Course: Python

Ali ZAINOUL <contact@clearcode.fr>

Crystal Clear Code
June 16, 2025

- 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()
- Collections: Dictionnaires (dict)
  - Constructeur
  - Indiçage
  - Opérateur `in`, fonction `del()`
  - Méthodes `keys()`, `values()`, `items()`, `update()`, `get()`
  - Copie et références
- Collections: Classe Set
  - Constructeur

** Types de données non-modifiables **

Utilité des types non-modifiables (optimisation mémoire)

- Les types non-modifiables, tels que les tuples et les chaînes de caractères, offrent l'avantage de l'optimisation mémoire.
- En Python, une fois qu'un objet non modifiable est créé, son contenu ne peut pas être modifié.
- Cela permet à l'interpréteur Python d'effectuer certaines optimisations en termes de gestion de la mémoire, car il sait que ces objets resteront inchangés.
- Ainsi, les types non-modifiables peuvent contribuer à réduire l'utilisation de la mémoire dans les programmes Python, ce qui peut être crucial pour les applications exigeantes en termes de performances et d'efficacité.

Fonction `id()`

- La fonction `id()` retourne l'identifiant de l'objet donné. Cet identifiant est unique pour chaque objet et reste constant pendant toute la durée de vie de l'objet.
- Documentation officielle
- **Usage**: `id(object)`.
- **Exemple**:

```
1 # Example of using the id() function
2 x = 42
3 print("ID of x:", id(x))
```

Fonction `hash()`

- La fonction `hash()` retourne la valeur de hachage d'un objet. Les objets qui sont égaux doivent avoir le même hachage, mais l'inverse n'est pas nécessairement vrai.
- Documentation officielle
- **Usage**: `hash(object)`.
- **Exemple**:

```
1 # Example of using the hash() function
2 x = 42
3 print("Hash de x:", hash(x))
```

Opérateur `is`

- L'opérateur `is` vérifie si deux variables font référence au même objet en mémoire.
- Si les deux variables référencent le même objet, `is` renvoie `True`; sinon, il renvoie `False`.
- **Usage**: `object1 is object2`
- Documentation officielle
- Exemple GFG
- **Exemple**:

```
# Example of using the is operator
x = [1, 2, 3]
y = x
print(x is y)  # True, because x and y references the same
    object
```

La classe `str` en Python

- La classe `str` est utilisée pour représenter les chaînes de caractères en Python.
- Les objets de type `str` sont **ordonnées immutables**, ce qui signifie qu'ils ne peuvent pas être modifiés après leur création.
- Documentation officielle
- **Exemple**:

```
1 # Example of using the str class
2 s = "Hello, world!"
3 print(s) # Displays: Hello, world!
4 print(s.__class__) # Displays: <class 'str'>
```

Méthodes importantes de la classe `str`

- La classe `str` offre de nombreuses méthodes pour manipuler les chaînes de caractères, telles que `split()`, `replace()`, `lower()`, `upper()`, `strip()`, `join()`, etc.
- Ces méthodes permettent de réaliser des opérations courantes sur les chaînes de caractères, comme diviser une chaîne en plusieurs parties, remplacer des sous-chaînes, convertir la casse, etc.
- Documentation officielle des méthodes str

Ordonnancement de la classe str

```python
# Example showing the ordering of the str class in Python
s = "hello"
print("Original:", s)

# Access by positive and negative index
print("First character:", s[0])  # Displays the first character 'h'
print("Second character:", s[1])  # Displays the second character 'e'
print("Last character:", s[-1])  # Displays the last character 'o'

# Using a for loop to iterate over each character with index
print("Traversal with a for loop and indices:")
for i in range(len(s)):
    print("Character at index", i, ":", s[i])

# Slicing with positive and negative indices
print("Slicing with positive and negative indices:")
print("First three characters:", s[:3])  # Displays 'hel'
print("Last three characters:", s[-3:])  # Displays 'llo'

# Using a for loop to iterate over each character
print("Traversal with a for loop:")
for char in s:
    print(char)
```

Tuple

- Les tuples sont des structures de données immuables et ordonnées en Python. Ils peuvent contenir des éléments de différents types et sont définis par des parenthèses.
- **Exemple de constructeur en Python**:

```python
# Example of tuple constructor in Python
t = (1, 2, 3)
print("Tuple:", t)  # Output: Tuple: (1, 2, 3)
```

Tuple - Constructeur

- En Python, les tuples peuvent être créés à l'aide du constructeur `tuple()` en spécifiant les éléments à inclure dans le tuple. Les éléments peuvent être de n'importe quel type de données.
- **Exemple en Python:**

```python
# Example of tuple constructor in Python
t = tuple([1, 2, 3, 4, 5])
print("Tuple:", t)  # Output: Tuple: (1, 2, 3, 4, 5)
```

Tuple - Indiçage

- Les tuples peuvent être indéxés pour accéder à leurs éléments individuels. L'indiçage commence à partir de 0 pour accéder aux éléments du tuple.
- **Exemple d'indiçage en Python:**

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

Tuple - Itération

- Les tuples peuvent être itérés à l'aide d'une boucle `for`, ce qui permet d'accéder à chaque élément du tuple.
- **Exemple d'itération en Python:**

```python
# Example of iteration over tuple in Python
t = (1, 2, 3)
for element in t:
    print(element)
```

Tuple - Opérateurs +, * et in

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

```python
# Example of operators in tuple in Python
t1 = (1, 2, 3)
t2 = (4, 5, 6)
print("Concatenated tuple:", t1 + t2)
# Output: Concatenated tuple: (1, 2, 3, 4, 5, 6)
print("Repeated tuple:", t1 * 2)
# Output: Repeated tuple: (1, 2, 3, 1, 2, 3)
print("Check if 2 in tuple:", 2 in t1)
# Output: Check if 2 in tuple: True
```

Tuple - Méthodes `count()` et `index()`

- Les tuples prennent en charge les méthodes `count()` et `index()` pour compter le nombre d'occurrences d'un élément et pour trouver l'index de la première occurrence d'un élément donné, respectivement.
- **Exemple de méthodes en Python:**

```
1 # Example of methods in tuple in Python
2 t = (1, 2, 2, 3, 4)
3 print("Count of 2:", t.count(2))     # Output: Count of 2: 2
4 print("Index of 3:", t.index(3))     # Output: Index of 3: 3
```

La classe `Tuple` en Python

- Un tuple est une collection de valeurs qui est **ordonnée** et **immutable**.
- Les tuples sont créés à l'aide de parenthèses `()` et chaque valeur est séparée par une virgule :

```python
# Examples of tuple manipulation
colors = ("red", "green", "blue")
print(colors[1])  # Result: "green"
```

- Vous pouvez **accéder** aux éléments individuels d'un tuple en utilisant leur **indice**, mais **vous ne pouvez pas modifier** les éléments eux-mêmes. Les tuples sont utiles pour représenter des collections **fixes de valeurs**.
- Documentation officielle de la classe Tuple.
- Documentation GFG des méthodes de la classe Tuple.

Méthodes importantes de la classe `tuple`

- La classe `tuple` offre quelques méthodes pour travailler avec les tuples, telles que `count()` et `index()`.
- Ces méthodes permettent de compter le nombre d'occurrences d'un élément dans un tuple et de trouver l'index de la première occurrence d'un élément donné.
- Documentation officielle des méthodes tuple

Ordonnancement de la classe tuple

```python
# Extended example showing the ordering of the tuple class in Python
t = (1, 2, 3, 4, 5)
print("Original:", t)

# Access by positive and negative index
print("First element:", t[0])  # Displays the first element 1
print("Second element:", t[1])  # Displays the second element 2
print("Last element:", t[-1])  # Displays the last element 5

# Using a for loop to iterate over each element with index
print("Traversal with a for loop and indices:")
for i in range(len(t)):
    print("Element at index", i, ":", t[i])

# Slicing with positive and negative indices
print("Slicing with positive and negative indices:")
print("First three elements:", t[:3])  # Displays (1, 2, 3)
print("Last three elements:", t[-3:])  # Displays (3, 4, 5)

# Using a for loop to iterate over each element
print("Traversal with a for loop:")
for element in t:
    print(element)
```

Principe du depacking de variables

- Principe du depacking de variables, permet d'extraire les valeurs contenues dans une structure de données complexe (comme un tuple ou une liste) et de les assigner à des variables individuelles.
- **Exemple en Python**:

```python
# Example of variable unpacking in Python
coordinates = (3, 4)
x, y = coordinates
print("x:", x, "y:", y)  # Output: x: 3 y: 4
```

Tableau d'octets (bytes)

- Les objets de type `bytes` en Python sont utilisés pour stocker une séquence d'octets. Ils sont immuables et peuvent contenir des valeurs dans la plage de 0 à 255.
- **Exemple de constructeur en Python**:

```python
# Example of bytes constructor in Python
b = bytes([65, 66, 67])
print(f'Bytes object: {b}')  # Output: Bytes object: b'ABC'
```

Objet None et fonction `repr()`

- L'objet `None` est utilisé pour représenter la nullité ou l'absence de valeur. La fonction `repr()` renvoie une représentation sous forme de chaîne de caractères de l'objet passé en argument.
- **Exemple en Python**:

```python
# Example of None object and repr() function in Python
x = None
print("None object:", x)
# Output: None object: None
print("Representation of x:", repr(x))
# Output: Representation of x: None
```

** Types de données modifiables **

La classe `List` en Python

- Une liste est une collection de valeurs qui est **ordonnée** et **mutable**.
- Les listes sont créées à l'aide de crochets `[]` et chaque valeur est séparée par une virgule :

```
# Example of list manipulation
fruits = ["apple", "banana", "orange"]
print(fruits[0])  # Result: "apple"
fruits.append("grape")
print(fruits)  # Result: ["apple", "banana", "orange", "grape"]
```

- Vous pouvez **accéder** et **modifier** des éléments individuels d'une liste en utilisant leur **indice**. Vous pouvez également **ajouter** ou **supprimer** des éléments à l'aide de méthodes intégrées telles que `append()`, `insert()`, `remove()` et `pop()`.
- Documentation officielle de la classe List

Méthodes importantes de la classe `liste`

- La classe `liste` offre quelques méthodes pour travailler avec les listes, telles que `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `clear()`, `index()`, `count()` et `sort()`.
- Ces méthodes permettent d'ajouter des éléments à une liste, d'étendre une liste avec les éléments d'une autre liste, d'insérer un élément à une position donnée, de supprimer un élément spécifique, de retirer et retourner le dernier élément, de vider la liste, de trouver l'index de la première occurrence d'un élément, de compter le nombre d'occurrences d'un élément et de trier la liste.
- Documentation officielle des méthodes liste

Ordonnancement de la classe liste

```python
# Exemple étendu montrant l'ordonnancement de la classe list en Python
l = [1, 2, 3, 4, 5]
print("Original:", l)

# Accès par indice positif et négatif
print("Premier élément:", l[0])  # Affiche le premier élément 1
print("Deuxième élément:", l[1])  # Affiche le deuxième élément 2
print("Dernier élément:", l[-1])  # Affiche le dernier élément 5

# Utilisation d'une boucle for pour parcourir chaque élément avec indice
print("Parcours avec une boucle for et des indices:")
for i in range(len(l)):
    print("Élément à l'indice", i, ":", l[i])

# Slicing avec des indices positifs et négatifs
print("Slicing avec des indices positifs et négatifs:")
print("Premiers trois éléments:", l[:3])  # Affiche [1, 2, 3]
print("Derniers trois éléments:", l[-3:])  # Affiche [3, 4, 5]

# Utilisation d'une boucle for pour parcourir chaque élément
print("Parcours avec une boucle for:")
for element in l:
    print(element)
```

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:**

```
# 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:**

```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

```python
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 _

```python
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:**

```
# 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:**

```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.

La classe `Dict` en Python

- Un dictionnaire est une collection de paires `clé-valeur` qui est **non ordonnée** et **mutable**.
- Les dictionnaires sont créés à l'aide d'accolades `{}` et chaque paire `clé-valeur` est séparée par deux-points `:` ,illustration:

```
# Example of dictionary manipulation
person = {"name": "Ali", "age": 30, "city": "MTP"}
print(person["name"])  # Result: "Ali"
person["age"] = 31
print(person)  # Result: {"name": "Ali", "age": 31, "city": "MTP"}
```

- Vous pouvez **accéder** et **modifier** des valeurs dans un dictionnaire en utilisant leurs **clés**. Vous pouvez également **ajouter** ou **supprimer** des paires `key-value` à l'aide de méthodes intégrées telles que `update()`, `pop()` et `copy()`.

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}
```

La classe `Set` en Python

- Un ensemble est une collection de valeurs qui est **non ordonnée** et **unique**.
- Les ensembles sont créés à l'aide d'accolades `{}` ou de la fonction `set()` et chaque valeur est séparée par une virgule :

```python
# Example of set manipulation
fruits = {"apple", "banana", "orange"}
print("apple" in fruits)  # Result: True
fruits.add("grape")
print(fruits)  # Result: {"apple", "banana", "orange", "grape"}
```

- Vous pouvez vérifier si une valeur est dans un ensemble en utilisant l'opérateur `in`. Vous pouvez également **ajouter** ou **supprimer** des éléments à l'aide de méthodes intégrées telles que `add()`, `remove()` et `discard()`.

Comparaison des Structures de Données

Type	Mutabilité	Ordre	Unicité
Tuple	Immutable	Ordonné	Peut contenir des doublons
Liste	Mutable	Ordonnée	Peut contenir des doublons
Ensemble	Mutable	Non ordonné	Pas de doublons
Dictionnaire	Mutable	Clés Ordonnées (>= 3.7)	Clés uniques, mais les valeurs peuvent être dupl

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 `[ ]`.

```
# Example of list comprehension
squares_comprehension_list = [x**2 for x in range(10)]
print(squares_comprehension_list)
# Output: [0, 1, 64, 4, 36, 9, 16, 49, 81, 25]
print(type(squares_comprehension_list))
# 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 `{ }`.

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

```python
# 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'>
```

Tableau sur les structures de données

Data Structure	Unique	Assignable	Ordered*	Mutable	Hashable
STR					
TUPLE					
LIST					
SET					
FROZEN SET					
DICT**					

- Dict : Unique Keys, Values may be the same for diffent keys. (*)
- Subscriptible == Ordered (**)

Relations

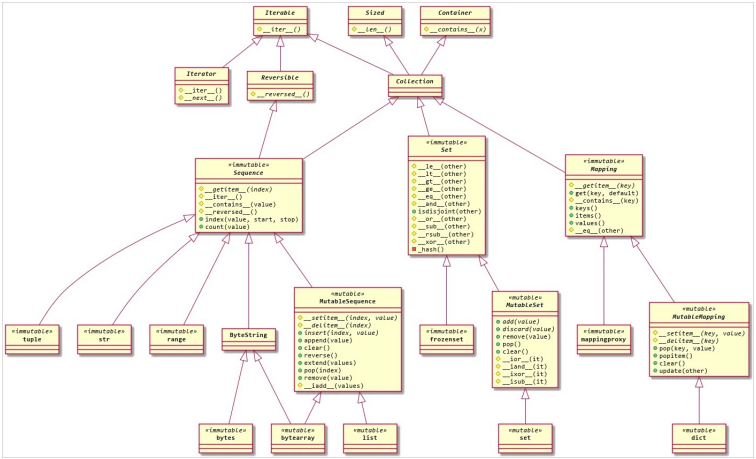

Figure: DataStruct Class Diagram