# Une liste est un objet auquel on peut passer des messages

## Doc. 3,3

**Objectif**

Faire grandir et rétrécir une liste en lui envoyant des messages, en particulier `append` et `pop`, et savoir si un message modifie la liste ou renvoie une valeur.

**Q1.** On veut noter la liste des courses au fur et à mesure qu'on y pense. Au départ, on n'a pensé à rien : la liste est vide. Prévoir ce qui se passe lorsqu'on saisit les deux instructions suivantes dans la console, puis vérifier :

```
>>> courses = []
>>> courses[0] = "pain"
```
Python

Dans le Doc. 3,2, on contournait la difficulté en fabriquant d'abord une liste de la bonne taille, par exemple `[0] * 6`, puis en la remplissant. Cela suppose de connaître la taille à l'avance. Pour une liste de courses, ce n'est pas le cas : on ne sait pas combien de choses on va y écrire. Il faut un moyen de dire à la liste « ajoute cet élément au bout ».

## §1. Envoyer un message à une liste

### §1.1. Objets, messages et méthodes

En Python, chaque valeur est un **objet** : un nombre, une chaîne de caractères, un tuple, une liste. Pour l'instant, retenons qu'un objet regroupe des données et des actions qu'il sait effectuer sur ces données. Les objets seront étudiés en détail en terminale. Ici, on apprend seulement à s'en servir : pour demander à un objet d'effectuer l'une de ses actions, on lui **envoie un message**. On écrit le nom de l'objet, un point, puis le message suivi de parenthèses :

```
courses.append("pain")
```
Python

Cette instruction se lit : « on envoie à la liste `courses` le message `append` (ajouter, en anglais), avec l'argument `"pain"` ». La liste reçoit le message et réagit : elle place `"pain"` à sa fin. La façon dont un objet réagit à un message s'appelle une **méthode**. On dit indifféremment « envoyer le message `append` » ou « appeler la méthode `append` ».

Ce n'est pas la première fois que vous envoyez un message. Dans le Doc. 2,2, l'écriture `"...{}...".format(...)` envoyait le message `format` à une chaîne de caractères.

Comparons avec une fonction. Pour connaître la longueur d'une liste, on écrit `len(courses)` : on **donne** la liste à la fonction `len`, qui l'examine. Avec `courses.append("pain")`, on **s'adresse** à la liste elle-même, et c'est elle qui se modifie.

**Q2.** Saisir les instructions suivantes dans la console, en prévoyant le résultat de chaque ligne avant de la valider :

```
>>> courses = []
>>> courses.append("pain")
>>> courses
```
Python

```
>>> len(courses)
>>> courses.append("lait")
>>> courses.append("œufs")
>>> courses
>>> courses[2]
```

À quel indice se trouve l'élément qu'on vient d'ajouter avec `append` ? Exprimer cet indice à l'aide de `len(courses)`.

**À retenir**

En Python, une liste est un **objet**. On lui envoie un **message** avec la notation pointée : `liste.message(arguments)`. La façon dont l'objet réagit au message s'appelle une **méthode**.

`t.append(x)` ajoute l'élément `x` à la fin de la liste `t`. La liste est modifiée, sa longueur augmente de 1. Cela fonctionne aussi à partir de la liste vide `[]`.

## §1.2. Construire une liste en partant de la liste vide

La méthode `append` permet une façon de construire une liste qu'on utilisera très souvent : partir de la liste vide, puis ajouter les éléments un par un, au fil d'une boucle.

**Exemple résolu**
On veut la liste des multiples de `k` compris entre 1 et `n`.

```python
def multiples(n: int, k: int) -> list[int]:
    """Retourne la liste des multiples de k compris entre 1 et n.

    >>> multiples(20, 3)
    [3, 6, 9, 12, 15, 18]
    """
    resultat = []                # 1. on part de la liste vide
    for i in range(1, n + 1):    # 2. on examine chaque entier de 1 à n
        if i % k == 0:           # 3. s'il convient...
            resultat.append(i)   #    ... on l'ajoute au bout de la liste
    return resultat              # 4. on retourne la liste construite
```

Suivons l'appel `multiples(20, 3)`. Au départ, `resultat` vaut `[]`. Pour `i` égal à 1 et 2, le test est faux : rien ne se passe. Pour `i` égal à 3, le test est vrai : la liste devient `[3]`. Elle devient `[3, 6]` au tour où `i` vaut 6, et ainsi de suite. On n'a jamais eu besoin de savoir combien de multiples on allait trouver : la liste grandit d'elle-même.

**Q3.** En suivant la même méthode, écrire la fonction `diviseurs` dont la spécification est :

```python
def diviseurs(n: int) -> list[int]:
```

```
    """Retourne la liste des diviseurs de l'entier n, avec n >= 1,
    rangés dans l'ordre croissant.

    >>> diviseurs(12)
    [1, 2, 3, 4, 6, 12]
    >>> diviseurs(7)
    [1, 7]
    """
```

Pourquoi la méthode `[0] * taille` du Doc. 3,2 serait-elle mal adaptée ici ?

**Q4.** **Retour sur le Loto.** Dans le Doc. 3,2, la fonction `loto` partait de `[0] * 6` et utilisait une variable `nbre_tirees` pour savoir où ranger le prochain numéro. Réécrire cette fonction en partant de la liste vide et en utilisant `append` :

```python
def loto() -> list[int]:
    """Retourne une liste de 6 entiers différents, tirés au hasard
    entre 1 et 49 (bornes incluses).
    """
```

Quelle variable de l'ancienne version devient inutile ? Par quoi est-elle remplacée ?

**Coups de pouce**

1. On ne sait pas combien de tirages il faudra, puisqu'un numéro déjà sorti est rejeté. La boucle reste donc une boucle `while`.
2. La boucle doit s'arrêter quand la liste contient 6 numéros. Quelle expression donne le nombre de numéros déjà rangés ?

## §2. Retirer un élément : le message pop

Le message `pop` fait l'inverse de `append` : il retire un élément de la liste. Mais il fait aussi autre chose, que l'on va découvrir.

Q5. Saisir les instructions suivantes dans la console, en prévoyant le résultat de chaque ligne avant de la valider :

```Python
>>> pile = [3, 7, 12, 5]
>>> pile.pop()
>>> pile
>>> x = pile.pop(0)
>>> x
>>> pile
>>> pile.pop()
>>> pile.pop()
>>> pile.pop()
```

1. Quel élément `pop()` retire-t-il lorsqu'on ne lui donne pas d'argument ? Et `pop(0)` ?
2. Contrairement à `append`, la ligne `pile.pop()` affiche quelque chose. Que représente ce qui est affiché ?
3. Pourquoi la dernière ligne provoque-t-elle une erreur ?

**À retenir**

`t.pop()` retire le dernier élément de la liste `t` et le renvoie. `t.pop(i)` retire l'élément d'indice `i` et le renvoie. Dans les deux cas, la longueur de la liste diminue de 1.

Envoyer `pop` à une liste vide provoque l'erreur `IndexError`. Avant un `pop`, il faut donc s'assurer que la liste n'est pas vide, par exemple avec `len(t) > 0`.

Les deux messages `append` et `pop()` agissent sur la **fin** de la liste. Utilisés ensemble, ils font de la liste une pile d'assiettes : on pose une assiette sur le dessus, on reprend toujours celle du dessus. C'est exactement ce que fait la commande « Annuler » d'un traitement de texte. Chaque action est ajoutée au bout d'un historique. Annuler, c'est retirer la dernière action effectuée.

**Q6.** **Sans exécuter le programme**, prévoir les affichages du programme suivant. Écrire le contenu de la liste `historique` après chaque ligne qui la modifie. Vérifier ensuite.

```python
historique = []
historique.append("écrire Bonjour")
historique.append("mettre en gras")
historique.append("écrire Monde")
derniere = historique.pop()
print("Annulé :", derniere)
historique.append("souligner")
derniere = historique.pop()
print("Annulé :", derniere)
derniere = historique.pop()
print("Annulé :", derniere)
print(historique)
```

**Q7.** Un élève veut écrire une fonction qui retourne les éléments d'une liste dans l'ordre inverse. Il a l'idée de vider la liste avec `pop()`, en ajoutant au fur et à mesure chaque élément retiré dans une nouvelle liste :

```python
def inverser(t: list[int]) -> list[int]:
    """Retourne une nouvelle liste formée des éléments de t
    pris dans l'ordre inverse.

    >>> inverser([12, 15, 9, 14])
    [14, 9, 15, 12]
    """
    resultat = []
```

```
    while len(t) > 0:
        resultat.append(t.pop())
    return resultat
```

1. Recopier la fonction, puis saisir dans la console :

```
>>> releve = [12, 15, 9, 14]
>>> r = inverser(releve)
>>> r
>>> releve
```

   La fonction retourne-t-elle le bon résultat ? Quel autre effet a-t-elle ?
2. Expliquer cet effet à l'aide du Doc. 3,2.
3. Corriger la fonction pour que la liste reçue en argument ne soit pas modifiée, en conservant l'idée de l'élève.

**Coups de pouce**

1. Au moment de l'appel `inverser(releve)`, combien de listes existent en mémoire ? Combien de noms les référencent ?
2. Le message `pop` modifie la liste qui le reçoit. Sur quelle liste faudrait-il l'envoyer pour que `releve` ne soit pas touchée ?

## §3. Un message qui modifie la liste ne renvoie rien

**Q8.** Une élève veut ajouter le lait à sa liste de courses. Elle écrit :

```
>>> courses = ["pain"]
>>> courses = courses.append("lait")
>>> print(courses)
None
```

Sa liste a disparu. Elle essaie alors de ranger des notes dans l'ordre croissant avec le message `sort` (trier) :

```
>>> notes = [12, 15, 9, 14]
>>> print(notes.sort())
None
>>> notes
[9, 12, 14, 15]
```

1. Que renvoie le message `append` ? Et le message `sort` ?
2. Expliquer, étape par étape, ce qui s'est passé lors de l'instruction `courses = courses.append("lait")`.
3. Comment fallait-il écrire cette instruction ?

Cette erreur est l'une des plus fréquentes chez les débutants, et chez les autres. Elle vient d'une confusion entre deux sortes de messages. Certains messages, comme `append` ou `sort`, **modifient** la liste et renvoient `None`. D'autres, comme `pop`, modifient la liste **et** renvoient une valeur utile. Il en existe enfin qui ne modifient rien et se contentent de renvoyer une information sur la liste, comme on va le voir dans la partie 4.

Puisque `append` modifie la liste elle-même, tous les alias de cette liste voient la modification. L'opérateur `+`, qui met deux listes bout à bout, se comporte autrement.

**Q9.** Prévoir, puis vérifier, le contenu de `n` et de `m` à la fin de chacune des deux séquences :

```
>>> n = [1, 2, 3]
>>> m = n
>>> m.append(4)
>>> n
>>> m
```

```
>>> n = [1, 2, 3]
>>> m = n
>>> m = m + [4]
>>> n
>>> m
```

Expliquer la différence.

**À retenir**

Un message qui modifie la liste, comme `append` ou `sort`, renvoie `None`. On l'écrit seul sur sa ligne, **sans affectation** : `t.append(x)` et jamais `t = t.append(x)`.

`pop` modifie la liste et renvoie l'élément retiré, qu'on peut ranger dans une variable : `x = t.pop()`.

Un message envoyé à une liste agit sur la liste elle-même : tous ses alias voient la modification. L'expression `t + [x]`, elle, fabrique une nouvelle liste.

## §4. D'autres messages

Une liste comprend d'autres messages que `append` et `pop`. La commande `help(list)`, dans la console, en donne la liste complète, en anglais. En voici quelques-uns, traduits :

| Message | Effet | Renvoie |
|---|---|---|
| `t.insert(i, x)` | insère `x` juste avant l'élément d'indice `i` | `None` |
| `t.remove(x)` | retire la première occurrence de `x` ; erreur `ValueError` si `x` est absent | `None` |
| `t.index(x)` | ne modifie pas la liste ; erreur `ValueError` si `x` est absent | l'indice de la première occurrence de `x` |
| `t.count(x)` | ne modifie pas la liste | le nombre d'occurrences de `x` |
| `t.reverse()` | renverse l'ordre des éléments | `None` |
| `t.sort()` | range les éléments dans l'ordre croissant | `None` |
| `t.copy()` | ne modifie pas la liste | une nouvelle liste, copie de `t` |

**Q10.** À l'aide de ces descriptions, et **sans exécuter les instructions**, prévoir le résultat de chaque ligne. Indiquer pour chacune si la liste `t` est modifiée. Vérifier ensuite dans la console.

```python
>>> t = [5, 8, 2, 8]
>>> t.insert(1, 10)
>>> t
>>> t.remove(8)
>>> t
>>> t.index(2)
>>> t.count(8)
>>> t.remove(7)
>>> t.reverse()
>>> t
```

**Remarque.** Dans la question sur la fonction `inverser`, on aurait pu écrire `copie = t.copy()` à la place de `copie = [x for x in t]`. Les deux instructions fabriquent une nouvelle liste contenant les mêmes éléments.

**À retenir**

Avant d'utiliser un message, il faut savoir répondre à deux questions : modifie-t-il la liste ? que renvoie-t-il ?

`insert`, `remove`, `reverse` et `sort` modifient la liste et renvoient `None`. `pop` la modifie et renvoie l'élément retiré. `index`, `count` et `copy` ne la modifient pas et renvoient une information ou une nouvelle liste.

## §5. Exercices de synthèse

**Consigne**

Écrire chaque fonction dans la partie principale de l'éditeur, avec sa spécification complète (signature et documentation). Tester ensuite chacun des exemples de la documentation dans la console. Aucune de ces fonctions ne doit modifier la liste qu'elle reçoit en argument.

**Q11. Sans doublons.** Écrire la fonction `sans_doublons` dont la spécification est :

```python
def sans_doublons(t: list[int]) -> list[int]:
    """Retourne une nouvelle liste contenant les éléments de t,
    chacun une seule fois, dans l'ordre de leur première apparition.

    >>> sans_doublons([3, 1, 3, 2, 1])
    [3, 1, 2]
    >>> sans_doublons([])
    []
    """
```

Pourquoi ne peut-on pas obtenir ce résultat simplement avec une liste en compréhension filtrée par `if x not in t` ?

**Q12.** **Pairs et impairs.** Écrire la fonction `separer` dont la spécification est :

```python
def separer(valeurs: list[int]) -> tuple[list[int], list[int]]:
    """Retourne un tuple (pairs, impairs) : pairs est la liste des
    éléments pairs de valeurs, impairs celle des éléments impairs,
    dans l'ordre où ils apparaissent dans valeurs.

    >>> separer([4, 7, 10, 3, 8])
    ([4, 10, 8], [7, 3])
    """
```

Tester ensuite, dans la console, l'affectation multiple `p, i = separer([4, 7, 10, 3, 8])` (Doc. 3,1).

## §5.1. Pour aller plus loin

**Q13.** **Une file d'attente.** À un guichet, les clients sont servis dans leur ordre d'arrivée : le premier arrivé est le premier servi. On décrit une matinée par une liste d'événements. Un prénom signale l'arrivée d'un client, qui se place au bout de la file. Le caractère `"*"` signale que le guichetier sert un client : celui qui attend depuis le plus longtemps.

```python
def guichet(evenements: list[str]) -> tuple[list[str], list[str]]:
    """Retourne un tuple (servis, en_attente) : servis est la liste
    des clients servis, dans l'ordre où ils l'ont été, en_attente
    celle des clients qui attendent encore à la fin.
    On suppose que la file n'est jamais vide quand "*" survient.

    >>> guichet(["Ana", "Bilal", "*", "Chloé", "Dylan", "*"])
    (['Ana', 'Bilal'], ['Chloé', 'Dylan'])
    """
```

1. Écrire la fonction `guichet`.
2. Un élève a écrit `pop()` au lieu de `pop(0)`. **Sans exécuter son programme**, prévoir ce que renvoie l'appel de la documentation. Pourquoi les clients ont-ils raison de se plaindre ?

**Coups de pouce**

1. La file est une liste qui part vide. Que faut-il lui envoyer quand un client arrive ?
2. Le client servi est celui qui est arrivé le premier. À quel indice de la file se trouve-t-il ?
3. Le client servi doit à la fois quitter la file et rejoindre la liste `servis`. Un seul message renvoie l'élément qu'il retire.

## §6. Bilan

**À l'issue de cette activité, je sais...**

- expliquer la différence entre appeler une fonction, comme `len(t)`, et envoyer un message à une liste, comme `t.append(x)` ;
- construire une liste en partant de la liste vide et en ajoutant ses éléments avec `append` dans une boucle ;
- retirer le dernier élément d'une liste, ou celui d'indice `i`, avec `pop`, et utiliser l'élément renvoyé ;
- éviter d'envoyer `pop` à une liste vide ;
- dire, pour un message donné, s'il modifie la liste et ce qu'il renvoie, et ne jamais écrire `t = t.append(x)` ;
- prévoir l'effet d'un message envoyé à une liste qui a plusieurs alias, notamment lorsqu'elle est passée en argument à une fonction ;
- utiliser une liste comme une pile (`append` et `pop()`) ou comme une file (`append` et `pop(0)`).

Dans le Doc. 3,4, on utilise les listes et les tuples pour simuler un gaz avec Pyxel.