Dans les chapitres précédents, nous avons parcouru des séquences avec l'intention de parcourir tous leurs éléments. Même quand on n'a parcouru que les éléments d'indice pairs d'une séquence, on a parcouru en totalité la sous-séquence restreinte aux indices pairs.

Pour certains problèmes, on souhaite effectuer un calcul basé sur un parcours d'itérable, mais arrêter le parcours dès que la réponse au problème est connue.

1 Exemple de l'appartenance

On cherche à savoir si un caractère `c` appartient à une chaîne `chaine`, sans utiliser l'opérateur `in`.

1.1 Exemple de code correct, mais qui ne fait pas ce qu'on veut

Si on reprend les algorithmes des chapitres précédents, on écrira la fonction suivante :

```
def appartient_sans_interruption(c : str, chaine : str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient le caractère c.

    precondition : len(c) == 1
    Exemples :
    $$$ appartient_sans_interruption('e', 'panier')
    True
    $$$ appartient_sans_interruption('o', 'desert')
    False
    $$$ appartient_sans_interruption('o', '')
    False
    """
    trouve = False
    for car in chaine:
        if car == c:
            trouve = True
    return trouve
```

Cette fonction renvoie un résultat correct. En effet :

- si durant une itération on rencontre le caractère `c`, le booléen `trouve` passe à `True`
- si durant le parcours on ne rencontre jamais le caractère `c`, le booléen `trouve` reste à `False`.

Mais ce n'est pas le parcours qu'on veut. En effet on parcourt la séquence en entier. Alors qu'on pourrait très bien s'arrêter une fois le caractère trouvé. Dans ce cas on sait que la fonction renvoie `True` et il est inutile de continuer le parcours de la chaîne.

1.2 Exemple de code incorrect

Considérons cette version, qu'on trouve souvent dans les travaux étudiants.

```
def appartient_incorrect(c : str, chaine : str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient le caractère c.

    precondition : len(c) == 1
    Exemples :
    $$$ appartient_incorrect('r', 'panier')
    True
    $$$ appartient_incorrect('o', 'desert')
    False
    $$$ appartient_incorrect('o', '')
    False
    """
    trouve = False
    for car in chaine:
        if car == c:
            trouve = True
        else:
            trouve = False
    return trouve
```


Ce code est faux, et pourtant les tests ne détectent pas l'erreur.

Ce code est faux car il met à jour la valeur de `trouve` à chaque itération. Donc la valeur renvoyée par la fonction vaut `True` si le dernier élément est `c` et `False` sinon. Or il se trouve que l'appel `appartient_incorrect('r', 'panier')` utilise une chaine dans laquelle le caractère cherché est en dernière position. L'appel renvoie bien `True`, ce qui laisse croire que le code est correct. Par contre l'appel à `appartient_incorrect('r', 'paniers')` vaudra `False` car `s` est différent de `r`, alors qu'on s'attend à ce que l'appel vale `True`. Un test basé sur cet appel détecte l'erreur.

Cet exemple montre bien que les tests peuvent donner l'impression que le code d'une fonction est correcte alors que ce n'est pas le cas.

Dans notre exemple il valait mieux éviter de positionner uniquement l'élément à chercher en début ou en fin de chaîne. On peut tester avec le caractère :

- en début de chaîne
- en fin de chaîne
- en milieu de chaîne

1.3 Exemple avec un parcours interrompu, avec un `while`

Quand on arrête le parcours dès que le caractère à chercher est trouvé, le nombre d'itérations ne peut pas être calculé à l'avance. Nous allons donc utiliser une boucle `while`.

Le principe est le suivant :

- on utilise un indice `i` commençant à 0, qui parcourt les indices des caractères de la chaîne. Cet indice `i` doit rester strictement inférieur à `len(chaine)`, sinon il désignera un indice hors de la chaine ;
- on arrête le parcours des caractères une fois que le caractère `c` a été trouvé.

On a alors 2 raisons d'arrêter la boucle :

- soit `i` a atteint la valeur `len(chaine)` (et alors on a parcouru toute la chaîne sans trouver le caractère cherché)
- soit `chaine[i]` vaut `c` (et alors on a trouvé le caractère cherché)

L'expression qui déclenche l'**arrêt** de la boucle est donc un `or`.

Au contraire, on **continue** à itérer si `i` est strictement inférieur à `len(chaine)` **et** `chaine[i]` ne vaut pas `c`.

Le corps de la boucle consiste uniquement à aller voir l'élément suivant. On a donc :

```
def appartient(c : str, chaine : str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient le caractère c.

    precondition : len(c) == 1
    Exemples :
    $$$ appartient('e', 'panier')
    True
    $$$ appartient('o', 'desert')
    False
    """
    i = 0
    while i < len(chaine) and chaine[i] != c:
        i = i + 1
    ... # ???
```

Une fois la boucle terminée, on se demande quelle valeur renvoyer. On se pose la question de la raison de l'arrêt de la boucle : on rappelle qu'après la boucle on sait que la condition de boucle est fausse.

Après la boucle, la condition `i < len(chaine) and chaine[i] != c` est donc fausse. On se remémore la table de vérité de l'opérateur `and`. Une expression `x and y` vaut `False` dans 2 cas :

- `x` vaut `False` et dans ce cas `y` n'est pas évalué (opérateur paresseux, ce cas regroupe 2 lignes de la table de vérité du `and`)
- `x` vaut `True` et `y` vaut `False`

Donc :

- soit `i < len(chaine)` vaut `True` et `chaine[i] != c` vaut `False`. Dans ce cas il existe un `i` tel que `chaine[i] == c` vaut `True`, on a trouvé le caractère cherché. Donc la fonction doit renvoyer `True`.
- soit `i < len(chaine)` vaut `False` et `chaine[i] != c` n'est pas évalué. Dans ce cas, `i` a été incrémenté jusqu'à atteindre `len(chaine)` et `i == len(chaine)` vaut `True`. Toute la séquence a donc été parcourue sans trouver le caractère cherché. La fonction doit renvoyer `False`.

On constate dans les 2 cas précédents que :

- soit `i < len(chaine)` vaut `True` et alors la fonction renvoie `True`
- soit `i < len(chaine)` vaut `False` et alors la fonction renvoie `False`

La valeur de l'expression `i < len(chaine)` permet donc de déterminer pourquoi la boucle s'est terminée, de conclure sur l'appartenance ou non du caractère `c` à la chaîne `chaine` et sur la valeur à renvoyer.

On obtient alors la fonction suivante :

```
def appartient(c : str, chaine : str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient le caractère c.

    precondition : len(c) == 1
    Exemples :
    $$$ appartient('e', '')
    False
    $$$ appartient('e', 'panier')
    True
    $$$ appartient('o', 'desert')
    False
    """
    i = 0
    while i < len(chaine) and chaine[i] != c:
        i = i + 1
    return i < len(chaine)
```

NB : certain · es étudiant · es objectent que ce type de boucle est bien compliqué à comprendre et qu'il est bien plus simple de parcourir toute la séquence. Néanmoins la science des algorithmes informatiques est indissociable du calcul de la **complexité** des algorithmes, c'est à dire du temps ou des ressources mémoire nécessaires au calcul étant donnée la forme des données du problème. Pour des débutants, on peut se dire que si le caractère cherché est en 2ème position d'une chaine de taille 1 million, parcourir toute la séquence ou s'arrêter après 2 itérations fait une sacrée différence, encore plus quand le problème devient plus complexe qu'une simple recherche de caractère.

1.4 Exemple avec un parcours interrompu, avec un `for`

Dans le cas où le parcours à interrompre est inclus dans le corps d'une fonction qui calcule et renvoie un résultat, on peut utiliser une boucle `for` et l'interrompre avec l'instruction `return` associée à la fonction.

Néanmoins vous verrez au second semestre des procédures qui ne renvoient rien et trient la séquence passée en paramètre. Il n'y a alors pas d'instruction `return` dans la fonction, et c'est une boucle `while` qui sera utilisée dans l'algorithme. Comprendre le fonctionnement de la boucle `while` précédente est donc très important.

```
def appartient_for(c : str, chaine : str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient le caractère c.
    precondition : len(c) == 1
    Exemples :
    $$$ appartient_for('e', 'panier')
    True
    $$$ appartient_for('o', 'desert')
    False
    $$$ appartient_for('o', '')
    False
    """
    for car in chaine:
        if car == c:
            return True
    return False
```

Attention une erreur courante consiste à ajouter au `if` un `else: return False`. Dans ce cas la fonction termine à la première itération et seul le premier élément est examiné.

Attention pour cette fin de semestre vous devez savoir interrompre un parcours par une boucle `for` et une boucle `while`.

1.5 Autre exemple

Supposons qu'on veuille écrire une fonction qui renvoie `True` ssi la chaîne paramètre contient au moins `n` occurrences du caractère `'a'`, et qu'on veuille interrompre le parcours dès que le résultat de la fonction est connu.

Si on parcourt toute la chaîne en mettant à jour un compteur `cpt` qui compte le nombre de `'a'`, en renvoyant après le parcours la valeur de `cpt >= n`, alors le code de la fonction est correct. Il renvoie bien le résultat attendu. **MAIS** ce n'est pas ce qu'on veut.

Ce qu'on veut est interrompre le parcours de la chaîne **dès que** `cpt` atteint `n`, soit avec une boucle `while`, soit avec une boucle `for` et un `return`.

2 Cas des prédicats "pour tout" et "il existe"

La structure algorithmique de ces prédicats est un grand classique que vous devez apprendre.

2.1 Prédicat de type "il existe"

Un tel prédicat renvoie `True` ss'il existe au moins un élément dans l'itérable qui satisfait une propriété `P`.

Exemples de propriétés : `elt` est une majuscule, `elt` est positif…

On interrompt la boucle quand on a trouvé un élément qui satisfait `P` (on dit que cet élément est un exemple pour `P`).

Le schéma général pour la boucle `while` est alors :

```
i = 0
while i < len(iter) and not P(iter[i]): # iter[i] n'est pas un exemple pour P
    i = i + 1
return i < len(iter)
```

Le schéma général pour la boucle `for` est alors :

```
for elem in iter:
    if P(elt): # elt est un exemple pour P
        return True
return False
```


Université de Lille
Parcours interrompu d'itérable
2025-2026
UE PROG, L1S1 MI
Département Informatique de la Faculté des Sciences et Technologies

3 Prédicat de type "tous les elts sont tels que"

Un tel prédicat renvoie `True` ssi tous les éléments dans l'itérable satisfont une propriété `P`.

C'est le dual du problème "il existe".

Cette fois on interrompt le parcours quand on a trouvé un contre-exemple pour `P`, c'est à dire un élément qui ne satisfait pas `P`. Le prédicat renvoie alors `False`.

Le schéma général pour la boucle `while` est alors :

```
i = 0
while i < len(iter) and P(iter[i]): # on n'a pas de contre-exemple pour P
    i = i + 1
return i == len(iter)
```

Le schéma général pour la boucle `for` est alors :

```
for elem in iter:
    if not P(elt): # on a un contre-exemple pour P
        return False
return True
```


On découvre :

- les façons d'écrire une boucle `while` à ne pas utiliser
- les façons d'écrire une boucle `for` à ne pas utiliser
- quand utiliser une boucle `for` ou une boucle `while`
- les bonnes pratiques de nommage
- … dans le cadre de cette UE.

1 Nommage

D'un manière générale, le nom de la variable doit être porteur de sens.

De plus, certains noms de variables sont passés dans la pratique courante en informatique et sont réservés à certains usages :

- `i`, `j` ou `k` (ou `ind` et `indice` en français) représentent des indices
- `c` (ou `car` en français) représente un caractère
- `s` (ou `ch` en français) représente une chaîne de caractères
- `l` et `lst` (ou `liste` en français) représentent une liste
- *etc.*

C'est une très mauvaise pratique d'utiliser ces noms pour un autre usage.

Vous ne devez donc par exemple **pas** écrire :

```
for c in range(liste):
for i in chaine:
```

La bonne pratique est d'écrire :

```
for i in range(liste):
for c in chaine:
```

Pour un parcours de liste, le nom doit être évocateur du contenu de la liste. Si `liste` contient des mots, on écrira :

```
for mot in liste:
```

ou

```
for mot in lmots:
```

Pour un exercice utilisant une liste d'entiers sans autre précision, on pourra par exemple utiliser `elt` ou `elem` :

```
for elt in liste:
```

mais **pas**

```
for i in liste
```

2 Imbrication

Nous avons vu que les boucles `for` peuvent être imbriquées sans nuire à la lisibilité du code, notamment quand le code calcule un produit cartésien.

Quand le calcul est autre, il est plus judicieux d'utiliser une fonction auxiliaire qui code la boucle `for` interne. Par ex pour écrire une fonction qui supprime un caractère donné d'une liste de chaînes on écrira :

```
def supprime_tous_car(liste:list[str], car:str) -> list[str]:
    res = []
    for chaine in liste:
        res.append(supprime_car(chaine, car))
    return res
```

Le corps de `supprime_car` contient la 2ème boucle `for`.

Dans le cadre de ce cours, on évitera d'imbriquer une boucle `while` dans une autre boucle.

Si nécessaire, on préférera de même appeler une fonction auxiliaire – contenant une boucle `while` – dans une boucle.

3 Pas de `return` dans un while

On reprend l'exemple de l'amphi :

```
def contient_au_moins_une_majuscule_while(chaine:str) -> bool:
    """Renvoie True ssi la chaîne `chaine` contient au moins une majuscule.
    precondition : len(c) == 1
    Exemples :
    $$$ contient_au_moins_une_majuscule_while('sEl')
    True
    $$$ contient_au_moins_une_majuscule_while('sel')
    False
    $$$ contient_au_moins_une_majuscule_while('')
    False
    """
    i = 0
    while i < len(chaine) and not chaine[i].isupper():
        i = i + 1
    return i < len(chaine)
```

On pourrait être tenté d'utiliser une boucle `while` dont la condition est simplement `i < len(chaine)` et d'écrire dans le corps de la boucle quelque-chose comme :

```
if chaine[i].isupper():
    return True
```

On considère dans l'UE qu'un `return` dans une boucle `while` est une mauvaise pratique indiquant que la sémantique du `while` n'est pas comprise.

Au lieu d'indiquer quand on quitte la boucle, on indique quand elle continue : les boucles `while` sont faites pour ça. On utilise donc bien une expression composée dans la condition de boucle :
`while i < len(chaine) and not chaine[i].isupper()`

4 Quand utiliser un `for` ? un `while` ?

La boucle `while` est strictement plus puissante que la boucle `for`, c'est-à-dire que toute boucle `for` peut être traduite en une boucle `while`, mais que certaines boucles `while` ne peuvent pas être traduites en une boucle `for`.

Les boucles `for` sont faites pour parcourir un itérable. Elles conviennent quand le problème à résoudre contient un itérable (dont on connaît la taille) et qu'on sait donc calculer à l'avance le nombre d'itérations qui seront exécutées. Ici "à l'avance" signifie "avant d'exécuter la boucle".

les boucles `while` peuvent parcourir un itérable mais ne nécessitent pas d'itérable comme les boucles `for`. Elles sont plus générales, et permettent de coder des problèmes pour lesquels on ne peut pas calculer à l'avance le nombre d'itérations. Elle peuvent aussi coder tout ce qui est codé avec une boucle `for`.

Néanmoins, dans le cadre de cette UE, nous vous demandons de ne pas utiliser systématiquement des boucles `while`.

4.1 Quand on n'a pas d'itérable

C'est le cas facile : on utilise une boucle `while` !

Cette boucle est justement faite pour s'arrêter quand une condition quelconque devient fausse et non quand on a parcouru tous les éléments d'un itérable.

Par exemple :

Université de Lille
Boucles et bonnes pratiques dans l'UE
2025-2026
UE PROG, L1S1 MI

- on fait un calcul impliquant une suite et un seuil (ex : au bout de combien de jour la taille du blob atteindra ou dépassera 1m ? ou itérer tant que la taille du blob est strictement inférieure à 1m)
- on fait une saisie clavier avec vérification de la saisie (ex : demander la saisie d'un entier positif et itérer tant que la valeur saisie n'est pas correcte)
- on souhaite produire `n` valeurs aléatoires contraintes (ex : tirer aléatoirement 10 valeurs différentes comprises entre 1 et 100 - on ne sait pas combien de tirages aléatoires seront nécessaires, itérer tant que le nombre de valeurs différentes n'est pas 10)

4.2 Quand on parcourt les éléments d'une séquence de manière séquentielle

Dans l'UE on a uniquement vu des algorithmes qui parcourent une séquence (une liste, une chaîne, un intervalle) de manière séquentielle, c'est à dire en appliquant à tous ses éléments de la gauche vers la droite le même traitement à chaque itération.

La boucle `for` permet de programmer ces algorithmes, en utilisant la séquence appropriée. On peut, par exemple, parcourir une liste :

- de la gauche vers la droite
- de la droite vers la gauche en utilisant une sous-partie de la liste et un pas négatif
- en considérant uniquement les éléments d'indices pairs, avec un pas bien choisi…

4.2.1 Parcours des éléments

En semaine 6, on a vu des parcours qui examinent uniquement les éléments d'une séquence, sans passer par leur indice. Par exemple :

- calculer la somme des éléments d'une liste
- calculer l'élément maximum d'une liste
- découper une chaîne selon des séparateurs donnés

On a utilisé une boucle `for` car utiliser une boucle `while` oblige à passer par un parcours sur les indices de l'itérable alors que l'indice n'est pas nécessaire. Utiliser les indices n'améliore pas la lisibilité de la boucle ni du reste du code.

4.2.2 Autres parcours

Dans les autres cas on a utilisé soit un `for`, soit un `while`.

Quand il est nécessaire d'itérer sur les indices d'un itérable et non ses éléments, par exemple :

- l'indice apparaît dans la spécification (ex : calculer l'indice du maximum)
- parcours de 2 séquences "en parallèle", en appariant les éléments de même indice dans les 2 séquences.

Quand on a besoin d'itérer sur les paires d'éléments contigus d'un itérable avec un `while` ou un `for` + `range` ou une variable locale…

Quand on effectue un parcours interrompu de séquence (semaine 9) avec un `while` ou un `for` + `return`

Vous verrez au S2 des procédures qui nécessitent l'utilisation d'une boucle `while`.

4.3 Quand le parcours est non séquentiel

Parfois on peut avoir besoin de parcourir une séquence en calculant pendant l'itération l'indice du prochain élément à traiter en fonction des valeurs de la séquence.

Par exemple, la fonction suivante parcourt les éléments d'une liste en fonction de la valeur de l'élément courant de la liste.

```
def extraction(liste:list[int]) -> list[int]:
    """
    Le premier indice considéré est 0 (si possible).
    Ensuite :
    On garde dans le résultat l'élément à cet indice, mettons `n1`,
    puis on saute à l'élément d'indice `n1` (s'il existe, sinon arrêt).
```

Département Informatique de la Faculté des Sciences et Technologies

```
    Ensuite :
    On garde dans le résultat l'élément à cet indice, mettons `n2`,
    puis on saute à l'élément d'indice `n1+n2` (s'il existe, sinon arrêt).
    etc...

    Précondition : les éléments sont positifs strictement
    Exemple(s) :
    $$$ extraction([])
    []
    $$$ extraction([2, 1, 3, 2, 2, 1, 2, 5, 3])
    [2, 3, 1, 2, 3]
    $$$ extraction([2, 1, 3, 2, 2, 1, 2, 5])
    [2, 3, 1, 2]
    $$$ extraction([100, 2])
    [100]
    """
    res = []
    i = 0
    while 0 <= i and i < len(liste):
        elem = liste[i]
        res.append(elem)
        i = i + elem
    return res
```

Dans ce cas on ne pourra pas utiliser un `for` car la séquence à parcourir séquentiellement doit être calculée en fonction des valeurs de la liste, qui sont connues pendant l'itération. On ne peut donc pas calculer à l'avance la séquence à parcourir, donc la boucle `for` n'est pas utilisable.

Vous verrez d'autres exemples célèbres de parcours dont les indices sautent un peu partout dans l'itérable au S2 (recherche dichotomique).