Informatique et Calcul Scientifique

Tuples, dictionnaires

09.10.2024

La fois passée, on a vu...

- Les listes en Python
- Les opérations sur les listes : ajouter/enlever des éléments, lire/modifier des éléments, calculs sur les listes...
- Compréhension de liste
- Copies superficielles de listes et mutabilité

Aujourd'hui, on verra...

Deux autres types de structure de données en Python :

- Les tuples
- Les dictionnaires.

Tuples

Un **tuple**, comme une liste, permet de regrouper et d'indexer des données.

A la différence d'une liste, un tuple est un objet **non mutable** ! On peut donc

- Créer un tuple
- Accéder à ses éléments en mode lecture uniquement.

On ne peut ni modifier un élément d'un tuple, ni lui ajouter/enlever des éléments.

On peut cependant effectuer sur un tuple toutes les opérations qu'on a vues sur les listes et qui ne les modifient pas.

Créer un tuple

Un tuple est déclaré en utilisant des parenthèses rondes `()`.

- L'instruction `t = (0, 2, 4, 6)` crée un tuple contenant les éléments 0, 2, 4 et 6 et l'affecte à la variable `t`.
- Pour créer un tuple vide : `t = ()`
- Pour créer un tuple à un élément : `t = (1,)` par exemple.

```
# sans virgule, on affecte
# a t la valeur 1
t = (1)
print(t, type(t))

# avec virgule, on affecte
# a t le tuple contenant 1
t = (1,)
print(f"{t} est un {type(t)}\
de taille {len(t)}")
```

Output :

```
1 <class 'int'>
(1,) est un <class 'tuple'> de
taille 1
```

Créer un tuple

On peut créer un tuple à partir de n'importe quel autre objet itérable : une liste, un autre tuple, un range, une chaîne de caractères...

```
t1 = tuple(range(1, 8, 2))
t2 = tuple([1, 3, 5, 7])
t3 = tuple("1357")
print(t1)
print(t2)
print(t3)
```

Output :

```
(1, 3, 5, 7)
(1, 3, 5, 7)
('1', '3', '5', '7')
```

Calculs sur les tuples

Toutes les opérations qui ne modifient pas les listes s'appliquent aussi pour les tuples.

- `len(t)` donne le nombre d'éléments du tuple `t`
- Les éléments de `t` sont indexés de `0` à `len(t)-1` (ou de `-1` à `-len(t)` depuis la fin), `t[i]` est l'élément `i`
- `max(t)`, `min(t)`, `sum(t)` donnent le maximum, minimum, et la somme des éléments de `t` si ces quantités sont bien définies
- `t.count(x)`, `t.index(x)` donnent le nombre d'ocurrences / l'index de la première occurrence de la valeur `x` dans le tuple `t`
- `t1 + t2`, `t1 * n`, `n * t1` créent un nouveau tuple résultant de la concaténation de `t1` avec `t2` ou de `t1` avec lui-même `n` fois
- Les expressions booléennes `x in t` et `x not in t` permettent de vérifier l'appartenance d'un élément de valeur `x` au tuple `t`

Parcourir un tuple

Un tuple étant un objet itérable, on peut le parcourir à l'aide d'une boucle `for` .

- On peut itérer sur ses éléments...

```
t = (1, 3, 5, 7)

for x in t:
    print(x+1)
```

```
Output :
2
4
6
8
```

- ... ou sur ses indices

```
t = (1, 3, 5, 7)

for i in range(len(t)):
    print(i+1, t[i]+1)
```

```
Output :
1 2
2 4
3 6
4 8
```

Un tuple est immutable

Une fois qu'un tuple a été créé, on ne peut plus réassigner un de ses éléments :

```
t = (1, "abc")
t[1] = "cde"
```

Output :

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

- Mais un tuple, comme une liste, contient des références à des objets en mémoire, et peut donc contenir une référence à un objet mutable, par exemple une liste, et cet objet peut changer !

```
t = ([1, 2], "abc")
t[0].append(3)
print(t)
```

Output :

```
([1, 2, 3], 'abc')
```

Différences entre une liste et un tuple

La grande différence entre une liste et un tuple est que le premier est mutable et le second immutable. Ceci a deux conséquences principales :

1. Un tuple prend moins de place en mémoire qu'une liste. Les opérations effectuées sur celui-ci (lecture, itération) sont ainsi plus rapides que sur les listes. La différence n'est cependant pas très importante pour ce qui nous intéresse.
2. La liste pouvant être modifiée, on l'utilisera plutôt pour stocker des données qui sont susceptibles d'évoluer au cours du programme.On utilisera plutôt un tuple pour transporter des informations d'un bout à l'autre de notre code en s'assurant que celles-ci restent inchangées.

Packing, unpacking

On peut simultanément affecter à une variable plusieurs valeurs qui seront "emballées" dans un tuple. C'est l'opération de **packing**.

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

Output :

```
(1, 2, 3)
```

De même, on peut affecter un tuple de taille `n` à `n` variables simultanément : le tuple sera "déballé" en `n` éléments qui seront affectés aux `n` variables. C'est l'opération de **unpacking**

```
x, y, z = t
print(x, y, z, type(x))
```

Output :

```
1 2 3 <class 'int'>
```

Derrière une affectation multiple se cachent donc une opération de packing et de unpacking.

```
x, y, z = 1, 2, 3
print(x, y, z)
```

Output :

```
1 2 3
```

Packing, unpacking

Lorsqu'une fonction retourne plusieurs valeurs, elle retourne en fait un tuple contenant ces valeurs.

- Si une fonction retourne `n` valeurs, on peut donc directement affecter à `n` variables les valeurs retournées par l'appel de la fonction.

```
def aire_perim_rectangle(x, y):
    aire = x * y
    perimetre = 2 * x + 2 * y
    return aire, perimetre

t = aire_perim_rectangle(2, 3)
print(t, type(t))

a, p = aire_perim_rectangle(2, 3)
print(a, p)
```

Output :

```
(6, 10) <class 'tuple'>
6 10
```

Dictionnaires

Lors de l'écriture de programmes, on peut être amené à manipuler des objets plus complexes. On aimerait par exemple associer un "mot" avec une certaine valeur. Souvenez-vous des arguments par mot-clés !

Une telle structure de données dite par correspondance existe en Python. Il s'agit de **dictionnaires**.

- Un dictionnaire (`dict`) est une structure de données qui permet d'indexer des objets (les valeurs) non pas avec des entiers $0, 1, 2, \cdots$ mais avec des **clés**.
- Il s'agit d'un ensemble de paires `dict = {clé: valeur}` .

```python
dict_007 = {
    "nom": "Bond",
    "prenom": "James",
    "dob": 1920,
}
print(dict_007["dob"])
```

Output :

1920

Dictionnaires

- On peut penser à un dictionnaire de langues : il contient des paires (mot, définition). On peut accéder à la définition en cherchant le mot correspondant dans le dictionnaire.
- Comme un dictionnaire de langues, un dictionnaire Python stocke des valeurs, ou des informations (les définitions) indexées par des clés uniques (les mots).

```
d = {
    "pommier": "arbre qui porte des
                        pommes",
    "lune": "satellite de la Terre",
    "avion": "engin volant",
}

print(d["avion"])
```

Output :

engin
volant

Opérations sur les dictionnaires

Comme les listes, les dictionnaires sont des objets **mutables**.

On peut donc :

- Créer un dictionnaire
- Effectuer des opérations de lecture :
  - Lire un élément d'un dictionnaire
  - Parcourir les clés/valeurs d'un dictionnaire
  - ...
- Effectuer des opérations d'écriture :
  - Modifier un élément d'un dictionnaire
  - Ajouter/enlever un élément d'un dictionnaire
  - ...

Créer un dictionnaire

On peut créer un dictionnaire :

- vide, avec une paire d'accolades vide : `d = {}`
- contenant déjà des paires (clé, valeur) avec la syntaxe

```
d = {clé_1:valeur_1, clé_2:valeur_2, ..., clé_n:valeur_n,}
```

Pour la lisibilité on peut aussi introduire des retours de ligne :

```
d = {
    cle_1: valeur_1,
    cle_2: valeur_2,
    ...
    cle_n: valeur_n,
}
```

. La virgule après le dernier élément du dictionnaire est optionnelle mais c'est une bonne pratique de l'inclure.

Créer un dictionnaire

- Les clés peuvent être n'importe quel objet non mutable [1].

```
d = {
    1: "poire",
    "deux": "pomme",
    3.0: "prune",
    (4,5): "fraise",
}
```

```
d = {[1, 2]: 3}
```

Output :

```
TypeError: unhashable type: 'list'
```

- Les clés doivent être uniques. Si on définit plusieurs paires (clé, valeur) avec la même clé, seule la dernière sera gardée.

```
d = {1:123, 2:123, 1:345}
print(d)
```

Output :

```
{1: 345, 2: 123}
```

1. En fait, n'importe quel objet **hashable**. Cette contrainte est liée à la manière dont les dictionnaires sont stockés en mémoire en Python. Pour plus d'info voir la documentation Python.

Accéder à un élément

On peut accéder à l'élément de clé `key` du dictionnaire `d` avec la syntaxe `d[key]`, si un tel élément existe :

```
d_fruits = {
    10: "poire",
    20: "pomme",
    30: "prune",
}
print(d_fruits[10])
```

Output :

poire

S'il n'y a pas d'élément de clé `key` dans `d`, `d[key]` produira une erreur :

```
print(d_fruits[5])
```

Output :

KeyError: 5

Accéder à un élément

Pour éviter une erreur si l'élément recherché n'existe pas dans le dictionnaire, on peut exécuter l'instruction `d.get(key)`.

La méthode `get` de la classe `dict` retourne l'élément de clé `key` si un tel élément existe, et retourne `None` autrement.

```
d_fruits = {
    10: "poire",
    20: "pomme",
    30: "prune",
}
print(d_fruits.get(10))
print(d_fruits.get(5))
```

Output :

```
poire
None
```

Insérer/modifier un élément

L'instruction `d[key] = val` permet :

- s'il n'existe pas d'élément de clé `key` dans le dictionnaire `d`, d'insérer la paire `(key, val)`
- s'il existe un élément de clé `key` dans le dictionnaire `d`, de modifier sa valeur à `val`.

```
d_fruits = {}
d_fruits[10] = "poire"
print(d_fruits)
d_fruits[20] = "pomme"
print(d_fruits)
d_fruits[10] = "fraise"
print(d_fruits)
```

Output :

```
{10: 'poire'}
{10: 'poire', 20: 'pomme'}
{10: 'fraise', 20: 'pomme'}
```

Supprimer un élément

L'instruction `del d[key]` supprime l'élément de clé `key` du dictionnaire `d`

```
d = {10: "poire", 20:"pomme"}
del d[10]
print(d)
```

Output :

{20: 'pomme'}

- S'il n'existe pas d'élément de clé `key` dans `d`, `del d[key]` produit une erreur.

```
del d[3]
```

Output :

KeyError: 3

Appartenance

On peut vérifier l'appartenance d'une clé à un dictionnaire avec le mot-clé `in` :

```
d = {
    "pomme": "malus domestica",
    "poire": "pyrus communis",
    "fraise": "fragaria x ananassa",
}

print("pomme" in d)
print("mangue" not in d)
```

Output :

```
True
True
```

Parcourir un dictionnaire

Un dictionnaire est un objet **itérable**. Comme pour les ranges, listes, strings, tuples, on peut itérer sur ses **clés** avec une boucle `for` .

- Les éléments du dictionnaire sont alors parcourus dans le même ordre que celui dans lequel il a été rempli.

```
for x in d:
    print(x)
```

Output :

```
pomme
poire
fraise
```

Parcourir les clés d'un dictionnaire nous donne accès également aux valeurs associées.

```
for x in d:
    print(f"d[{x}] = {d[x]}")
```

Output :

```
d[pomme] = malus domestica
d[poire] = pyrus communis
d[fraise] = fragaria ×
ananassa
```

Parcourir un dictionnaire

On peut aussi parcourir les clés, les valeurs ou les paires (clé, valeur) d'un dictionnaire `d` en itérant respectivement sur les vues[2] `d.keys()`, `d.values()` ou `d.items()`.

```
d_fruits = {
    10: "poire",
    20: "pomme",
    30: "prune",
}

for x in d_fruits.keys():
    print(x)
print()

for x in d_fruits.values():
    print(x)
```

Output :

```
10
20
30

poire
pomme
prune
```

2. Vues en Python

Parcourir un dictionnaire

Itérer sur la vue `d.items()` revient à itérer sur des tuples.

```
for x in d_fruits.items():
    print(x)

print()

# tuple unpacking
for x, y in d_fruits.items():
    print(x, y)
```

Output :

```
(10, 'poire')
(20, 'pomme')
(30, 'prune')

10 poire
20 pomme
30 prune
```

Créer un dictionnaire à partir d'une liste ou d'un tuple

L'instruction `dict(L)` crée un dictionnaire à partir d'une liste `L` de paires (de tuples de taille 2) :

```
L = [(1,"a"), (2,"b"), (3, "c")]
d = dict(L)
print(d)
```

Output :

```
{1: 'a', 2: 'b', 3: 'c'}
```

De même on peut créer un dictionnaire à partir d'un tuple de paires (donc un tuple de tuples) :

```
t = ((1,"a"), (2,"b"), (1, "c"))
d = dict(t)
print(d)
```

Output :

```
{1: 'c', 2: 'b'}
```

Compréhension de dictionnaire

Comme pour les listes, on peut créer de manière compacte un dictionnaire en compréhension.

```
d = {cle:val for element in ensemble_depart if condition}
```

Les deux programmes ci-dessous créent le même dictionnaire :

```
d = {1: 1, 2: 4, 3: 9, 4: 16}
```

```
d = {}
for x in range(1, 5):
    d[x] = x**2
```

```
d = {x : x**2 for x in range(1, 5)}
```

Compréhension de dictionnaire

Les deux programmes ci-dessous créent le même dictionnaire. Lequel ?

```
L = [10, 20, 30]
t = ("a", "b", "c")

d = {}
for x in L:
    for y in t:
        d[x] = y
```

```
L = [10, 20, 30]
t = ("a", "b", "c")

d = {x:y for x in L for y in t}
```

Compréhension de dictionnaire

Les deux programmes ci-dessous créent le même dictionnaire

```
d = {10: 'a', 20: 'b', 30: 'c'}
```

```
L = [10, 20, 30]
t = ("a", "b", "c")

d = {}
for i in range(3):
    d[L[i]] = t[i]
```

```
L = [10, 20, 30]
t = ("a", "b", "c")

d = {L[i]:t[i] for i in range(3)}
```

Application : dictionnaires imbriqués

Que se passe-t-il si on veut stocker des données plus complexes ? Il est possible de créer un dictionnaire dont les valeurs sont elles-mêmes des dictionnaires. On parle alors de **dictionnaire imbriqué**.

Celui-ci prend la forme suivante :

```
dict_imbrique = {
    "dictA" : {"cle_1A": valeur_1A , "cle_2A" : valeur_2A},
    "dictB" : {"cle_1B": valeur_1B , "cle_2B" : valeur_2B},
}
```

Remarquons que les clés de chaque dictionnaire (`cle_1A` , `cle_1B`) ne sont pas nécessairement les mêmes.

Dictionnaires imbriqués : exemple

Prenons l'exemple d'un hôpital. Celui-ci stocke les données de chaque patient, ainsi que les tests que celui-ci a effectués.

Pour chaque nouveau patient, on créera un dictionnaire `patient1`. Dans celui-ci, on stockera dans le dictionnaire `informations` toutes les informations le concernant. On associe ce dictionnaire en tant que valeur du mot-clé `infos`.

```
patient1 = {}
informations = {'name':'Alice', 'DOB': '27.06.1992',
                'weight': 56, 'sex' : 'F'}
patient1["infos"] = informations

print(patient1)
```

Ce code produira le texte suivant :

```
patient1 = {'infos': {'name': 'Alice', 'DOB':
'27.06.1992', 'weight': 56, 'sex': 'F'}}
```

Dictionnaires imbriqués : exemple

Admettons qu'on mesure la fréquence cardiaque ainsi que le taux d'oxygène de cette personne toutes les minutes pendant 5 minutes. On stocke ces résultats dans deux dictionnaires : HR et o2, associés aux valeurs `Heart rate` et `Oxygen level`.

```
time = [0,1,2,3,4,5]
freq = [88, 90, 94, 82, 86, 85]
o2_level = [96, 97, 96, 96, 97, 96]

HR = {"time":time, "frequency": freq}
o2 = {"time":time, "O2": o2_level}
```

Pour simplifier le stockage des résultats, on crée un dictionnaire `measurements` qui contient HR et o2.

```
measurements = {"Heart rate": HR, "Oxygen level" : o2,}
```

La commande `print(measurements)` aura donc comme output :

```
measurements = {'Heart rate': {'time': [0, 1, 2, 3, 4, 5],
'frequency': [88, 90, 94, 82, 86, 85]}, 'Oxygen level': {'time': [0,
1, 2, 3, 4, 5], 'O2': [96, 97, 96, 96, 97, 96]}}
```

Dictionnaires imbriqués : exemple

Finalement, on peut inclure le dictionnaire `measurements` dans le dictionnaire patient1. On a ainsi trois niveaux de dictionnaires imbriqués les uns dans les autres.

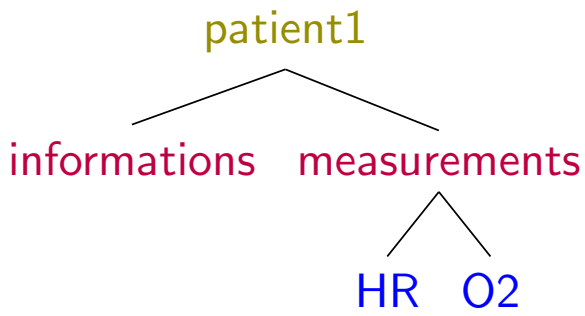
```
patient1["tests"] = measurements

print(patient1)
```

```
patient1 = {'infos': {'name': 'Alice', 'DOB':
'27.06.1992', 'weight': 56, 'sex': 'F'},'tests': {'Heart
rate': {'time': [0, 1, 2, 3, 4, 5], 'frequency': [88,
90, 94, 82, 86, 85]},'Oxygen level': {'time': [0, 1, 2,
3, 4, 5], 'O2': [96, 97, 96, 96, 97, 96]}}}
```

Dictionnaires imbriqués : exemple

Voici ce que donne la situation schématiquement :



Bien entendu, pour un nouveau patient on créera un dictionnaire `patient2` que l'on placera dans un nouveau dictionnaire `hopital` qui se situera à un niveau hiérarchique supérieur.

Pour accéder aux informations du patient 1 :

```
print(patient1['infos']['name'])
print(patient1['tests']['Heart rate'])
```

```
Output :
Alice
{'time':
[0, 1, 2, 3, 4, 5],
'frequency':
[88, 90, 94, 82, 86,
85]}
```

De retour aux fonctions

- Il est possible de définir une fonction qui prend un nombre variable d'arguments en entrée, comme la fonction `print`.
- La syntaxe pour définir une fonction qui prend un nombre variable d'arguments est la suivante :

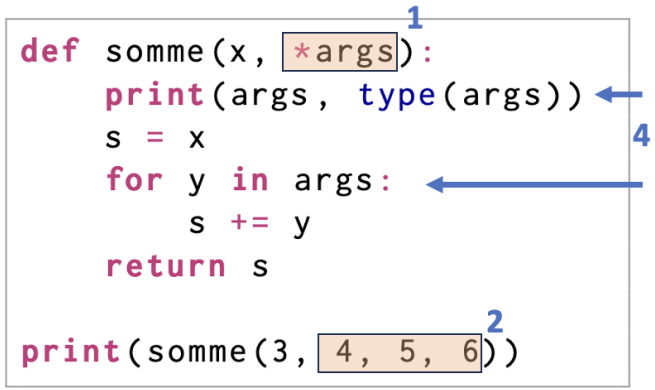
```
def somme(x, *args):
    print(args, type(args))
    s = x
    for y in args:
        s += y
    return s

print(somme(3, 4, 5, 6))
```

Output :

```
(4, 5, 6) <class 'tuple'>
18
```

Fonctions avec un nombre variable d'arguments



1. Dans la définition de la fonction, après les paramètres obligatoires, on définit un paramètre précédé d'un symbole `*`
2. Dans l'appel de la fonction, on fournit un argument pour chaque paramètre obligatoire, puis encore autant d'arguments qu'on veut
3. Au moment de l'appel, l'interpréteur Python emballe (packing) les arguments restants dans un tuple et c'est ce tuple qui est passé à la fonction
4. Dans la fonction, on a donc accès à ce tuple `args` et on peut itérer sur ses éléments.
   Remarque : `args` n'est au final qu'un *nom* de variable.

Fonctions avec un nombre variable d'arguments nommés

On peut de même définir une fonction avec un nombre variable d'arguments **nommés** avec la syntaxe suivante :

```
def somme(x, **kwargs):
    print(kwargs, type(kwargs))
    s = x
    for v in kwargs.values():
        s += v
    return s

print(somme(3, y = 4, z = 5))
```

```
Output :
{'y': 4, 'z': 5} <class
'dict'>
12
```

Fonctions avec un nombre variable d'arguments

- Dans la définition de la fonction, après les paramètres obligatoires, on définit un paramètre précédé d'un symbole `**`
- Dans l'appel de la fonction, on fournit un argument pour chaque paramètre obligatoire, puis encore autant d'arguments nommés qu'on veut
- Au moment de l'appel, l'interpréteur Python emballe (packing) les arguments restants dans un **dictionnaire** qui est passé à la fonction
- Dans la fonction, on a donc accès à ce dictionnaire et on peut itérer sur ses clés, ou sur les vues `items()` ou `values()` .