1B_python_data_type_usage

November 27, 2023

1 Python Random Module

Python offre des fonctions pour générer des nombres aléatoires. Voici quelques exemples:

[1]:
```
import random

# Print random element
print(random.random())

# Print random uniform element
print(random.uniform(10, 20))

# Print random integer element
print(random.randint(10, 20))

# Print random element
print(random.randrange(10, 40, 8))

list1 = ['a', 'b', 'c', 'd', 'e', 11]

# get random item from list1
print(random.choice(list1))

# Shuffle list1
random.shuffle(list1)

# Print the shuffled list1
print(list1)
```

```
0.10642393775129433
17.300922202109238
10
34
c
['a', 'e', 'b', 'd', 11, 'c']
```

- random.random()ˆ retourne un nombre flottant aléatoire entre 0 et 1
- random.uniform(a, b)ˆ retourne un nombre flottant aléatoire entre a et b (inclus)
- random.randint(a, b)ˆ retourne un nombre entier aléatoire entre a (inclus) et b (inclus)

- random.randrange(a, b, step): retourne un nombre entier aléatoire entre a (inclus) et b (exclus) avec un pas de step
- random.choice(sequence): retourne un élément aléatoire de la séquence
- random.shuffle(sequence): mélange la séquence

2 Python Mathematics

Python offre des fonctions mathématiques pour effectuer des opérations mathématiques plus complexes. Voici quelques exemples:

vous trouverez plus d'informations sur le module math ainsi que toutes les fonctiosn proposées ici

[ ]:
```
import math
print(f"PI\t\t:\t{math.pi}")
print(f"e\t\t:\t{math.e}")
print(f"ceil(4.4)\t:\t{math.ceil(4.4)}")
print(f"floor(4.4)\t:\t{math.floor(4.4)}")
print(f"fabs(-4.4)\t:\t{math.fabs(-4.4)}")
print(f"factorial(4)\t:\t{math.factorial(4)}")
print(f"pow(2,2)\t:\t{math.pow(2,2)}")
print(f"sqrt(4)\t\t:\t{math.sqrt(4)}")
print(f"exp(3)\t\t:\t{math.exp(3)}")
print(f"log(1000)\t:\t{math.log(1000)}")
print(f"log10(1000)\t:\t{math.log10(1000)}")
print(f"log2(1024)\t:\t{math.log2(1024)}")
print(f"sin(PI)\t\t:\t{math.sin(math.pi)}")
print(f"cos(PI)\t\t:\t{math.cos(math.pi)}")
print(f"degrees(PI)\t:\t{math.degrees(math.pi)}")
print(f"radians(180)\t:\t{math.radians(180)}")
```

3 Python Lists

Les listes sont des objets qui peuvent contenir plusieurs éléments. Ils sont définis par des crochets et les éléments sont séparés par des virgules. Les éléments peuvent être de différents types.

3.1 Create a List

Pour créer une liste, vous pouvez utiliser la fonction list() ou les crochets [].

```
# Create a list
my_list = list()
my_list = []
```

[2]:
```
import timeit
print(timeit.timeit('[]', number=10**7))
print(timeit.timeit('list()', number=10**7))
```

```
0.13962950000131968
0.32745191699905263
```

Note: La création d'une liste vide en utilisant `[]` est ~2X plus rapide que d'utiliser la fonction `list()`

Autres Façons de créer une liste:

[ ]:
```
l = [1, 2, 3, 4, 5]
print(l)

l2 = list(range(1, 6))
print(l2)

l3 = ['A'] * 5
print(l3)

l4 = list('Hello')
print(l4)

l5 = list(map(lambda x: x*x, range(1, 6)))
print(l5)

l6 = [x*x for x in range(1, 6)]
print(l6)

l_diff = [1, 1.23, "salut", "s"]
print(l_diff)
```

Une liste peut : - contenir des éléments de différents types (nombres, chaînes de caractères, booléens, listes, etc.) - contenir des éléments dupliqués

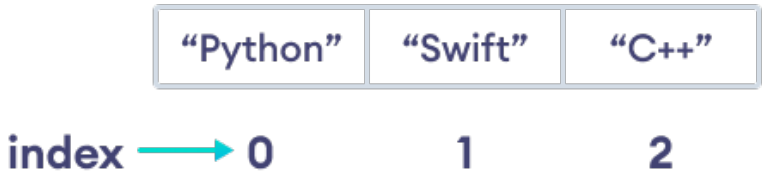
[ ]:
```
# list with elements of different data types
list1 = [1, "Hello", 3.4]

# list with duplicate elements
list2 = [1, "Hello", 3.4, "Hello", 1]

# empty list
list3 = []

# nested list
list4 = [[1,2], [3,4]]


# lambda function
list5 = list(map(lambda x: x**2, range(1, 10, 2)))

print(list1)
print(list2)
print(list3)
```

```
print(list4)
print(list5)
```

3.2 Acceder aux elements d'une liste

En python les listes sont des objets indexés. Cela signifie que chaque élément de la liste a un index. L'index est un nombre entier qui commence à 0. Pour accéder à un élément de la liste, vous devez utiliser l'index de cet élément.

[ ]:
```
languages = ["Python", "Swift", "C++"]

# access item at index 0
print(languages[0])

# access item at index 2
print(languages[2])
```



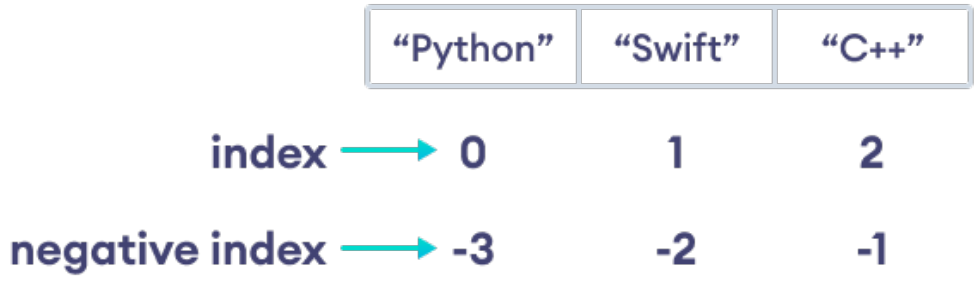
Source >

Note: Les index des listes commencent à 0

3.3 Les index négatifs

Les index négatifs sont utilisés pour accéder aux éléments de la liste en partant de la fin. L'index -1 correspond au dernier élément de la liste, l'index -2 correspond à l'avant-dernier élément de la liste, etc.

[ ]:
```
languages = ["Python", "Swift", "C++"]

print(languages[-3]) # print(languages[0])

print(languages[-1])
```



Source

Note: Si un index n'existe pas, une erreur de type `IndexError` sera levée

3.4 List slicing

Vous pouvez accéder à une partie de la liste en utilisant la notation de tranche. La notation de tranche est `list[start:end:step]`. Les éléments de la liste sont inclus de l'index de début à l'index de fin - 1. Le pas est facultatif et par défaut, il est égal à 1.

[ ]:
```
# List slicing in Python
my_list = ['A','n','t','o','n','i','o']

# items from index 2 to index 4
print(my_list[2:5])

# items from index 5 to end
print(my_list[5:])

# items from begining to index 5
print(my_list[:5])

# items beginning to end
print(my_list[:])
print(my_list)

# List slicing with negative index
print(my_list[2:-2:1])

print(my_list[::2])

print(my_list[::-1])
```

Ici :

- `my_list[2:5]` - accéder aux éléments de la liste de l'index 2 à l'index 4
- `my_list[5:]` - accéder aux éléments de la liste de l'index 5 à la fin de la liste
- `my_list[:5]` - accéder aux éléments de la liste du début de la liste à l'index 4
- `my_list[:]` - accéder à tous les éléments de la liste
- `my_list[2:-2]` - accéder aux éléments de la liste de l'index 2 à l'avant-dernier élément de la liste
- `my_list[::2]` - accéder à tous les éléments de la liste avec un pas de 2

Note: Lors du `slicing`, le premier index est inclus et le dernier index est exclu

Ajouter un élément à une liste

Pour ajouter un élément à une liste, vous pouvez utiliser la méthode `append()`, `extends()` ou `insert()`.

Il est possible d'ajouter des éléments aux `listes`, car les `listes` sont des objets mutables (modifiables).

1.

3.4.1 append()

La méthode `append()` ajoute un élément à la fin de la liste.

[ ]:
```
numbers = [21, 34, 54, 12]

print("Before Append:", numbers)

# using append method
numbers.append("32")

print("After Append:", numbers)
```

2.

3.4.2 extend()

La méthode `extend()` ajoute tous les éléments d'un itérable (list, tuples, string, dictionary, etc.) à la fin de la liste.

[ ]:
```
numbers = [1, 3, 5]

even_numbers = [2, 4, 6]

# add elements of even_numbers to the numbers list
numbers.extend(even_numbers)

print("List after extend:", numbers)
```

3.

3.4.3 insert()

La méthode `insert()` ajoute un élément à un index spécifique de la liste.

[ ]:
```
numbers = [10, 30, 40]




# insert an element at index 1 (second position)
#numbers.insert(1, 20)

print(numbers)

print(numbers[:1] + [20, 25] + numbers[1:])
```

3.5 Changer la valeur d'un élément d'une liste

Les listes en python sont mutables, ce qui signifie que vous pouvez modifier les valeurs des éléments de la liste. Pour changer une valeur d'un élément de la liste, vous devez utiliser l'index de cet élément. et assigner une valeur en utilisant l'opérateur d'affectation `=`.

[ ]:
```
languages = ['Python', 'Swift', 'C++']
print(languages)
# changing the third item to 'C'
languages[2] = 'C'

print(languages)
```

3.6 Suppression d'un élément

Pour supprimer un élément de la liste, vous pouvez utiliser la méthode del, `remove()` ou `pop()`.

1.

3.6.1 del

[ ]:
```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']
print(languages)

# deleting the second item
del languages[1]
print(languages)

# deleting the last item
del languages[-1]
print(languages)
# delete the first two items
del languages[0 : 2]
print(languages)
```

2.

3.6.2 remove()

La fonction `remove()` supprime le premier élément de la liste dont la valeur est égale à la valeur passée en paramètre.

[ ]:
```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R', 'Python']
print(languages)
# remove 'Python' from the list
languages.remove('Python')
print(languages)
```

3.

3.6.3 pop()

La méthode `pop()` supprime l'élément à l'index spécifié et retourne cet élément.

```
[ ]: languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']
print(languages[::-1])

poped = languages.pop(0)
print(poped)

# remove and return the last item
print(languages.pop())
print(languages)

# remove and return the second last item
print(languages.pop(-2))

print(languages)
```

Methodes de listes

Voici quelques méthodes utiles pour les listes:

Method	Description
`append()`	Ajoute un élément à la fin de la liste
`extend()`	Ajoute tous les éléments d'un itérable (list, tuples, string, dictionary, etc.) à la fin de la liste
`insert()`	Ajoute un élément à un index spécifique de la liste
`remove()`	Supprime le premier élément de la liste dont la valeur est égale à la valeur passée en paramètre
`pop()`	Supprime l'élément à l'index spécifié et retourne cet élément
`clear()`	Supprime tous les éléments de la liste
`index()`	Retourne l'index du premier élément de la liste dont la valeur est égale à la valeur passée en paramètre
`count()`	Retourne le nombre d'éléments de la liste dont la valeur est égale à la valeur passée en paramètre
`sort()`	Trie les éléments de la liste
`reverse()`	Inverse l'ordre des éléments de la liste
`copy()`	Retourne une copie de la liste

3.7 Itérer sur une liste

Pour itérer sur une liste, vous pouvez utiliser la boucle `for` ou la boucle `while`.

[ ]:
```
my_list = [3, 8, 1, 6, 0, 8, 4]
i=0
# Iterate over a list using for loop
for i, item in enumerate(my_list):
    print(item)
    if item == 0:
        print(i)

#print(my_list[5])

# Iterate over a list using while loop
i = 0
while i < len(my_list):
    if my_list[i] == 3:
        print(i)
    i += 1
```

3.8 Check if an element exists in a list

Pour vérifier si un élément existe dans une liste, vous pouvez utiliser l'opérateur `in` ou `not in`.

[ ]:
```
languages = ['Python', 'Swift', 'C++']

print('C' in languages)
print('Python' in languages)

print('C' not in languages)
print('Python' not in languages)
```

3.9 Longueur d'une liste

Pour obtenir la longueur d'une liste, vous pouvez utiliser la fonction `len()`.

[ ]:
```
languages = ['Python', 'Swift', 'C++']

print("List: ", languages)

print("Total Elements: ", len(languages))
```

3.10 List Comprehension

La compréhension de liste est une façon élégante de définir et de créer des listes en python. Nous pouvons créer des listes en utilisant une boucle `for` et une condition `if` sur cette boucle. La syntaxe de la compréhension de liste est la suivante:

`[expression for item in list]`

Supposons que nous voulions séparer les lettres du mot `human`. L'approche classique consiste à utiliser une boucle `for` comme ceci:

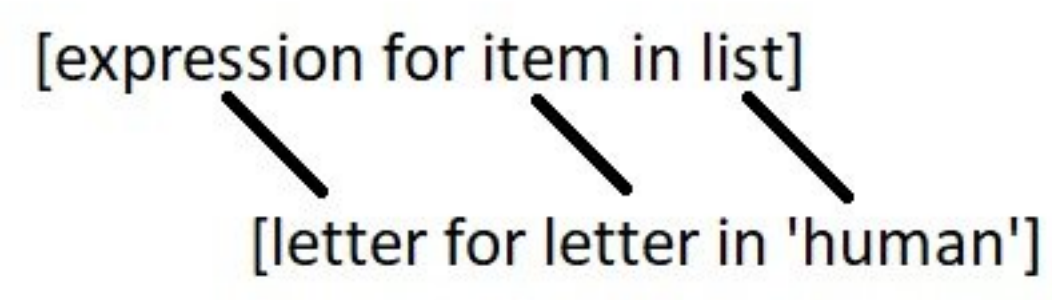
[ ]:
```
h_letters = []

for letter in 'human':
    h_letters.append(letter.upper())

print(h_letters)
```

Cepeandant, en utilisant la compréhension de liste, nous pouvons le faire en une seule ligne de code.

[ ]:
```
h_letters = []

for letter in 'human':
    h_letters.append(letter.upper())

print(h_letters)

print([{i: letter.lower()} for i, letter in enumerate('HuMan')])
```

3.10.1 Syntax de la compréhension de liste

```
[expression for item in list]
```



Source

3.11 List comprehensions avec Condition

La syntaxe de la compréhension de liste avec condition est la suivante:

```
[expression for item in list if condition]
```

[ ]:
```
number_list = [x*2 for x in range(10) if x % 2 == 1]
print(number_list)
```

3.12 List comprehensions avec plusieurs conditions

La syntaxe de la compréhension de liste avec plusieurs conditions est la suivante:

```
[expression for item in list if condition1 and condition2]
```

Le comportement de la compréhension de liste avec plusieurs conditions est similaire à l'utilisation de la clause `and` dans une boucle `for`.

[ ]:
```
num_list = [y for y in range(100) if (y % 2 == 0) and (y % 5 == 0)]
print(num_list)
```

Ici, la `compréhension de liste` vérifie si: - `y` est divisible par 2 - `y` est divisible par 5

`y` sera ajouté à la liste si les deux conditions sont `True` (and).

3.13 if...else With List Comprehension

La syntaxe de la compréhension de liste avec `if...else` est la suivante:

```
[true_expression if condition else false_expression for item in list]
```

[ ]:
```
obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
```

3.14 Nested loops in List Comprehensions

La syntaxe de la compréhension de liste avec des boucles imbriquées est la suivante:

```
[expression for item in list1 for item2 in list2]
```

Supposons que nous voulions calculer la transposée de la matrice suivante:

1	2	3	4
4	5	6	8
9	1	3	2

1	4	9
2	5	1
3	6	3
4	8	2

Nous pouvons utiliser le code python basique suivant:

[ ]:
```
# Initialize an empty list to store the transposed matrix
transposed = []

# Original matrix
matrix = [[1, 2, 3, 4], [4, 5, 6, 8], [9,1,3,2]]
print(matrix[1][1])

# Iterate over the columns of the original matrix
for i in range(len(matrix[0])): # boucle sur les colonnes
    # Initialize an empty list to store the transposed row
    transposed_row = []
```

```
    # Iterate over the rows of the original matrix
    for row in matrix:
        # Append the element at the corresponding column to the transposed row
        transposed_row.append(row[i])

    # Append the transposed row to the transposed matrix
    transposed.append(transposed_row)

# Print the transposed matrix
print(transposed)
```

Ce code utilise deux boucles `for` imbriquées pour transposer la matrice.

Nous pouvons le faire en une seule ligne de code en utilisant la compréhension de liste imbriquée.

[ ]:
```
# Original matrix
matrix = [[1, 2, 3, 4], [4, 5, 6, 8], [9,1,3,2]]

# Transpose the matrix using a list comprehension
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]

# Print the transposed matrix
print(transposed)
```

3.15 Bonus

Il est possible d'obtenir, le meme résultat, en utilisant la fonction `map()` et la fonction `lambda`.

[ ]:
```
def square(n):
    return n*n

squares = list(map(square, [1, 2, 3, 4, 5]))
print(squares)
```

3.16 List comprehension Conclusion

- La compréhension de liste est une façon élégante de définir et de créer des listes en python.
- Les listes de compréhension sont plus rapides que les boucles `for` et `while` traditionnelles. Elles sont aussi plus compactes.
- `/!\` Les listes de compréhension peuvent être difficiles à lire. Essayez d'éviter les listes de compréhension imbriquées de plus de deux niveaux ou trop longues
- Chaque compréhension de liste peut être réécrite en boucle `for` traditionnelle. Mais chaque boucle `for` ne peut pas être réécrite en compréhension de liste.

Python Tuples

Les tuples sont des séquences immuables, c'est-à-dire qu'elles ne peuvent pas être modifiées après leur création. Les tuples sont utilisés pour stocker des collections d'éléments de données. Un tuple est similaire à une liste. Cependant, les tuples utilisent des parenthèses et les listes utilisent des crochets.

3.17 Creation de tuples

Pour créer un tuple, il suffit de meetre les éléments entre les `()` et de les séparer par des virgules. Il est également possible d'utiliser la fonction `tuple()`.

> **Note**: Un tuple avec un seul élément doit avoir une virgule après l'élément, sinon il sera considéré comme un type différent.
>
> **Note**: Les parenthèses ne sont pas obligatoires pour créer un tuple. Mais il est fortement recommandé d'utiliser des parenthèses.

[ ]:
```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)
my_tuple = tuple()
print(my_tuple)

# Tuple having 1 item
my_tuple = 1
print(type(my_tuple), my_tuple)

my_tuple = 1,
print(type(my_tuple), my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)

# tuple can be created without parentheses
my_tuple = 3, 4.6, "dog"
print(my_tuple)
```

3.18 Acceder aux elements d'un tuple

Comme pour les listes, les tuples sont indexés et vous pouvez accéder aux éléments d'un tuple en utilisant des indices. l'index commence à 0.

1. ### Indexing

Pour accéder à un élément d'un tuple, vous pouvez utiliser l'opérateur `[]` avec l'index correspondant. Si l'index n'existe pas, une exception `IndexError` est levée. Si on utilise un index de type autre que `int`, une exception `TypeError` est levée.

[ ]:
```
# accessing tuple elements using indexing
letters = ("A", "n", "t", "o", "n", "i", "o")

print(letters[0:3])
print(letters[5])
```

2.

3.18.1 Negative indexing in tuples

Les tuples prennent également en charge l'indexation négative. Cela signifie que vous pouvez accéder aux éléments d'un tuple en utilisant des indices négatifs. L'index `-1` fait référence au dernier élément, `-2` fait référence à l'avant-dernier élément, et ainsi de suite.

[ ]:
```
# accessing tuple elements using negative indexing
letters = ("A", "n", "t", "o", "n", "i", "o")
letters[0] = "h"
print(letters[-1])
print(letters[-3])
```

3.

3.18.2 Slicing of Tuple

Vous pouvez accéder à une plage de valeurs dans un tuple en utilisant la notation de tranche. La syntaxe de la tranche de tuple est la suivante:

`tuple[start:stop:step]`

[ ]:
```
# accessing tuple elements using slicing
my_tuple = ("A", "n", "t", "o", "n", "i", "o")

# elements 2nd to 4th index
print(my_tuple[1:4])

# elements beginning to 2nd
print(my_tuple[:-5])

# elements 5th to end
print(my_tuple[5:])

# elements beginning to end
print(my_tuple[:])

# all the elements with step 2
print(my_tuple[::-1])
```

Change or delete a tuple

Les tuples sont immuables, ce qui signifie que vous ne pouvez pas modifier ou mettre à jour les valeurs des éléments d'un tuple. Vous ne pouvez pas non plus supprimer d'éléments d'un tuple.

Cependant, vous pouvez supprimer un tuple entier en utilisant l'instruction `del`.

```
[ ]: my_tuple = ("A", "n", "t", "o", "n", "i", "o")

# changing element value
#my_tuple[0] = "a"    # TypeError: 'tuple' object does not support item␣
 ↪assignment

# deleting a tuple
del my_tuple

print(my_tuple)  # NameError: name 'my_tuple' is not defined
```

3.19 Python tuples methods

Method	Description
`count()`	Retourne le nombre de fois qu'un élément apparaît dans un tuple
`index()`	Recherche l'élément spécifié et renvoie son index

```
[ ]: my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))
print(my_tuple.index('p'))
```

3.20 Iterating Through a Tuple

Vous pouvez itérer à travers un tuple en utilisant une boucle `for`. Vous pouvez également utiliser la boucle `while` pour itérer à travers un tuple. Il est aussi intéressant de noter que l'on peut `unpack` un tuple.

```
[ ]: languages = ('Python', 'Swift', 'C++')

# iterating through the tuple
for language in languages:
    print(language, end=" ")

print()
i=0
while i < len(languages):
    print(languages[i], end=" ")
    i += 1

print()
# tuple unpacking is also possible
languages = ('Python', 'Swift', 'C++')
```

```
# unpacking tuple
language1, language2, language3 = languages

print(language1, language2, language3, sep=" ")

for i, v in enumerate(languages):
    print(v)
```

3.21 Vérifier si un élément est présent dans un tuple

Pour vérifier si un élément est présent dans un tuple, vous pouvez utiliser l'opérateur `in` et `not in`.

[ ]:
```
languages = ('Python', 'Swift', 'C++')

print('C' in languages)
print('Python' in languages)
print('C' not in languages)
print('Python' not in languages)
```

3.22 Avantages d'utiliser des tuples

Comme les tuples sont similaires aux listes, les deux sont utilisés dans des situations. Cependant, il existe quelques avantages à utiliser des tuples au lieu de listes.

- Souvent on utilise les `tuples` quand on veut stocker des données de type **hétérogènes**(differentes). Les listes sont utilisées pour stocker des données de type **homogènes**(même type).
- Comme les tuples sont immuables, le traitement des tuples est plus rapide que celui des listes.
- Comme les `tuples` sont immuables, il est possible de les utilisés comme clé dans un dictionnaire. Les listes ne peuvent pas être utilisées comme clé dans un dictionnaire car les listes peuvent être modifiées après leur création.
- Si vous avez des donnnées qui ne doivent pas être changées, vous devez utiliser un tuple, car vos donnée seront protégées contre l'écriture `write protected`.

4 Python Strings

En programmation un `string` est une séquence de caractères. par exemple, `"hello"` est un string de 5 caractères, contenant les caractères `h`, `e`, `l`, `l` et `o`. Pour créer un string en python, il suffit d'entourer les caractères avec des guillemets simples ou doubles. Les guillemets simples et doubles sont équivalents.

```
# string avec des guillemets simples
print('hello')

# string avec des guillemets doubles
print("hello")
```

4.1 Acceder aux elements d'un string

Les strings sont indexés et vous pouvez accéder aux caractères d'un string en utilisant des indices. l'index commence à 0. Les strings se comportent comme une liste de caractères, donc nous pouvons utiliser la notation de tranche `slicing` pour accéder à une partie d'un string. on peut aussi utiliser des indices négatifs.

[ ]:
```
greet = 'hello'

# access 1st index element
print(greet[1]) # "e"

# access 4th last element - negative indexing
print(greet[-4]) # "e"

# access character from 1st index to 3rd index
print(greet[1:4])  # "ell"
```

> **Note**: si on utilise un index qui n'existe pas, une exception `IndexError` est levée. Si on utilise un index de type autre que `int`, une exception `TypeError` est levée.

4.2 Les strings sont immuables

Les strings sont immuables, ce qui signifie que vous ne pouvez pas modifier ou mettre à jour les caractères d'un string. Cependant, vous pouvez créer un nouveau string en concaténant des caractères d'un string existant ou en utilisant la méthode `replace()`.

[ ]:
```
message = 'Hola Amigos como estan'
message[0] = 'O'
l = message.split(" ")
print((l))
```

[ ]:
```
message = 'Hola Amigos'

# assign new string to message variable
message = 'Hello Hello Friends'

print(message); # prints "Hello Friends"

message = message.replace('Hello', 'Hi')
print(message)
```

Python Multiline Strings

En python, les strings multilignes sont des strings avec des sauts de ligne. Les strings multilignes peuvent être créés en utilisant trois guillemets simples ou doubles. Les guillemets simples et doubles sont équivalents.

[ ]:
```
# multiline string
message = """
```

```
Never gonna give you up
    Never gonna let you down"""

print(message)

print('''Never gonna give you up
Never gonna let you down''')
```

Python String Operations

Operation	Description
`+`	Concaténation - Ajoute des valeurs de part et d'autre de l'opérateur
`*`	Répétition - Crée de nouvelles chaînes, concaténant plusieurs copies de la même chaîne
`[]`	Tranche - Donne la tranche de la chaîne
`[:]`	Tranche de plage - Donne la tranche de la chaîne
`in`	Membre - Renvoie `true` si un caractère existe dans la chaîne
`not in`	Membre - Renvoie `true` si un caractère n'existe pas dans la chaîne
`r/R`	Raw String - Supprime l'interprétation des caractères d'échappement
`%`	Format - Effectue un formatage de chaîne

[ ]:
```
# concatenation
print('hello ' + 'world')

# repetition
print('hello' * 3)

# slicing
print('hello'[1:3])

# in
print('h' in 'hello')

# not in
print('h' not in 'hello')

# raw string
print(r'hello\nworld')

# format
print('hello %s %d' % ('world', 12))
print('hello {} {}'.format('world', 12))
```

```
# f-strings
print(f"hello {'world'} {12}")
```

4.3 Comparing Strings

Les strings peuvent être comparés en utilisant les opérateurs de comparaison `==`, `!=`, `>`, `<`, `>=` et `<=`. Les strings sont comparés en fonction de la valeur Unicode de chaque caractère dans la chaîne.

[ ]:
```
print('a' > 'b')
print('a' < 'b')
print('a' == 'b')
print('a' != 'b')

print('antonio' < 'antonia')
```

4.4 Joining Strings

Les strings peuvent être concaténés en utilisant l'opérateur `+` ou en utilisant la méthode `join()`. La méthode `join()` prend un iterable et renvoie la chaîne concaténée.

[ ]:
```
# concatenation
print('hello' + ' world')

# join
print(' '.join(['hello', 'world']))
print('_'.join(['hello', 'world']))
print(' - '.join(['hello', 'world']))
print("hello ".join(['Nicolas\n', "Georgios\n", "Homan\n", "Elena"]))
a = ["A", "N", "T"]
print("".join(a))
```

4.5 Iterating Through a String

Vous pouvez itérer à travers un string en utilisant une boucle `for`. Vous pouvez également utiliser la boucle `while` pour itérer à travers un string.

[ ]:
```
greet = 'Hello'

# iterating through greet string
for letter in greet:
    print(letter)
```

4.6 Python String Length

La fonction `len()` renvoie la longueur d'une chaîne.

[ ]:
```
greet = 'Hello'

# count length of greet string
print(len(greet))
```

4.7 Python String Methods

voici une liste de certaines méthodes de string couramment utilisées.

Method	Description
`capitalize()`	Convertit le premier caractère en majuscule
`count()`	Renvoie le nombre de fois qu'un élément apparaît dans la chaîne
`encode()`	Renvoie une version encodée de la chaîne
`find()`	Recherche dans la chaîne une valeur spécifiée et renvoie la position de où elle a été trouvée
`format()`	Formate les valeurs spécifiées dans une chaîne
`index()`	Recherche dans la chaîne une valeur spécifiée et renvoie la position de où elle a été trouvée
`isalnum()`	Renvoie `true` si tous les caractères de la chaîne sont alphanumériques
`isalpha()`	Renvoie `true` si tous les caractères de la chaîne sont alphabétiques
`isnumeric()`	Renvoie `true` si tous les caractères de la chaîne sont numériques
`islower()`	Renvoie `true` si tous les caractères de la chaîne sont en minuscules
`join()`	Joindre les éléments d'un itérable à la fin de la chaîne
`lower()`	Convertit une chaîne en minuscules
`upper()`	Convertit une chaîne en majuscules
`replace()`	Renvoie une chaîne où une valeur spécifiée est remplacée par une valeur spécifiée
`partition()`	Renvoie un tuple en trois parties
`[r/l]strip()`	Retourne une chaîne sans espaces
`split()`	Divise la chaîne à l'espace

Source

[ ]:
```
# capitalize()
print('hello'.capitalize())

# count()
print('hello'.count('l'))

# encode()
print(type('hello'.encode()))
```

```
# find()
print('hello'.find('l'))

# format()
print('hello {}'.format('world'))

# index()
print('hello'.index('l'))

# isalnum()
print('hello'.isalnum())

# isalpha()
print('hello'.isalpha())
```

[ ]:
```
# isnumeric()
print('hello'.isnumeric())

# islower()
print('hello'.islower())

# lower()
print('HELLO'.lower())

# upper()
print('hello'.upper())

# replace()
print('hello'.replace('h', 'H'))

# partition()
print('hello'.partition('l'))

# rstrip()
print('     hello    '.strip())

# split()
print('hello world'.split('o'))
print('o'.join('hello world'.split('o')))
```

4.8 Python String escape characters

Les caractères d'échappement sont des caractères spéciaux qui sont utilisés pour représenter des caractères non imprimables tels que les nouvelles lignes, les tabulations et les caractères de retour arrière. Les caractères d'échappement sont représentés par un antislash suivi d'un caractère. Les caractères d'échappement couramment utilisés sont répertoriés ci-dessous.

Escape Character	Description
`\newline`	Retour à la ligne
`\\`	Antislash
`\'`	Guillemet simple
`\"`	Guillemet double
`\a`	Son de la cloche
`\b`	Retour arrière
`\f`	Saut de page
`\n`	Nouvelle ligne
`\r`	Retour chariot
`\t`	Tabulation
`\v`	Tabulation verticale
`\ooo`	Valeur octale
`\xhh`	Valeur hexadécimale

[ ]:
```
print('hello\nworld')
print('hello\tworld')
print('hel\tworld')
print('hello\rworld')
print('hello\vworld')
print('hello\aworld')
print('hello'+ '\b'*3 +'world')
```

5 Set en python

Un set est une collection non ordonnée d'éléments **uniques**. Les sets sont utilisés pour stocker plusieurs éléments dans une seule variable.

5.1 Creation d'un Set

Un set est créé en utilisant la fonction `set()` ou en utilisant des accolades `{}`. Cependant, pour créer un set vide, vous devez utiliser la fonction `set()`, car `{}` crée un dictionnaire vide. Les `Set` permentent d'avoir des éléments de type différents, mais il ne peut pas avoir des éléments mutables comme des listes, des dictionnaires ou des sets, comme éléments.

[ ]:
```
# create an empty set
empty_set = set()

# create an empty dictionary
empty_dictionary = {}

# check data type of empty_set
print('Data type of empty_set:', type(empty_set))

# check data type of dictionary_set
print('Data type of empty_dictionary', type(empty_dictionary))
```

```
# create a set of integer type
student_id = {112, 114, 116, 118, 115, 118, 116, 118}
print('Student ID:', student_id)

# create a set of string type
vowel_letters = {'a', 'e', 'i', 'o', 'u'}
print('Vowel Letters:', vowel_letters)

# create a set of mixed data types
mixed_set = {'Hello', 101, -2, 'Bye'}
print('Set of mixed data types:', mixed_set)
```

> **Note**: L'ordre peut varier car les `Set` sont des collections non ordonnées, vous ne pouvez pas être sûr de l'ordre dans lequel les éléments seront imprimés.

5.2 Duplicate Values in Set

Les `Set` ne peuvent pas avoir d'éléments dupliqués. Si vous créez un set avec des éléments dupliqués, les doublons seront automatiquement supprimés.

[ ]:
```
numbers = {2, 4, 6, 6, 2, 8}
print(numbers)
```

5.3 add and update d'éléments dans un Set

Les `Set` sont des collections non modifiables, ce qui signifie que vous ne pouvez pas modifier directement les éléments existants d'un set. Cependant, vous pouvez ajouter de nouveaux éléments à un set existant. Pour ajouter un seul élément à un set, utilisez la méthode `add()`. Pour ajouter plusieurs éléments à un set, utilisez la méthode `update()`.

[ ]:
```
numbers = {21, 34, 54, 12}

print('Initial Set:',numbers)

# using add() method
numbers.add(34)
# assert 34 is in set

print('Updated Set:', numbers)
```

[ ]:
```
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple', 'Lacoste']

companies.update(tech_companies)

print(companies)
```

5.4 Suppression d'éléments d'un Set

Vous pouvez supprimer des éléments d'un set existant en utilisant la méthode `remove()` ou la méthode `discard()`. La méthode `remove()` supprime l'élément spécifié d'un set. Si l'élément spécifié n'existe pas, une erreur `KeyError` est levée. La méthode `discard()` supprime l'élément spécifié d'un set. Si l'élément spécifié n'existe pas, aucune erreur n'est levée.

[ ]:
```
languages = {'Swift', 'C', 'C++', 'Python', 'Java'}
print('Initial Set:',languages)
# remove 'Java' from a set
removedValue = languages.discard('Java')
print('Set after disccard():', languages)
random_value = languages.pop()
print('Set after pop():', languages, "Random value", random_value)
languages.remove('Swift')
print('Set after remove():', languages)
languages.remove('Java') # KeyError: 'Java'
```

5.5 Python Set built-in methods

voici une liste de certaines méthodes de set couramment utilisées.

Method	Description
`all()`	Renvoie `true` si tous les éléments d'un set sont vrais (ou si le set est vide)
`any()`	Renvoie `true` si l'un des éléments d'un set est vrai. Si le set est vide, renvoie `false`
`enumerate()`	Renvoie un objet énumérable. Il contient la valeur et l'index de tous les éléments du set sous forme de paires
`len()`	Renvoie la longueur (le nombre d'éléments) d'un set
`max()`	Renvoie l'élément de valeur maximale d'un set
`min()`	Renvoie l'élément de valeur minimale d'un set
`sorted()`	Renvoie une nouvelle list contenant tous les éléments du set triés
`sum()`	Renvoie la somme de tous les éléments d'un set

[ ]:
```
# Création d'un set
my_set = {1,2,3}

print(all(my_set))
print(any(my_set))
print(len(my_set))
print(max(my_set))
print(min(my_set))
print(sorted(my_set))
```

```
print(sum(my_set))
```

5.6 Iterating Through a Set

Vous pouvez itérer à travers un set en utilisant une boucle `for`.

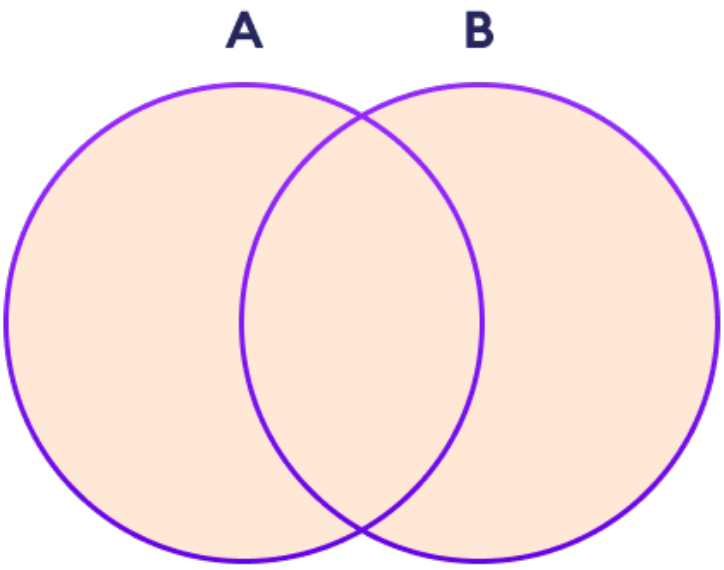
[ ]:
```
fruits = {"Apple", "Peach", "Mango"}

# for loop to access each fruits
for fruit in (fruits):
    print(fruit)

for i, v in enumerate(fruits):
    print(i, v)
```

5.7 Set operations

Les `Set` premettent d'effectuer des opérations mathématiques comme l'`union`, l'`intersection`, la `différence` et la `différence symétrique`. Vous pouvez créer des ensembles en utilisant des accolades `{}` ou la fonction `set()`.

5.7.1 Union de deux Sets

L'union de deux sets renvoie un nouveau set contenant tous les éléments des deux sets. On utilise l'opérateur `|` ou la méthode `union()` pour effectuer l'union.



Source

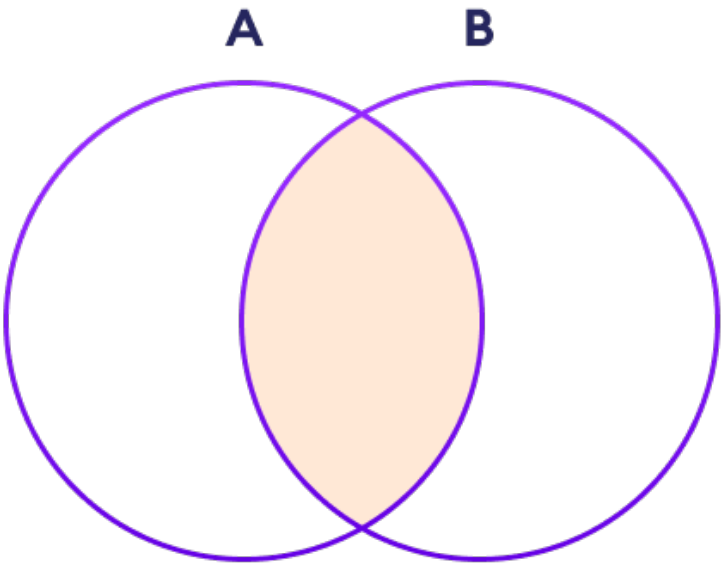
[ ]:
```
# first set
A = {1, 3, 5}
```

```
# second set
B = {0, 2, 4, 5}

# perform union operation using |
print('Union using |:', A | B)

# perform union operation using union()
print('Union using union():', A.union(B))
```

Note: `A|B` et `union()` sont équivalent à `A  B` en mathématique.

5.7.2 Set Intersection

L'intersection de deux sets renvoie un nouveau set contenant uniquement les éléments communs aux deux sets. On utilise l'opérateur `&` ou la méthode `intersection()` pour effectuer l'intersection.



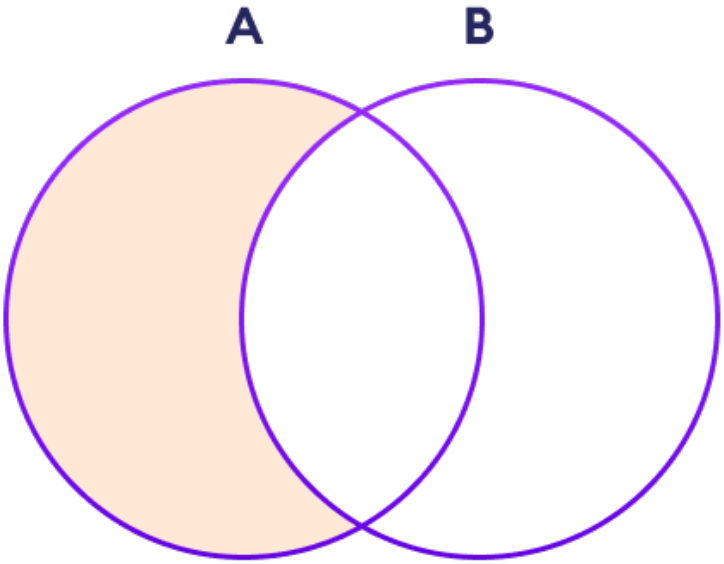
Source

[ ]:
```
# first set
A = {1, 3, 5}

# second set
B = {1, 2, 3}

# perform intersection operation using &
print('Intersection using &:', A & B)

# perform intersection operation using intersection()
print('Intersection using intersection():', A.intersection(B))
```

Note: `A&B` et `intersection()` sont équivalent à `A  B` en mathématique.

5.7.3 Set Difference

La différence de deux sets renvoie un nouveau set contenant les éléments uniquement présents dans le premier set et non dans le second. On utilise l'opérateur `-` ou la méthode `difference()` pour effectuer la différence.



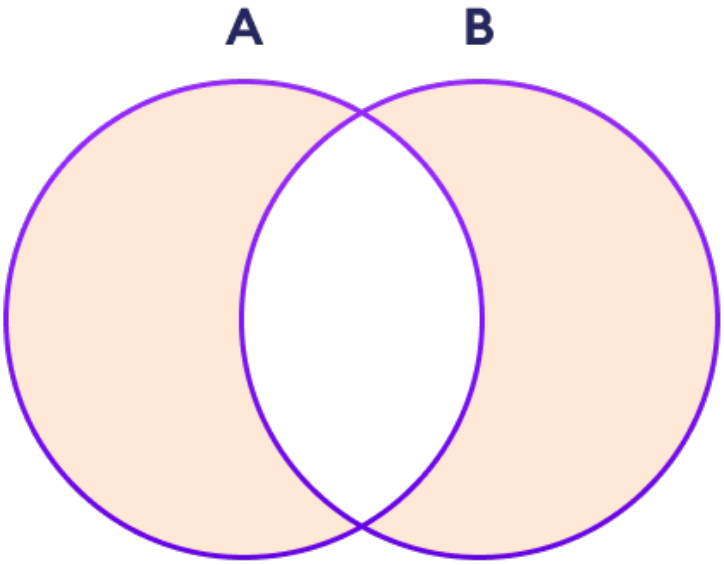
Source

[ ]:
```
# first set
A = {2, 3, 5}

# second set
B = {1, 2, 6}

# perform difference operation using &
print('Difference using &:', A - B)

# perform difference operation using difference()
print('Difference using difference():', A.difference(B))
```

Note: `A-B` et `A.difference(B)` sont équivalent à `A \ B` en mathématique.

5.7.4 Set Symmetric Difference

La différence symétrique de deux sets renvoie un nouveau set contenant tous les éléments des deux sets, à l'exception des éléments communs aux deux sets. On utilise l'opérateur `^` ou la méthode `symmetric_difference()` pour effectuer la différence symétrique.



Source

[ ]:
```
# first set
A = {2, 3, 5}

# second set
B = {1, 2, 6}

# perform difference operation using &
print('using ^:', A ^ B)

# using symmetric_difference()
print('using symmetric_difference():', A.symmetric_difference(B))
```

5.7.5 Verifier si deux sets sont égaux

Deux sets sont égaux si et seulement si tous les éléments d'un set sont présents dans l'autre set. On utilise l'opérateur == pour vérifier l'égalité.

[ ]:
```
# first set
A = {1, 3, 5}

# second set
B = {3, 5, 1}

# perform difference operation using &
if A == B:
    print('Set A and Set B are equal')
else:
    print('Set A and Set B are not equal')
```

5.8 Autres fonctions sur les sets

Fonction	Description
`add()`	Ajoute un élément à un set
`clear()`	Supprime tous les éléments d'un set
`copy()`	Renvoie une copie d'un set
`difference()`	Renvoie la différence entre deux ou plusieurs sets
`difference_update()`	Supprime tous les éléments d'un set qui sont présents dans d'autres sets spécifiés
`discard()`	Supprime l'élément spécifié
`intersection()`	Renvoie un set, qui est l'intersection de deux autres sets
`intersection_update()`	Supprime les éléments d'un set qui ne sont pas présents dans d'autres sets spécifiés
`isdisjoint()`	Renvoie `true` si deux sets n'ont pas d'éléments en commun
`issubset()`	Renvoie `true` si tous les éléments d'un set sont présents dans un autre set (set contenant)
`issuperset()`	Renvoie `true` si tous les éléments d'un set sont présents dans un autre set (set contenu)
`pop()`	Supprime un élément d'un set
`remove()`	Supprime l'élément spécifié
`symmetric_difference()`	Renvoie un set contenant tous les éléments de deux sets, à l'exception des éléments communs aux deux sets
`symmetric_difference_update()`	Insère les éléments de la différence symétrique entre deux sets dans le premier set
`union()`	Renvoie un set contenant tous les éléments de deux ou plusieurs sets
`update()`	Met à jour un set avec l'union de lui-même et d'autres

6 Les dictionnaires en Python

Les dictionnaires sont des structures de données utilisées pour stocker des données sous forme de paires clé-valeur. Les dictionnaires sont indexés par des clés. Les clés sont uniques dans un dictionnaire. Les dictionnaires sont construits avec des accolades `{}` et les paires clé-valeur sont séparées par des virgules.

6.1 Creation de dictionnaires

Pour créer un dictionnaire, il suffit de mettre les paires clé-valeur entre les `{}` et de les séparer par des virgules. Il est également possible d'utiliser la fonction `dict()`. Les pairs clé-valeur sont séparées par des virgules et la clé et la valeur sont séparées par `:`.

[ ]:
```python
# creating a dictionary
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  ("GB", "EU"): 6,
  "hobbies": ["surfing", "programming", "making music"],
  "Italy": "Naples",
  67.5: "sixty-seven"
}

# printing the dictionary
print(country_capitals)
```

Note: Les clés doivent être uniques dans un dictionnaire, c'est-à-dire qu'il ne peut y avoir qu'une seule occurrence d'une clé donnée dans un dictionnaire.

Note: Les clés doivent être de type immuable, c'est-à-dire qu'elles doivent être de type `string`, `number` ou `tuple`.

[ ]:
```python
# Valid dictionary

my_dict = {
  1: "Hello",
  (1, 2): "Hello Hi",
  3: [1, 2, 3]
}

print(my_dict)

# Invalid dictionary

my_dict = {
  1: "Hello",
  [1, 2]: "Hello Hi",
}

print(my_dict)
my_dict = {}
my_dict = dict()
```

[ ]:
```python
import timeit
print(timeit.timeit('{}', number=10**7))
print(timeit.timeit('dict()', number=10**7))
```

6.2 Longueur d'un dictionnaire

Pour obtenir la longueur d'un dictionnaire, vous pouvez utiliser la fonction `len()`.

[ ]:
```
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}

# get dictionary's length
print(len(country_capitals))
```

6.3 Acceder aux valeurs d'un dictionnaire

Pour accéder à une valeur d'un dictionnaire, vous pouvez utiliser l'opérateur `[]` avec la clé correspondante. Si la clé n'existe pas, une exception `KeyError` est levée. Si on utilise une clé de type autre que `string`, `number` ou `tuple`, une exception `TypeError` est levée. On peut aussi utiliser la méthode `get()`.

[ ]:
```
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}

print(country_capitals["United States"])
print(country_capitals["England"])
#print(country_capitals["France"])

print(country_capitals.get("United States"))
print(country_capitals.get("France"))
print(country_capitals.get("France", "Pays non reconnu"))
```

Manipuler les dictionnaires ### Modifier les valeurs d'un dictionnaire

Les dictionnaires sont mutables, ce qui signifie que vous pouvez modifier les valeurs d'un dictionnaire après sa création.

Pour changer la valeur d'un élément d'un dictionnaire, on utilise l'opérateur `[]` avec la clé correspondante. Si la clé n'existe pas, une nouvelle paire clé-valeur est ajoutée au dictionnaire. Si on utilise une clé de type autre que `string`, `number` ou `tuple`, une exception `TypeError` est levée.

[ ]:
```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples",
  "England": "London"
}
# change the value of "Italy" key to "Rome"
country_capitals["Italy"] = "Rome"
print(country_capitals)
```

6.3.1 Ajouter des éléments à un dictionnaire

Pour ajouter un nouvel élément à un dictionnaire, on utilise l'opérateur `[]` avec une nouvelle clé et on lui attribue une valeur. Si la clé existe déjà, la valeur existante est remplacée par la nouvelle valeur. Sinon la nouvelle paire clé-valeur est ajoutée au dictionnaire.

[ ]:
```
country_capitals = {
  "United States": "New York",
  "Italy": "Rome"
}

# add an item with "Germany" as key and "Berlin" as its value
country_capitals["Germany"] = "Berlin"

print(country_capitals)
```

> **Note**: On peut aussi utiliser la méthode `update()` pour ajouter un nouvel élément à un dictionnaire.

[ ]:
```
# update method on country_capitals dictionary
country_capitals = {
  "United States": "New York",
  "Italy": "Naple"
}

# update the dictionary with key-value pairs
country_capitals.update({"France": "Paris"})
country_capitals.update({"Germany": "Berlin", "Italy": "Rome"})

print(country_capitals)
```

6.3.2 Supression d'un élément

Pour supprimer un élément d'un dictionnaire, on utilise l'instruction `del`. Si la clé n'existe pas, une exception `KeyError` est levée. On peut aussi utiliser la méthode `pop()`, `popitem()` et `clear()`.

[ ]:
```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples",
  "England": "London",
  "Germany": "Berlin",
  "France": "Paris",
}
# delete item having "United States" key
#del country_capitals["United States"]
#print(country_capitals)
#print(country_capitals.popitem())
#print(country_capitals)
#print(country_capitals.pop("Italyy"))
```

```
#print(country_capitals)
country_capitals.clear()
print(country_capitals)
```

Dictionary Methods

Method	Description
`clear()`	Supprime tous les éléments d'un dictionnaire
`copy()`	Retourne une copie d'un dictionnaire
`get()`	Retourne la valeur de la clé spécifiée
`items()`	Retourne une liste contenant une séquence de tuples, chacun représentant une paire clé-valeur
`keys()`	Retourne une liste contenant les clés d'un dictionnaire
`pop()`	Supprime l'élément avec la clé spécifiée
`popitem()`	Supprime le dernier élément inséré
`setdefault()`	Retourne la valeur de la clé spécifiée. Si la clé ne figure pas dans le dictionnaire, la clé est insérée avec la valeur spécifiée
`update()`	Met à jour le dictionnaire avec les paires clé-valeur spécifiées
`values()`	Retourne une liste de toutes les valeurs d'un dictionnaire

6.4 Vérifier l'existence d'une clé

Pour vérifier si une clé existe dans un dictionnaire, on utilise l'opérateur `in` et `not in`.

[ ]:
```
my_dict = {1: "Hello", "Hi": 25, "Howdy": 100}

print(1 in my_dict)

print("Howdy" not in my_dict)

print("Hello" in my_dict.values())
```

Note: les operateurs `in` et `not in` check uniquement les clés et non les valeurs.

6.5 Iterating Through a Dictionary

Depuis Python 3.7, les dictionnaires sont ordonnés. ce qui signifie que vous pouvez itérer à travers un dictionnaire en utilisant une boucle `for`.

[ ]:
```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples"
}
```

```
for country in country_capitals:
    print(country)

print("----------")

for country in country_capitals:
    capital = country_capitals[country]
    print(capital)


for capitals in country_capitals.values():
    print(capitals)

print("----------")

for country in country_capitals.keys(): # this time the for loop throught a␣
 ↪list and no more a dict
    print(country)

print("----------")

for country, capital in country_capitals.items():
    print(country, " - ", capital)
```

[ ]: