
Les boucles permettent de parcourir les éléments d'une liste et d'appliquer les mêmes actions sur chaque élément de la liste quelque soit sa longueur.

# Parcourir une liste

On a souvent besoin de parcourir toutes les entrées d'une liste en performant une action chaque élément, ceci peut être effectué à l'aide de l'instruction **for**.

```
cars = ['Audi', 'BMW', 'Bugatti', 'Alfa Romeo', 'Mercedes', 'Renault', 'Peugot']
```

```
for car in cars:
  print(car)
```

```
Audi
BMW
Bugatti
Alfa Romeo
Mercedes
Renault
Peugot
```

Le concept de boucles est important car c'est la manière la plus utilisée pour automatiser les tâches répétitives.

```
for car in cars:
  print(f"{car.upper()}")
```

```
AUDI
BMW
BUGATTI
ALFA ROMEO
MERCEDES
RENAULT
PEUGOT
```

# Indentation

Python utilise l'indentation afin de déterminer comme une ligne ou un ensemble de lignes sont reliées au reste du programme. Les indentations sont utilisées afin d'améliorer la lisibilité du code. L'omission des indentations peut engendrer des erreurs.

```
cars = ['Audi', 'Maserati', 'Bugatti']


for car in cars:
print(car)
```

```
  File "<ipython-input-6-a235cc7830eb>", line 2
    print(car)
        ^
IndentationError: expected an indented block
```

SEARCH STACK OVERFLOW

Parfois, l'omission des indentations n'engendre aucune erreur mais le résultat final est faussé. Il s'agit d'une erreur logique, la syntaxe est valide mais le code ne produit pas le résultat escompté.

```
for car in cars:
  print(f"{car.upper()}")
print(f"{car.lower}")
```

Quand on indente une ligne qui n'a pas besoin d'être indentée, Python affiche une erreur.

```
message = "Hello World!"
  print(message)
```

```
  File "<ipython-input-7-26a7768477c7>", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

SEARCH STACK OVERFLOW

Au cas où une ligne est indentée après une boucle, cette ligne sera exécutée une fois par élément. Ceci produit une erreur logique. Si les deux points : sont omis après une instruction

**for**, une erreur de syntaxe est affichée.

```
for car in cars
  print(car)
```

```
      File "<ipython-input-8-695ff4b0828d>", line 1
        for car in cars
                       ^
    SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

# Listes numériques

Les listes sont idéales pour le stockage des valeurs numériques. Python met à disposition des développeurs un ensemble d'outils afin de les aider à utiliser les listes numériques d'une manière efficiente même quelque soit leur taille.

### Utilisation de la fonction range

La fonction **range** permet de générer une série de nombres.

```
for value in range(1, 5):
  print(value)
```

```
    1
    2
    3
    4
```

Si le premier argument n'est pas mentionné, la valeur par défaut utilisée est 0:

```
for value in range(5):
  print(value)
```

```
    0
    1
    2
    3
    4
```

**Utiliser range pour générer une liste numérique** On peut convertir les résultats de la fonction **range** directement en utilisant la fonction **list**, en entourant range avec list, le résultat est une liste de nombres.

```
numbers = list(range(1,6))
```

```
numbers
```

```
[1, 2, 3, 4, 5]
```

On peut également préciser le 3ème argument afin de préciser le pas de la série (step).

```
even_numbers = list(range(2, 11, 2))
```

```
even_numbers
```

```
[2, 4, 6, 8, 10]
```

```
squares = []

for value in range(1, 11):
  square = value ** 2
  squares.append(square)

print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

# Statistiques

Plusieurs fonctions sont utiles lors de l'utilisation des listes de nombres. On peut facilement trouver le minimum, le maximum, et la somme de la liste des nombres.

```
temperatures = [23, 13, 34, 42, 12, 14, 15]
```

```
min(temperatures)
```

```
12
```

```
max(temperatures)
```

```
42
```

# Listes de compréhension

Les listes de compréhension permettent de générer des séries d'éléments en utilsant une seule ligne de code "One liner" en combinant la boucle **for** et l'action dans une seule ligne.

```
squares = [value**2 for value in range(1, 11)]

squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

# Utiliser une partie de la liste

On peut travailler avec une partie de la liste denotée **slice**. Afin de générer un slice, il suffit de spécifier l'index du premier élément ainsi que l'index du dernier élément.

```
cars = ['Audi', 'Maserati', 'Bugatti', 'Mercedes', 'Renault', 'Peugot', 'Citroen', 'Honda'

cars[1:4]
```

```
['Maserati', 'Bugatti', 'Mercedes']
```

```
cars[:4]
```

```
['Audi', 'Maserati', 'Bugatti', 'Mercedes']
```

```
cars[2:]
```

```
['Bugatti', 'Mercedes', 'Renault', 'Peugot', 'Citroen', 'Honda', 'Hyundai']
```

```
cars[-3:]
```

```
['Citroen', 'Honda', 'Hyundai']
```

```
for car in cars[:3]:
  print(car.upper())
```

```
AUDI
MASERATI
BUGATTI
```

# Copie d'une liste

Une liste peut être copiée et utilisée par la suite afin de laisser intacte la liste initiale. Une liste peut être copiée en utilise le slicing sans spécification de l'index de début ni celui de fin.

```
my_cars = cars[:]
```

```
my_cars.append('Fiat')
```

```
my_cars
```

```
['Audi',
 'Maserati',
 'Bugatti',
 'Mercedes',
 'Renault',
 'Peugot',
 'Citroen',
 'Honda',
 'Hyundai',
 'Fiat']
```

```
cars
```

```
['Audi',
 'Maserati',
 'Bugatti',
 'Mercedes',
 'Renault',
 'Peugot',
 'Citroen',
 'Honda',
 'Hyundai']
```

L'utilisation d'une copie à travers l'opérateur de l'affectation = ne permet pas de faire une copie, la variable crée n'est qu'un alias qui peut être utilisé à la place de la variable.

```
cars_list = cars
```

```
cars.append('Jaguar')
```

```
cars
```

```
['Audi',
 'Maserati',
 'Bugatti',
 'Mercedes',
 'Renault',
 'Peugot',
 'Citroen',
 'Honda',
```

```
        'Hyundai',
        'Jaguar']
```

```
cars_list
```

```
['Audi',
 'Maserati',
 'Bugatti',
 'Mercedes',
 'Renault',
 'Peugot',
 'Citroen',
 'Honda',
 'Hyundai',
 'Jaguar']
```

# Tuples

Les listes sont utilisées pour stocker les collections d'éléments qui peuvent changer tout au long du cycle de vie d'un programme. Les tuples sont des collections d'éléments qui ne peuvent pas changer. Les tuples sont des liste immutables.

### Définir un tuple

Les tuples sont définis de la même manière que les listes à l'exception des parenthèses qui sont utilisées à la place des crochets.

```
dimensions = (200, 50)
```

```
dimensions
```

```
(200, 50)
```

```
dimensions[0] = 250
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-00271e61a311> in <module>()
----> 1 dimensions[0] = 250

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

SEARCH STACK OVERFLOW

**Parcourir les éléments d'un tuple** On peut utiliser **for** pour parcourir les éléments d'un tuple.

```
for dimension in dimensions:
  print(dimension)
```

```
200
50
```