrappages(notes)
```

2 Mutabilité

Sur les listes et dictionnaires, on a utilisé des opérations qui en modifient des entrées, rajoutent et suppriment des éléments, etc. Ces opérations n'ont pas de "vrai" équivalent sur les tuples ou les chaînes de caractères.

Par exemple, testez le code suivant:

```
t = (1,2,3,4)

t[0] = 5
```

La différence est que les listes et dictionnaires sont des objets **mutables**, et les tuples et chaînes de caractères sont **non mutables**.

Type	Mutable/Non Mutable
`str`	Non mutable
`tuple`	Non mutable
`list`	Mutable
`dict`	Mutable

Un objet non mutable ne peut pas être modifié. Si on veut modifier une variable affectée à un objet non mutable, il faut construire un nouvel objet et réaffecter la variable à cet objet. En revanche, une opération de mutation sur un objet mutable garde l'affectation de la variable, et change directement l'objet sous-jacent.

Voici un exemple pour illustrer la différence:

```
# Sur un objet non mutable: réaffectation
s = "Bonjour"

s_test = s

s = s + " à tous"

print(s)
print(s_test)

# Sur un objet mutable: mutation
L = [1,2,3,4]

L_test = L
```

```
L[0] = 5

print(L)
print(L_test)
```

Le fait qu'on manipule des objets mutables peut avoir des conséquences inattendues.

```
L = [0 for i in range(5)]
M = [L for j in range(5)]

M[2][3] = 1

print(M)

def enleve_doublons(L1,L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [1,2,3,4]
L2 = [1,2,5,6]

enleve_doublons(L1,L2)
print(L1)
```

Dans des cas comme ci-dessus, il est utile de faire des copies d'un objet. La copie est un autre objet en mémoire, donc elle n'est pas modifiée si on modifie l'objet initial, et vice-versa.

```
L = [1,2,3,4]

L_copie = L[:]

L[0] = 5

print(L)
print(L_copie)
```