Course: Python language

Ali ZAINOUL <ali.zainoul.az@gmail.com>

for IBM - Needemand
July 17, 2024

1 Types de données modifiables

- Listes (list)
  - Constructeur
  - Indiçage
  - Itération
  - Opérateurs +, * et in
  - Méthodes `append()`, `insert()`, fonction `del()`, `sort()`, `reverse()`, `remove()`, `extend()`, `pop()`, `clear()`
  - Copie et références
- Fonction `sorted()`
- Principe de fonctionnement des objets itérables
- Fonctions `reversed()` et `range()`
  - Fonction reversed()
  - Fonction range()
- Dictionnaires (dict)
  - Constructeur
  - Indiçage
  - Opérateur `in`, fonction `del()`
  - Méthodes `keys()`, `values()`, `items()`, `update()`, `get()`

- Copie et références
- Set (set)
  - Constructeur
  - Opérateurs - |, &, et ^
- Résumé complet sur les structures de données

2 Structures conditionnelles et répétitives
- Les structures conditionnelles `if/elif/else`
- Les boucles d'itérations while et for
- Instructions break et continue
- Fonction enumerate()
- Bloc else sur structure répétitive
- Notion d'intension (Comprehension) List - Ensemble - Dict
  - Notion d'intension (Comprehension)
  - Liste en intension (comprehension list)
  - Ensemble en intension (comprehension set)
  - Dictionnaire en intension (comprehension dict)
  - Tuple en intension (comprehension tuple)

** Types de données modifiables **

Listes - Constructeur

- En Python, les listes peuvent être créées à l'aide du constructeur `list()` ou `[]` en spécifiant les éléments à inclure dans la liste. Les éléments peuvent être de n'importe quel type de données.
- **Exemple en Python**:

```python
# Example of list constructor in Python
l = list([1, 2, 3, 4, 5])
print("List:", l)  # Output: List: [1, 2, 3, 4, 5]
anotherl = [1, 2, 3, 4, 5]
print("List:", anotherl)  # Output: List: [1, 2, 3, 4, 5]
```

Listes - Indiçage

- Les listes peuvent être indéxées pour accéder à leurs éléments individuels. L'indiçage commence à partir de 0 pour accéder aux éléments de la liste.
- **Exemple d'indiçage en Python:**

```python
# Example of indexing in list in Python
l = [1, 2, 3]
print("First element:", l[0])  # Output: First element: 1
print("Last element:", l[-1])   # Output: Last element: 3
```

Listes - Itération

- Les listes peuvent être itérées à l'aide d'une boucle `for`, ce qui permet d'accéder à chaque élément de la liste.
- **Exemple d'itération en Python:**

```
# Example of iteration over list in Python
l = [1, 2, 3]
# foreach loop-like
for element in l:
    print(element)
# for loop \textit{via} indexes
print("Traditional for loop:")
for i in range(len(l)):
    print("Element at index ", i, ": ", l[i])
```

Listes - Opérateurs +, * et in

- Les listes supportent les opérateurs + et *, ainsi que l'opérateur `in` pour la vérification d'appartenance.
- **Exemple d'opérateurs en Python:**

```python
# Example of operators in list in Python
l1 = [1, 2, 3]
l2 = [4, 5, 6]
print("Concatenated list:", l1 + l2)
# Output: Concatenated list: [1, 2, 3, 4, 5, 6]
print("Repeated list:", l1 * 2)
# Output: Repeated list: [1, 2, 3, 1, 2, 3]
print("Check if 2 in list:", 2 in l1)
# Output: Check if 2 in list: True
print(*l1)
# Output: 1 2 3
print(*l2)
# Output: 4 5 6
```

Listes - Méthodes et fonctions

- Les listes prennent en charge diverses méthodes pour la manipulation des éléments, y compris l'ajout, l'insertion, la suppression, le tri, l'inversion, etc.
- Voici la Documentation officielle des méthodes de la classe List.

Listes - Méthodes et fonctions - Exemple

- **Exemple de méthodes en Python:**

```
# Example of methods in list in Python
l = [1, 2, 3, 4]
l.append(5)              # Output: [1, 2, 3, 4, 5]
l.insert(2, 10)          # Output: [1, 2, 10, 3, 4, 5]
del l[1]                 # Output: [1, 10, 3, 4, 5]
l.sort()                 # Output: [1, 3, 4, 5, 10]
l.reverse()              # Output: [10, 5, 4, 3, 1]
l.remove(3)              # Output: [10, 5, 4, 1]
l.extend([6, 7, 8])      # Output: [10, 5, 4, 1, 6, 7, 8]
l.pop()                  # Output: [10, 5, 4, 1, 6, 7]
l.clear()                # Output: []
```

Listes - Copie et références

- Les listes peuvent être copiées d'une manière **superficielle** en utilisant l'opérateur d'assignation =, via l'opérateur d'indiçage :, ou encore la méthode `copy()`, tandis que les copies **en profondeur** nécessitent la fonction `deepcopy()` du module `copy`.
- **Exemple de copies en Python:**

```
1 # Example of copying lists in Python
2 import copy # for function deepcopy()
3 l1 = [1, 2, 3]
4 l2 = l1
5 l3 = l1[:]
6 l4 = l1.copy()
7 l5 = copy.deepcopy(l1)
```

Listes - Copie et références - Exemple

```python
import copy

# Creating an original list
l1 = [1, 2, 3]

# Copies of the original list
l2 = l1
l3 = l1[:]
l4 = l1.copy()
l5 = copy.deepcopy(l1)
lists = [l1, l2, l3, l4, l5]

def print_lists():
    for list in lists:
        print(list)

# Before modification of l1
print("# Before modification of l1")
print_lists()

# Modifying l1
l1.append(4)

# After modification of l1
print("# After modification of l1")
print_lists()
```

Listes - Différence entre `copy` et `deepcopy`

- La différence entre les méthodes `copy` et `deepcopy` réside dans la manière dont elles traitent les objets imbriqués lors de la copie de listes en Python.
- La méthode `copy` effectue une copie superficielle, créant une nouvelle liste mais partageant les objets imbriqués avec l'original. En revanche, la fonction `deepcopy` réalise une copie en profondeur, dupliquant tous les objets imbriqués pour créer une copie indépendante de l'original.
- Cet exemple illustre les propos discutés.

Fonction `sorted()`

- La fonction `sorted()` peut être utilisée pour trier une liste et renvoyer une nouvelle liste triée.
- **Exemple de tri en Python:**

```
# Example of sorting list in Python
l = [3, 1, 2, 5, 4]
sorted_list = sorted(l)
print("Sorted list:", sorted_list)
# Output: Sorted list: [1, 2, 3, 4, 5]
```

Principe de fonctionnement des objets itérables

- Les objets itérables sont des éléments essentiels en Python.
- Ils permettent de parcourir séquentiellement les éléments d'une collection de données.
- Les objets itérables implémentent le protocole d'itération.
- Ils peuvent être parcourus avec une boucle `for` ou avec les fonctions `iter()` et `next()`.
- `__iter__()`: Renvoie un itérateur sur l'objet.
- `__next__()`: Récupère le prochain élément de l'objet itérable.
- En comprenant le principe de fonctionnement des objets itérables et en les utilisant correctement, l'on écrit ainsi un code plus efficace et plus lisible pour manipuler les collections de données dans les programmes.

Exemple de fonctionnement des objets itérables

```
class MyIterable:
    def __init__(self, max):
        self.max = max
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.max:
            self.current += 1
            return self.current
        else:
            raise StopIteration

# Using the iterable
my_iterable = MyIterable(5)
for element in my_iterable:
    print(element)
```

Fonction reversed()

- La fonction `reversed()` est une fonction intégrée en Python utilisée pour inverser l'ordre des éléments d'une séquence.
- Elle prend un objet séquentiel (comme une liste, un tuple ou une chaîne de caractères) en tant qu'argument et renvoie un itérateur inversé.
- L'itérateur renvoyé par `reversed()` parcourt les éléments de la séquence de la fin au début.
- Usage: `reversed(sequence)`
- Paramètres:
  - `sequence`: La séquence à inverser.
- Documentation: Documentation officielle de la fonction reversed

Exemples de la fonction reversed()

```python
# Example with a string
my_string = "Hello"
my_string_reversed = reversed(my_string)
print(type(my_string_reversed))
# Output: <class 'reversed'>
for char in my_string_reversed:
    print(char)

# Example with a tuple
my_tuple = (1, 2, 3, 4, 5)
my_tuple_reversed = reversed(my_tuple)
print(type(my_tuple_reversed))
# Output: <class 'reversed'>
for value in my_tuple_reversed:
    print(value)

# Example with a list
my_list = [1, 2, 3, 4, 5]
my_list_reversed = reversed(my_list)
print(type(my_list_reversed))
# Output: <class 'list_reverseiterator'>
for element in my_list_reversed:
    print(element)
```

Remarque sur la différence de types d'itérateurs inversés

- Lorsque vous utilisez la fonction `reversed()` en Python pour inverser les éléments d'une séquence, le type de l'itérateur retourné peut varier en fonction du type de séquence d'entrée.
  - Pour les tuples, `reversed()` renvoie un objet itérateur de type `reversed`, spécifiquement conçu pour parcourir les éléments du tuple dans l'ordre inverse.
  - Pour les listes, `reversed()` renvoie un objet itérateur de type `list_reverseiterator`, qui offre des fonctionnalités spécifiques aux listes en plus de permettre l'itération dans l'ordre inverse.
- Il est important de noter cette différence de types, car cela peut affecter le comportement et les opérations que vous pouvez effectuer sur l'itérateur inversé, en fonction du type de la séquence d'origine.

Opérations spécifiques aux listes pour l'itérateur inversé

- L'objet itérateur de type `list_reverseiterator`, retourné par la fonction `reversed()` pour les listes, prend en charge certaines opérations spécifiques aux listes en plus de permettre l'itération dans l'ordre inverse.
  - Il est possible de modifier la liste d'origine pendant l'itération sans générer d'erreur.
  - On peut accéder aux éléments de la liste par leur indice pendant l'itération.
  - Il est également possible d'utiliser les méthodes spécifiques aux listes telles que `append()`, `extend()`, `pop()`, `remove()`, etc., pendant l'itération.
  - Enfin, l'itérateur peut être converti à nouveau en une liste à l'aide du constructeur `list()`.
- Pour plus d'informations sur ces opérations, consultez la documentation officielle.

Opérations spécifiques aux listes pour l'itérateur inversé - Exemples

```python
# Example of modifying the original list during iteration
original_list = [1, 2, 3, 4, 5]
for element in reversed(original_list):
    original_list.append(element * 2)
print(original_list)  # Output: [1, 2, 3, 4, 5, 10, 8, 6, 4, 2]

# Example of accessing elements by index during iteration
for index, element in enumerate(reversed(original_list)):
    print(f"Index: {index}, Element: {element}")

# Example of using list-specific methods during iteration
for element in reversed(original_list):
    if element % 2 == 0:
        original_list.remove(element)
print(original_list)  # Output: [1, 3, 5]

# Example of converting the iterator back to a list
list_reverseiterator_object = reversed(original_list)
reversed_list = list(list_reverseiterator_object)
print(reversed_list)  # Output: [5, 3, 1]
```

Limitations de reversed() sur les ensembles et les dictionnaires

- Les ensembles (sets) en Python sont des collections non ordonnées d'éléments uniques.
- Les dictionnaires (dictionaries) en Python sont des collections de paires clé-valeur non ordonnées.
- Les ensembles et les dictionnaires ne garantissent pas d'ordre spécifique pour leurs éléments.
- Les paires clé-valeur dans un dictionnaire et les éléments dans un ensemble sont stockés de manière non séquentielle.
- Par conséquent, il n'a pas de sens de "renverser" un ensemble ou un dictionnaire car ils n'ont pas de notion intrinsèque d'ordre pour leurs éléments.

Fonction range()

- La fonction `range()` est une fonction intégrée en Python utilisée pour générer une séquence d'entiers.
- Elle prend un, deux ou trois arguments, qui définissent le début, la fin (exclue) et le pas de la séquence.
- La séquence générée commence à partir du début, s'arrête avant la fin et utilise le pas pour incrémenter les valeurs.
- La fonction `range()` est couramment utilisée dans les boucles `for` pour parcourir des séquences d'entiers.
- Usage: `range(start, stop, step)`
- Paramètres:
  - `start`: Valeur de départ de la séquence (par défaut: 0)
  - `stop`: Valeur de fin de la séquence (non incluse)
  - `step`: Pas d'incrément entre les valeurs successives (par défaut: 1)
- Documentation: Documentation officielle de la fonction range

Exemples de la Fonction range()

```python
def printLine():
    print("----------------------------")

# Usage with only stop parameter (start defaults to 0, step defaults to 1)
for i in range(5):
    print(i)
printLine()

# Usage with specified start and specified stop (start defaults to 0, step defaults to 1)
for j in range(2,6):
    print(j)
printLine()

# Usage with specified start, stop, and step parameters
for k in range(1, 8, 2):
    print(k)
printLine()

# Usage with a negative step to generate a decreasing sequence
for l in range(6, 0, -1):
    print(l)
printLine()
```

Exemples de la Fonction range() avec Opérateur *

```python
def printLine():
    print("----------------------------")

# Usage with only stop parameter (start defaults to 0, step defaults to 1)
R1 = range(5)
print(*R1)
printLine()

# Usage with specified start and specified stop (start defaults to 0, step defaults to 1)
R2 = range(2,6)
print(*R2)
printLine()

# Usage with specified start, stop, and step parameters
R3 = range(1, 8, 2)
print(*R3)
printLine()

# Usage with a negative step to generate a decreasing sequence
R4 = range(6, 0, -1)
print(*R4)
printLine()
```

Exemples de la Fonction range() avec Opérateur _

```
def printLine():
    print("----------------------------")

# Example using the range() function to generate a sequence of numbers
for _ in range(4):
    print("Hello")
printLine()

# Example using the range() function with specified start and step parameters
for _ in range(1, 6, 2):
    print("Ignored")
printLine()

# Example using the range() function with only the stop parameter specified
for _ in range(5):
    print("Skipped")
printLine()

# Example using the range() function to generate an empty sequence
for _ in range(0):
    print("No output")
printLine()
```

Exemples de la Fonction range() avec Opérateurs * et _

```python
# Usage with the * operator
n = 5
for i in range(*[n]):
    print(i)

# Usage with the * operator to unpack a list
start_stop = [1, 10]
for j in range(*start_stop):
    print(j)

# Usage with the * operator to unpack a tuple
params = (10, 20, 2)
for k in range(*params):
    print(k)

# Usage with the _ operator to ignore a value
for _ in range(3):
    print("Hello")
```

Fonctions `reversed()` et `range()`

- Comme nous l'avons vu, les fonctions `reversed()` et `range()` sont donc utilisées pour créer des objets itérables en Python.
- **Exemple d'utilisation en Python:**

```python
# Example of iterable objects in Python
r = range(5)
print("Range:", list(r))
# Output: Range: [0, 1, 2, 3, 4]
reversed_r = reversed(r)
print("Reversed range:", list(reversed_r))
# Output: Reversed range: [4, 3, 2, 1, 0]
```

D’avantage d’exemples

- Depacking, range et opérateurs * et _.
- Range et opérateur _.
- Listes et opérateur _.
- Fonction reversed et structures de données.
- Fonction reversed et listes.
- Opérateurs * et _, fonction range et structures de données.

Dictionnaires - Constructeur

- En Python, les dictionnaires peuvent être créés à l'aide du constructeur `dict()` en spécifiant les paires clé-valeur.
- **Exemple en Python:**

```
# Example of dictionary constructor in Python
d = dict({'a': 1, 'b': 2, 'c': 3})
print("Dictionary:", d)
# Output: Dictionary: {'a': 1, 'b': 2, 'c': 3}
```

Dictionnaires - Indiçage

- Les dictionnaires peuvent être indéxés pour accéder à leurs éléments par clé.
- **Exemple d'indiçage en Python:**

```
# Example of indexing in dictionary in Python
d = {'a': 1, 'b': 2, 'c': 3}
print("Value of key 'b':", d['b'])
# Output: Value of key 'b': 2
```

Dictionnaires - Opérateur `in`, fonction `del()`

- Les dictionnaires prennent en charge l'opérateur `in` pour la vérification de l'existence de clés et la fonction `del()` pour la suppression d'éléments par clé.
- **Exemple en Python:**

```python
# Example of operators and functions in dictionary in Python
d = {'a': 1, 'b': 2, 'c': 3}
print("Check if key 'b' exists:", 'b' in d)
# Output: Check if key 'b' exists: True
del d['b']
print("Dictionary after deletion:", d)
# Output: Dictionary after deletion: {'a': 1, 'c': 3}
print(*d)
# Output: a c
```

Dictionnaires - Méthodes `keys()`, `values()`, `items()`, `update()`, `get()`

- Les dictionnaires fournissent diverses méthodes pour accéder aux clés, aux valeurs, aux paires clé-valeur, mettre à jour, et obtenir des valeurs par clé.
- **Exemple de méthodes en Python:**

```python
# Example of methods in dictionary in Python
d = {'a': 1, 'b': 2, 'c': 3}
print("Keys:", d.keys())       # Output: Keys: dict_keys(['a', 'b', 'c'])
print("Values:", d.values())   # Output: Values: dict_values([1, 2, 3])
print("Items:", d.items())     # Output: Items: dict_items([('a', 1), ('b', 2), ('c', 3)])
d.update({'d': 4})
print("Updated dictionary:", d)  # Output: Updated dictionary: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
print("Value for key 'a':", d.get('a'))  # Output: Value for key 'a': 1
```

Dictionnaires - Copie et références

- Les dictionnaires peuvent être copiés d'une manière **superficielle** en utilisant l'opérateur d'assignation =, via la méthode `copy()`, ou encore la méthode `dict()`, tandis que les copies **en profondeur** nécessitent la fonction `deepcopy()` du module `copy`.
- **Exemple de copies en Python:**

```
1 # Example of copying dictionaries in Python
2 import copy # for function deepcopy()
3 d1 = {'a': 1, 'b': 2, 'c': 3}
4 d2 = d1
5 d3 = d1.copy()
6 d4 = dict(d1)
7 d5 = copy.deepcopy(d1)
```

Dictionnaires - Copie et références - Exemple

```python
import copy

# Creating an original dictionary
d1 = {'a': 1, 'b': 2, 'c': 3}

# Copies of the original dictionary
d2 = d1
d3 = d1.copy()
d4 = copy.deepcopy(d1)
dicts = [d1, d2, d3, d4]

def print_dicts():
    for dict in dicts:
        print(dict)

# Before modifying d1
print("# Before modifying d1")
print_dicts()

# Modifying d1
d1['d'] = 4

# After modifying d1
print("# After modifying d1")
print_dicts()
```

Dictionnaires - Différence entre `copy` et `deepcopy`

- La différence entre les méthodes `copy` et `deepcopy` réside dans la manière dont elles traitent les objets imbriqués lors de la copie de dictionnaires en Python.
- La méthode `copy` effectue une copie superficielle, créant un nouveau dictionnaire mais partageant les objets imbriqués avec l'original. En revanche, la fonction `deepcopy` réalise une copie en profondeur, dupliquant tous les objets imbriqués pour créer une copie indépendante de l'original.
- Cet exemple illustre les propos discutés.

Set - Constructor

- En Python, les ensembles peuvent être créés en utilisant le constructeur `set()` en passant un itérable contenant les éléments à inclure dans l'ensemble.
- Alternativement, les ensembles peuvent également être créés en utilisant des accolades {} en listant les éléments séparés par des virgules.
- **Exemple en Python**:

```python
# Example of set constructor in Python
set1 = set([1, 2, 3, 4, 5])
print("Set:", set1)  # Output: Set: {1, 2, 3, 4, 5}
another_set = {1, 2, 3, 4, 5}
print("Set:", another_set)  # Output: Set: {1, 2, 3, 4, 5}
```

Set - Opérateurs - |, &, et ^

- Les ensembles supportent différents opérateurs tels que la différence (-), l'union (|), l'intersection (&), et la différence symétrique (^).
- Ces opérateurs permettent d'effectuer efficacement des opérations sur les ensembles.
- **Exemple des opérateurs en Python**:

```python
# Example of set operators in Python
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print("Difference:", set1 - set2)
# Output: Difference: {1, 2}
print("Union:", set1 | set2)
# Output: Union: {1, 2, 3, 4, 5}
print("Intersection:", set1 & set2)
# Output: Intersection: {3}
print("Symmetric Difference:", set1 ^ set2)
# Output: Symmetric Difference: {1, 2, 4, 5}
```

Résumé complet sur les structures de données

Data Type	Assignable	Subscriptable	Ordered	Mutable	Hashable
String	No c[i] = x	Yes print(c[i])	Yes	No	Yes
Tuple	No t[i] = x	Yes print(t[i])	Yes	No	Yes
List	Yes l[i] = x	Yes print(l[i])	Yes	Yes	No
Set	No s[i] = x	No print(s[i])	No	Yes	No
FrozenSet	No	No	No	No	Yes
Dict	Yes d['k'] = x	Yes print(d['k'])	Yes 3.7/No	Yes	No

- Résumé complet sur les structures de données.

** Structures conditionnelles et répétitives **

Structure conditionnelle `if` ...

- En algorithmique et en programmation, nous appelons **structure conditionnelle** l'ensemble des instructions qui testent si une condition est vraie ou non. Il y en a trois :
  - La structure conditionnelle **if** :

```
if condition:
      my_instructions
```

Structure conditionnelle `if ...else`

- **Suite :**
  - La structure conditionnelle **if ...else** :

```
if condition:
      my_instructions
else:
      my_other_instructions
```

Structure conditionnelle `if …elif …else`

- **Suite :**
  - La structure conditionnelle **if …elif …else :**

```
if condition_1:
      my_instructions_1
elif condition_2:
      my_instructions_2
else:
      my_other_instructions
```

Structure conditionnelle boucle `while`

- De plus, une **structure conditionnelle itérative** permet d'exécuter plusieurs fois (itérations) la même série d'instructions. L'instruction **while** exécute les blocs d'instructions tant que la condition de while est vraie. Le même principe s'applique à la boucle **for** (pour), les deux structures sont détaillées ci-dessous :
  - La structure conditionnelle itérative **while** :

```
while condition:
        mes_instructions
```

Structure conditionnelle: boucle `for`

- **Suite** :
  - La structure conditionnelle itérative `for` :

```
for i in range(begin, end, step):
      my_instructions
```

- **Remarque** : La différence majeure entre la boucle for et la boucle `while` est que la boucle for est utilisée lorsque le nombre d'itérations est connu, tandis que l'exécution se fait dans la boucle `while` jusqu'à ce que l'instruction dans le programme soit prouvée fausse.

Structure conditionnelle: boucle `for`

- La structure conditionnelle element-based `for` :

```
for element in collection:
    my_instructions
```

Instructions break et continue

- L'instruction `break` permet de sortir prématurément d'une boucle.
- L'instruction `continue` permet de passer à l'itération suivante dans une boucle.
- Elles sont souvent utilisées pour contrôler le flux d'exécution dans les boucles.

```
# Example of break and continue statements
for i in range(10):
    if i == 5:
        break  # Exit the loop when i equals 5
    print(i, end=' ')
print("\n")

for i in range(10):
    if i % 2 == 0:
        continue  # Skip the current iteration if i is even
    print(i, end=' ')
```

Fonction enumerate()

- La fonction `enumerate()` est utilisée pour énumérer les éléments d'une séquence tout en conservant leur indice.
- Elle renvoie un objet énumérable qui produit des tuples contenant à la fois l'index et la valeur de chaque élément de la séquence.
- Elle est couramment utilisée dans les boucles pour accéder à la fois à l'indice et à la valeur de chaque élément.

```
1 # Example of enumerate function
2 fruits = ['apple', 'banana', 'cherry']
3 for index, fruit in enumerate(fruits):
4     print(index, fruit)
```

Bloc else sur structure répétitive

- En Python, les boucles `for` et `while` peuvent avoir un bloc `else` qui est exécuté lorsque la boucle se termine normalement (sans être interrompue par un `break`).
- Le bloc `else` est exécuté après que la condition de la boucle devienne fausse.

```
1 # Examples of else block on loop
2 for i in range(5):
3     print(i)
4 else:
5     print("Loop completed without break")
```

```
1 for i in range(5):
2     if i == 3:
3         break
4 else:
5     print("Loop completed without break")
```

- Cela peut être utile pour exécuter du code après la boucle si aucune instruction `break` n'a été rencontrée.

Compréhensions en Python

- Les compréhensions sont une fonctionnalité de Python permettant de créer de manière concise des listes, des ensembles et des dictionnaires à partir d'itérables. Cependant, elles ne sont pas disponibles pour les tuples en raison de leur immutabilité.
  - Les listes (`list`) sont des collections mutables, ce qui signifie que leurs éléments peuvent être modifiés après leur création. Les compréhensions permettent de créer rapidement des listes en appliquant une opération à chaque élément d'un itérable.
  - Les ensembles (`set`) sont également mutables, mais ils exigent l'unicité parmi leurs éléments. Les compréhensions sont utiles pour créer des ensembles car elles garantissent automatiquement l'unicité des éléments.
  - Les dictionnaires (`dict`) sont des collections de paires clé-valeur mutables. Les compréhensions pour les dictionnaires permettent de créer des dictionnaires de manière concise en itérant sur un itérable et en générant des paires clé-valeur.

Compréhensions en Python - Suite

- Suite:
  - Cependant, les tuples (`tuple`) sont des collections immuables. Étant donné que les compréhensions impliquent la création d'une nouvelle collection en itérant sur un itérable et en appliquant une opération à chaque élément, il n'y a pas de moyen direct d'appliquer des compréhensions aux tuples.
- Bien que vous puissiez utiliser des expressions de générateur pour obtenir des résultats similaires avec les tuples, elles ne possèdent pas la même syntaxe que les compréhensions. Les expressions de générateur produisent un objet générateur, qui peut être utilisé pour évaluer les éléments de manière différente.

Liste en intension (comprehension list)

- Les compréhensions de listes sont des constructions syntaxiques concises et élégantes pour la création de listes en Python.
- Elles permettent de créer rapidement et efficacement des listes en appliquant une expression à chaque élément d'une séquence.
- Les compréhensions de listes sont écrites entre crochets `[ ]`.

```
1 # Example of list comprehension
2 squares_comprehension_list = [x**2 for x in range(10)]
3 print(squares_comprehension_list)
4 # Output: [0, 1, 64, 4, 36, 9, 16, 49, 81, 25]
5 print(type(squares_comprehension_list))
6 # Output: <class 'list'>
```

Ensemble en intension (comprehension set)

- Les compréhensions d'ensembles sont des constructions syntaxiques concises et élégantes pour la création d'ensembles en Python.
- Elles permettent de créer rapidement et efficacement des ensembles en appliquant une expression à chaque élément d'une séquence.
- Les compréhensions d'ensembles sont écrites entre accolades `{ }`.

```
1 # Example of set comprehension
2 squares_comprehension_set = {x**2 for x in range(10)}
3 print(squares_comprehension_set)
4 # Output: {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
5 print(type(squares_comprehension_set))
6 # Output: <class 'set'>
```

Dictionnaire en intension (comprehension dict)

- Les compréhensions de dictionnaires sont des constructions syntaxiques concises et élégantes pour la création de dictionnaires en Python.
- Elles permettent de créer rapidement et efficacement des dictionnaires en appliquant une expression à chaque paire clé-valeur d'une séquence.
- Les compréhensions de dictionnaires sont écrites entre accolades `{x: }`.

```
# Example of dict comprehension
squares_comprehension_dict = {x: x**2 for x in range(10)}
print(squares_comprehension_dict)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
print(type(squares_comprehension_dict))
# Output: <class 'dict'>
```

Tuple en intension (comprehension tuple)

- Comme évoqué, la notion de compréhension de tuple n'existe pas à proprement parler, cependant en voulant appliquer une compréhension l'on obtient un objet de type `generator`

```
# Example of tuple comprehension
squares_comprehension_tuple = (x**2 for x in range(10))
# Works fine, but the result is not of type tuple
print(squares_comprehension_tuple)
# Output: <generator object <genexpr> at 0x104deb970>
print(*squares_comprehension_tuple)
# Output: 0 1 4 9 16 25 36 49 64 81
print(type(squares_comprehension_list))
# Output: <class 'generator'>
```