La boucle `for`

La boucle `for` est très pratique pour **boucler sur les éléments d'une collection**.

Les chaînes de caractères sont en quelque sorte une **collection** de caractères et on peut à ce titre boucler dessus de la même façon.

Enfin les ranges vus précédemment peuvent être itérées sans avoir à les convertir en liste.

```
ma_liste = [1, 2, 3, 4]
for nb in ma_liste:
    print(nb + 2) # Nous printons chaque chiffre dans ma_liste en ajoutant 2

ma_string = "python"
for char in ma_string:
    print(f"la lettre {char} !")

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

Pour accéder aux **index** des éléments lorsqu'on boucle sur une liste, il est pertinent d'utiliser `enumerate()`.

```
ma_liste = ['a', 'b', 'c', 'd', 'e']
for index, char in enumerate(ma_liste):
    print(f"la lettre {char} est à l'index {index}.")
```

📌 Vous remarquerez que la fonction `enumerate` permet à la fois d'obtenir la valeur de chaque élément de la liste mais également son index.

La boucle `for else`

Cette boucle se comporte comme une boucle `for` à l'exception qu'une branche `else` va pouvoir être placée en dessous du bloc `for` contenant un `break` et il s'exécutera si à aucun moment, la condition pour sortir de la boucle n'est remplie.

```
ma_liste = [False, False, False, False, False]

for booleen in ma_liste:
    if booleen is True:
        print("élément vrai trouvé!")
        break
else:
    print("Tous les éléments de la liste sont faux 😳")
```

La boucle `while`

La boucle `while` va répéter des instructions **tant que** sa condition de sortie n'est pas rempli.

```
total = 20
x = 0

while x < total:
    print(x)
    x += 1 # on n'oublie pas d'incrémenter x à chaque itération
```

📌 la notation `x++` n'existe pas en python

> ⚠️ Boucle infinie
>
> Attention à garantir une condition de sortie à votre boucle sous peine de créer une boucle infinie. Dans l'exemple précédent, si je n'incrémente pas `x` à chaque itération, il sera toujours inférieur à `total` et on ne sortira jamais de la boucle.

`continue` et `break`

Ces mot clés vont permettre de sortir de la boucle lorsqu'une condition n'est plus remplie au cours de l'itération.

`continue`

Permet d'empêcher la continuation de l'itération en cours et va passer à la suivante sans effectuer le reste du code de la boucle.

```
ma_liste = ["COUCOU", "PYTHON", "lowercase"]

# Chaque fois qu'on croise une string en uppercase on passe à l'itération suivante
sans exécuter le reste du code
for string in ma_liste:
    if string.isupper():
        continue
    print(string) # nous ne printons que "lowercase"
```

`break`

Permet de complètement sortir de la boucle courante.

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

for x in ma_liste:
    if x >= 4:
        print("C'en est trop pour moi, je m'en vais!")
        break
    print(x)
```

Les compréhensions de listes

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

Feature exclusive de python, elle permettent de boucler sur une liste et effectuer des opérations en utilisant une synthaxe plus compacte. **Cela nous retourne une nouvelle liste**.

Les applications courantes des "compréhensions" consistent à créer de nouvelles listes dans lesquelles chaque élément est le résultat de certaines opérations appliquées à chaque membre d'une autre séquence ou itérable, ou à créer une sous-séquence de ces éléments qui satisfont à une certaine condition.

Sans compréhension de liste, si nous voulons extraire les chiffres pairs d'une liste et les multiplier par 2, notre code pourrait ressembler à ceci.

```
ma_liste = [1, 2, 3, 4, 5]
liste_paire = []

for x in ma_liste:
    if x % 2 == 0: # modulo, permet de calculer le reste de la division
euclidienne entre x et 2
        liste_paire.append(x*2)

print(liste_paire) # [4, 8]
```

Avec une compréhension de liste, le code nécessaire est plus compact.

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

# structure : [<ACTION TRUE> for <ELEMENT> in <LISTE> if <CONDITION>]
liste_paire = [x*2 for x in ma_liste if x % 2 == 0]

print(liste_paire) # [4, 8]
```

> ⚠️ Compact ne rime pas avec qualité
>
> Les compréhensions de liste proposent une fonctionnalité intéressante pour faire des opérations sur des listes, mais comme pour les assignations parallèles et multiples de variables, ou l'opérateur ternaire, elles sont à utiliser avec mesure et doivent être évitées lorsque cela met en péril la lisibilité de votre code.
>
> ```
> sorted_datas = [data.strip(' ').split('/') for data in
> message.content.split(':')[1].split(',')]
> ```

`any` et `all`

Fonctionnement

Vont permettre de boucler sur une liste de conditions et de retourner `True` ou `False` si une des conditions, ou la totalité est `False`.

```
any_false = any([True, True, True, False])
all_true  = all([True, True, True])

print(any_false) # True au moins une condition est False
print(all_true)  # True car toutes les conditions sont True
```

Ces mots-clés peuvent être particulièrement utiles lorsqu'on a beaucoup de condition a vérifier dans le même `if`.

Exemple any

```
phone = "06 56 26 37 28"

if phone.startswith("06") or phone.startswith("07") or phone.startswith("04"):
    print("Numéro de téléphone valide")

# En utilisant any()
if any([phone.startswith("06"), phone.startswith("07"), phone.startswith("04")]):
    print("Numéro de téléphone valide")
```

Exemple all

```
mail = "mail@gmail.com"

if len(mail) > 8 and '@' in mail and mail.endswith(".com"):
    print("Mail valide")

# En utilisant all()
if all([len(mail) > 8, '@' in mail, mail.endswith(".com")]):
    print("Mail valide")
```

Exemples liste en compréhension

`any()` et `all()` peuvent être particulièrement puissants dans le cadre d'une compréhension de liste.

```
ma_liste = [1, 3, 5, 7, 9, 11, 13, 16]
print(all([x%2 != 0 for x in ma_liste]))

notes = [12, 13, 8, 18, 20]
if any([x >= 10 for x in notes]):
    print("un cancre est parmis nous!") # Au moins une des notes est en dessous de
10
```

Exercice: "16. Les boucles"