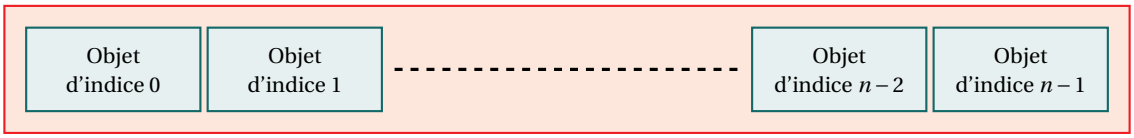
TP 2 INFO Structures de données séquentielles

L'objectif de ce TP est d'étudier de nouvelles structures de données qui sont plus élaborées que celles vues jusqu'à présent (entiers, flottants et booléens) : les **structures de données séquentielles** (ou linéaires). De manière schématique, une structure de données est dite séquentielle si elle est un assemblage de plusieurs objets que l'on peut représenter sous la forme d'un tableau en ligne.



Représentation schématique d'une structure de données séquentielle contenant n objets

Dans ce TP, pour chaque fonction rédigée, on ajoutera des spécifications et des assertions afin de vérifier le type des arguments en entrée et les préconditions portant sur ces arguments.

Partie I Objets immuables et objets mutables

Les objets se classent en deux catégories :

- les objets **immuables** : ce sont ceux que l'on ne peut pas modifier après leur création ;
- les objets **mutables** : ce sont ceux que l'on peut modifier après leur création.

Rappelons que la fonction `id` permet de récupérer l'adresse en mémoire d'un objet. Étudions l'exemple suivant.

```
>>> a = 1000 # On définit une variable a
>>> b = a # On définit une variable b avec a
>>> id(a), id(b) # Les variables a et b pointent vers le même emplacement en mémoire
(133819872603120, 133819872603120)

>>> a = 999 # On affecte une nouvelle valeur à la variable a
>>> a, b # La valeur de a a changé mais pas celle de b
(999, 1000)
>>> id(a), id(b) # Adresses mémoires de a et b
(133819872602960, 133819872603120)
```

On constate que l'instruction `a = 999` n'a pas modifié la valeur stockée en mémoire à l'emplacement pointé par la variable `a` au début du programme : Python a créé un nouvel objet en faisant pointer la variable `a` vers un nouvel emplacement en mémoire où il a stocké la valeur `999`.

Remarques 1 :

a) Les entiers, les flottants et les booléens sont des objets immuables.

b) Dans ce TP, nous allons voir des nouveaux objets immuables : les chaînes de caractères et les tuples.

c) Dans ce TP, nous allons voir des nouveaux objets mutables : les listes et les dictionnaires.

Partie II Les chaînes de caractères

II.A - Généralités

Les **chaînes de caractères** (*string* en anglais) sont représentées en Python par le type `str`. Une chaîne de caractères est un assemblage de caractères (lettres, chiffres et d'autres symboles) encadré par des guillemets doubles. Il est aussi possible d'utiliser des guillemets simples, mais nous éviterons de le faire afin d'éviter d'éventuels conflits avec le symbole apostrophe qui peut être présent dans la chaîne de caractères.

Exemple 1 : Les instructions ci-dessous définissent deux chaînes de caractères.

```
>>> chaine = "J'adore Python !" # Définition d'une chaîne de caractères
>>> chaine_vide = "" # Définition de la chaîne de caractères vide
```

Pour récupérer la **longueur** d'une chaîne de caractères, on utilise la fonction `len`.

```
>>> chaine = "J'adore Python !" # Définition d'une chaîne de caractères
>>> len(chaine) # Longueur de la chaîne de caractères chaine
16
```

Les chaînes de caractères sont des objets **itérables** : on peut les parcourir en utilisant une boucle `for` comme sur l'exemple ci-dessous.

```
>>> chaine = "test" # Définition d'une chaîne de caractères
>>> for caractere in chaine:
...         print(caractere)
t
e
s
t
```

Question 1 : Écrire une fonction `compter(chaine:str, car:str) -> int` prenant en argument un caractère `car` et qui renvoie le nombre d'occurrences de `car` dans `chaine`. Par exemple, l'exécution de la commande `compter("Papa est dans la cuisine", "a")` renvoie `4`.

On peut **accéder** aux éléments d'une chaîne de caractères en utilisant son **indice**.

```
>>> chaine = "J'adore Python !" # Définition d'une chaîne de caractères
>>> chaine[10] # Élément de chaine d'indice 10
't'
>>> chaine[20] # Renvoie une erreur car chaine n'a pas d'élément d'indice 20
IndexError: string index out of range
```

ATTENTION : Le premier élément d'une chaîne de caractères est d'indice 0. On en déduit que si la chaîne de caractères est de longueur n, alors les indices valides sont $0, 1, \ldots, n-1$.

ATTENTION : Pour accéder aux éléments d'une chaîne de caractères, on ne peut pas remplacer les crochets par des parenthèses : ces dernières sont utilisées exclusivement pour appeler une fonction.

```
>>> chaine = "J'adore Python !" # Définition d'une chaîne de caractères
>>> chaine(10) # Renvoie une erreur car on a utilisé des parenthèses
TypeError: 'str' object is not callable
```

Les chaînes de caractères sont des objets **immuables** : on ne peut pas modifier les éléments d'une chaîne de caractères, comme on peut le vérifier sur l'exemple ci-dessous.

```
>>> chaine = "J'adore Python !" # Définition d'une chaîne de caractères
>>> chaine[10] = a # Renvoie une erreur car on ne peut pas modifier chaine
TypeError: 'str' object does not support item assignment
```

Question 2 : Écrire une fonction `indice(chaine:str, car:str) -> int` prenant en argument un caractère `car` et qui renvoie l'indice de la première occurrence de `car` dans `chaine` si elle existe et `-1` sinon. Par exemple, l'exécution de la commande `indice("Papa est dans la cuisine", "a")` renvoie `1`.

Question 3 : Écrire une fonction `est_sous_chaine(chaine1:str, chaine2:str) -> bool` qui renvoie un booléen indiquant si la chaîne de caractères `chaine1` est un « morceau » de la chaîne de caractères `chaine2`. Par exemple, l'exécution de la commande `est_sous_chaine("tra", "extraordinaire")` renvoie **`True`**.

On peut effectuer des **extractions de tranche** d'une chaîne de caractères `ch` avec les différentes expressions

`ch[a:]`, `ch[:b]`, `ch[a:b]`, `ch[a::p]`, `ch[:b:p]`, `ch[::p]` ou `ch[a:b:p]`.

- Le paramètre `a` permet d'indiquer un indice de début (inclus) pour la tranche. Si le paramètre `a` n'est pas précisé, la tranche commence au début de la chaîne de caractères `ch`.
- Le paramètre `b` permet d'indiquer un indice de fin (exclu) pour la tranche. Si le paramètre `b` n'est pas précisé, la tranche se termine à la fin de la chaîne de caractères `ch`.
- Le paramètre `p` permet d'indiquer un pas pour la tranche, i.e. de préciser l'écart pour deux indices consécutifs de la tranche. Si le paramètre `p` n'est pas précisé, il est égal à 1 par défaut.

Exemple 2 : Voici quelques exemples de tranches pour une chaîne de caractères.

```
>>> ch = "0123456789" # On définit une chaîne de caractères ch
>>> ch[7:] # Chaîne des éléments de ch d'indice >= 7
'789'
>>> ch[2:7] # Chaîne des éléments de ch d'indice >= 2 et < 7
'23456'
>>> ch[2:8:2] # Chaîne des éléments de ch d'indice >= 2 et < 8 avec un pas de 2
'246'
>>> ch[::-1] # Chaîne des éléments de ch avec un pas de -1
'9876543210'
```

Question 4 : On considère la chaîne de caractères `chaine = "0123456789"`. Déterminer les expressions qui afficheront les résultats suivants en utilisant des tranches de `chaine`.

`"0123456789"` `"5"`	`"345"` `"789"`	`"02468"` `"13579"`	`"86420"` `"9753"`

Question 5 : Écrire une fonction `palindrome(chaine:str) -> bool` qui renvoie un booléen indiquant si `chaine` est un palindrome. Par exemple, l'exécution de la commande `palindrome("radar")` renvoie **`True`**.

Question 6 : Réécrire une seconde version « plus simple » de la fonction `est_sous_chaine(chaine1, chaine2)` définie dans la question 3 en utilisant des tranches de la chaîne de caractères `chaine2`.

II.B - Opérations arithmétiques sur les chaînes de caractères

On peut fabriquer une nouvelle chaîne de caractères en **concaténant** les chaînes de caractères `chaine1` et `chaine2` avec l'expression `chaine1 + chaine2`.

```
>>> chaine1 = "abc" # On définit une chaîne de caractères chaine1
>>> chaine2 = "uvwxyz" # On définit une chaîne de caractères chaine2
>>> chaine1 + chaine2 # Concaténation des chaînes de caractères chaine1 et chaine2
'abcuvwxyz'
```

Question 7 : Écrire une fonction `supprimer(chaine:str, car:str) -> str` prenant en argument un caractère `car` et qui renvoie la chaîne de caractères obtenue de `chaine` en supprimant toutes les occurrences de `car`. Par exemple, l'exécution de la commande `supprimer("Papa est là", "a")` renvoie `"Pp est là"`.

Étant donnés un entier positif `n` et une chaîne de caractères `chaine`, on peut effectuer la **multiplication** de `n` et de `chaine` en utilisant l'expression `n * chaine` : cette opération concatène `n` fois `chaine` avec elle-même.

```
>>> chaine = "abc" # Définition d'une chaîne de caractères
>>> 3 * chaine
'abcabcabc'
```

Question 8 : Écrire une fonction `triangle(n:int) -> None` qui affiche un triangle isocèle « croissant » de hauteur `n` composé de `#`. Voici un exemple ci-dessous.

```
>>> triangle(3)
#
##
###
```

II.C - Conversion de type

En Python, il est possible de faire certaines conversions de type. Il faut employer ces conversions avec des précautions : il n'est pas toujours simple d'en prédire le comportement.

Exemple 3 : Voici quelques exemples de conversions de type.

```
>>> a = 701
>>> str(a)
'701'
```

```
>>> a = 2.5
>>> str(a)
'2.5'
```

```
>>> chaine = "79"
>>> int(chaine)
79
```

```
>>> chaine = "1.2"
>>> float(chaine)
1.2
```

Dans certains cas, la conversion n'est pas possible.

```
>>> chaine = "10a"
>>> int(chaine)
ValueError: invalid literal for int() with base 10: '10a'
```

Remarque 2 : La conversion d'un nombre en chaîne de caractères permet d'accéder facilement aux chiffres de ce nombre.

Question 9 : Écrire une fonction `nb_chiffres(a:int) -> int` qui renvoie le nombre de chiffres de l'entier `a`.

Partie III Les tuples

III.A - Généralités

Les **tuples** sont représentés en Python par le type `tuple`. Un tuple est un ensemble d'objets de types quelconques (non nécessairement tous identiques) qui commence par une parenthèse ouvrante, termine par une parenthèse fermante et où tous les objets sont séparés par une virgule. Un tuple peut contenir plusieurs fois le même objet et même d'autres tuples. Les propriétés des tuples sont très proches de celles des chaînes de caractères.

Exemple 4 : Les instructions ci-dessous définissent deux tuples.

```
>>> t = (1, 3.15, -2, "Python", True, (1, 2, 3)) # Définition d'un tuple
>>> t_vide = () # Définition du tuple vide
```

Pour récupérer la **longueur** d'un tuple, on utilise la fonction `len`.

```
>>> t = (1, 3.15, -2, "Python", True, (1, 2, 3)) # Définition d'un tuple
>>> len(t) # Longueur du tuple t
6
```

Les tuples sont des objets **itérables** : on peut les parcourir en utilisant une boucle `for` comme ci-dessous.

```
>>> t = ("pi", 3.14, (3, 1, 4)) # Définition d'un tuple
>>> for element in t:
...         print(element)
pi
3.14
(3, 1, 4)
```

Question 10 : Écrire une fonction `somme(t:tuple) -> int` prenant en argument un tuple d'entiers `t` et qui renvoie la somme des éléments de `t`.

On peut **accéder** aux éléments d'un tuple en utilisant son **indice**.

```
>>> t = (1, 3.15, -2, "Python", True, (1, 2, 3)) # Définition d'un tuple
>>> t[2] # Élément du tuple t d'indice 2
-2
>>> t[8] # Renvoie une erreur car le tuple t n'a pas d'élément d'indice 8
IndexError: tuple index out of range
```

ATTENTION : Le premier élément d'un tuple est d'indice 0. On en déduit que si le tuple est de longueur n, alors les indices valides sont $0, 1, \ldots, n-1$.

Les tuples sont des objets **immuables** : on ne peut pas modifier les éléments d'un tuple, comme on peut le vérifier sur l'exemple ci-dessous.

```
>>> t = (1, 3.15, -2, "Python", True, (1, 2, 3)) # Définition d'un tuple
>>> t[1] = 2 # Renvoie une erreur car on ne peut pas modifier t
TypeError: 'tuple' object does not support item assignment
```

Question 11 :

1) Écrire une fonction `maxi(t:tuple) -> int` prenant en argument un tuple d'entiers `t` et qui renvoie la plus grande valeur de `t`. Par exemple, l'exécution de `maxi((1, 4, 2, 3))` renvoie `4`.
2) Écrire une fonction `maxi2(t:tuple) -> int` prenant en argument un tuple d'entiers `t` contenant au moins deux éléments et qui renvoie la seconde plus grande valeur du tuple `t`. Par exemple, l'exécution de la commande `maxi2((1, 4, 2, 3))` renvoie `3` tandis que celle de `maxi2((1, 4, 2, 4))` renvoie `4`.

Question 12 : Écrire une fonction `est_croissant(t:tuple) -> bool` prenant en argument un tuple d'entiers `t` et qui renvoie un booléen indiquant si les éléments de `t` sont rangés dans l'ordre croissant.

Question 13 : Écrire une fonction `ecart_minimal(t:tuple) -> tuple` prenant en argument un tuple d'entiers `t` et qui renvoie un tuple contenant deux éléments du tuple `t` dont l'écart est minimal. Par exemple, l'exécution de la commande `ecart_minimal((1, 4, -1, 2))` renvoie `(1, 2)`.

On peut effectuer des **extractions de tranche** d'un tuple `t` avec les différentes expressions

`t[a:]`, `t[:b]`, `t[a:b]`, `t[a::p]`, `t[:b:p]`, `t[::p]` ou `t[a:b:p]`.

- Le paramètre `a` permet d'indiquer un indice de début (inclus) pour la tranche. Si le paramètre `a` n'est pas précisé, la tranche commence au début du tuple `t`.
- Le paramètre `b` permet d'indiquer un indice de fin (exclu) pour la tranche. Si le paramètre `b` n'est pas précisé, la tranche se termine à la fin du tuple `t`.
- Le paramètre `p` permet d'indiquer un pas pour la tranche, i.e. de préciser l'écart pour deux indices consécutifs de la tranche. Si le paramètre `p` n'est pas précisé, il est égal à 1 par défaut.

Exemple 5 : Voici quelques exemples de tranches pour un tuple.

```
>>> t = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) # On définit un tuple t
>>> t[7:] # Tuple des éléments de t d'indice >= 7
(7, 8, 9)
>>> t[2:7] # Tuple des éléments de t d'indice >= 2 et < 7
(2, 3, 4, 5, 6)
>>> t[2:8:2] # Tuple des éléments de t d'indice >= 2 et < 8 avec un pas de 2
(2, 4, 6)
>>> t[::-1] # Tuple des éléments de t avec un pas de -1
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

III.B - Opérations arithmétiques sur les tuples

On peut fabriquer un nouveau tuple en **concaténant** deux tuples `t1` et `t2` avec l'expression `t1 + t2`.

```
>>> t1 = (1, 2, True) # On définit un tuple t1
>>> t2 = ("abc", 3.14) # On définit un tuple t2
>>> t1 + t2 # Concaténation des tuples t1 et t2
(1, 2, True, 'abc', 3.14)
```

Étant donnés un entier positif `n` et un tuple `t`, on peut effectuer la **multiplication** de `n` et de `t` en utilisant l'expression `n * t` : cette opération concatène `n` fois `t` avec lui-même.

```
>>> t = (1, "abc", 3.14) # Définition d'un tuple
>>> 3 * t
(1, 'abc', 3.14, 1, 'abc', 3.14, 1, 'abc', 3.14)
```

III.C - Conversion de type

En Python, il est possible de faire certaines conversions de type. Il faut employer ces conversions avec des précautions : il n'est pas toujours simple d'en prédire le comportement.

Exemple 6 : Voici quelques exemples de conversions de type.

```
>>> chaine = "abcde"
>>> tuple(chaine)
('a', 'b', 'c', 'd', 'e')
```

```
>>> t = (1, 2)
>>> str(t)
'(1, 2)'
```

Dans certains cas, la conversion n'est pas possible.

```
>>> a = 123
>>> tuple(a)
TypeError: 'int' object is not iterable
```

III.D - Algorithme de recherche dichotomique dans un tableau trié

Pour terminer, nous allons étudier des algorithmes pour rechercher un élément dans un tableau (représenté par un tuple dans notre cas).

Question 14 : Écrire une fonction `recherche(t:tuple, x:int) -> bool` prenant en argument un tuple d'entiers `t` et qui renvoie un booléen indiquant si `x` est un élément du tuple `t`.

Dans le cas où le tableau est trié, on peut utiliser l'algorithme de recherche dichotomique ci-dessous. Dans la dernière partie de ce TP, nous montrerons que cet algorithme est plus efficace que la recherche « naïve » utilisée dans la fonction précédente.

Algorithme de recherche dichotomique dans un tableau trié

Fonction `recherche_dicho(tab, x)`

Entrée : Un tableau non vide `tab` d'entiers trié dans l'ordre croissant, un entier `x`
Sortie : Un booléen indiquant si l'élément `x` est dans le tableau `tab`.

$a \leftarrow 0$
$b \leftarrow$ longueur du tableau `tab`

tant que b - a > 1 **faire**
  $c \leftarrow (a+b)$ // 2
  si `tab[c]` $\leqslant$ `x` **alors**
    $a \leftarrow c$
  sinon
    $b \leftarrow c$

retourner `tab[a] == x`

Question 15 : Écrire une fonction `recherche_dicho(t:tuple, x:int) -> bool` prenant en argument un tuple d'entiers rangés dans l'ordre croissant et implémentant l'algorithme décrit ci-dessus.

Partie IV Les listes

IV.A - Généralités

Les **listes** sont représentées en Python par le type `list`. Une liste est un ensemble d'objets de types quelconques (non nécessairement tous identiques) qui commence par un crochet ouvrant, termine par un crochet fermant et où tous les objets sont séparés par une virgule. Une liste peut contenir plusieurs fois le même objet et même d'autres listes.

Exemple 7 : L'instruction ci-dessous définie une liste `lst`.

```
>>> lst = [1, 2, True, 3.14, [-1, 2]]
```

Il est également possible de définir une liste par **compréhension** en s'appuyant sur la boucle `for`.

Exemple 8 : Voici quelques exemples de listes définies par compréhension.

```
>>> lst = [k**2 for k in range(6)]
>>> lst
[0, 1, 4, 9, 16, 25]
>>> lst = [k**2 for k in range(1,10) if k % 2 == 1]
>>> lst
[1, 9, 25, 49, 81]
>>> lst = [k for k in range(50) if k % 2 != 0 and k % 3 != 0]
>>> lst
[1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49]
```

Question 16 : En utilisant une définition par compréhension, générer les listes suivantes.

1) La liste `[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`.
2) La liste `[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]`.
3) La liste `[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`.
4) La liste `[-1.0, -0.8, -0.6, -0.4, -0.2, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0]`.
5) La liste `[0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2]`.
6) La liste dont les éléments sont les cubes des entiers compris entre -10 et 20 qui ne sont ni multiples de 2 ni multiples de 3, i.e. la liste `[-343, -125, -1, 1, 125, 343, 1331, 2197, 4913, 6859]`.

Question 17 : Écrire une fonction `subd(a:float, b:float, n:int) -> list` qui renvoie la liste des points de la subdivision régulière de l'intervalle $[a, b]$ en n parties. Par exemple, l'exécution de `subd(1.0, 2.0, 4)` renvoie la liste `[1.0, 1.25, 1.5, 1.75, 2.0]`.

Pour récupérer la **longueur** d'une liste on utilise la fonction `len`.

```
>>> lst = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst
>>> len(lst) # Longueur de la liste lst
7
```

Les listes sont des objets **itérables** : on peut les parcourir en utilisant une boucle `for` comme ci-dessous.

```
>>> lst = [1, 2.0, [1, 2]] # On définit une liste lst
>>> for element in lst:
...         print(element)
1
2.0
[1, 2]
```

On peut **accéder** aux éléments d'une liste en utilisant son **indice**.

```
>>> lst = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst
>>> lst[0] # Élément de lst d'indice 0
2
>>> lst[4] # Élément de lst d'indice 4
-1
>>> lst[10] # Renvoie une erreur car lst n'a pas d'élément d'indice 10
IndexError: list index out of range
```

ATTENTION : Le premier élément d'une liste est d'indice 0. On en déduit que si la liste est de longueur n, alors les indices valides sont $0, 1, \ldots, n-1$.

Les listes sont des objets **mutables** : on peut **modifier** les éléments d'une liste, comme on peut le vérifier sur l'exemple ci-dessous.

```
>>> lst = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst
>>> lst[3] = 0 # On modifie l'élément d'indice 3
>>> lst
[2, 1, 3, 0, -1, 5, 2]
```

Question 18 : Dans cet exercice, on présente et étudie l'algorithme du tri à bulles.

1) Écrire une fonction `parcours(lst:list) -> int` prenant en argument une liste `lst` = $[a_0, a_1, \ldots, a_{n-1}]$ composée d'entiers, qui modifie cette liste en permutant a_k et a_{k+1} si $a_{k+1} < a_k$ pour k variant de 0 à $n-2$ et qui renvoie le nombre de permutations effectuées. Par exemple, si `lst = [5, 4, 1, 6]`, alors l'exécution de `parcours(lst)` renvoie l'entier `2` et la liste `lst` devient `lst = [4, 1, 5, 6]`.
2) On souhaite utiliser la fonction `parcours` pour trier par ordre croissant une liste selon le principe suivant : pour une liste `lst` donnée en entrée, on répète l'application de la fonction `parcours` jusqu'à ce que `lst` devienne invariante par cette fonction, i.e. jusqu'à ce que la condition `lst == parcours(lst)` soit vérifiée. Écrire une fonction `tri_bulles(lst:list) -> None` prenant en argument une liste d'entiers et qui modifie cette liste en ordonnant ses éléments dans l'ordre croissant.
3) Expliquer pourquoi votre fonction `tri_bulles` se termine nécessairement et renvoie le résultat attendu.

Remarque 3 : On observe que les fonctions `parcours` et `tri_bulles` modifient la liste `lst` de manière pérenne alors qu'elle ne fait pas partie des valeurs de retour. Ce phénomène est possible car la liste `lst` est un objet mutable.

Plus généralement, en informatique, une fonction est dite à **effet de bord** (ou **effet secondaire**) si elle modifie l'état d'un objet autre que ceux de sa valeur de retour.

Pour cette raison, il est souvent nécessaire de travailler dans le corps de la fonction sur une copie de l'objet mutable concerné (voir sous-partie suivante) si on ne souhaite pas que les modifications effectuées par la fonction persistent à l'issue de l'appel.

Question 19 : Écrire une fonction `remplacer(lst:list, x, y) -> None` qui remplace les éléments de la liste `lst` égaux à `x` par l'élément `y`. Par exemple, si `lst = [1, 4, 1, 6]`, alors l'exécution de `remplacer(lst, 1, 0)` transforme la liste `lst` en `lst = [0, 4, 0, 6]`.

On peut **ajouter** un élément `x` à la fin d'une liste `lst` avec l'expression `lst.append(x)`.

```
>>> lst = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst
>>> lst.append(0) # Ajoute l'élément 0 à la fin de de lst
>>> lst # On vérifie que l'élément 0 a bien été ajouté à lst
[2, 1, 3, 1, -1, 5, 2, 0]
```

Question 20 : Soit $(F_n)_{n\in\mathbb{N}}$ la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et

$$\forall n \in \mathbb{N} \quad F_{n+2} = F_{n+1} + F_n.$$

Écrire une fonction `fibo(borne:int) -> list` qui renvoie la liste $[F_0, F_1 \ldots, F_k]$ contenant tous les éléments de la suite de Fibonacci inférieurs ou égaux à l'entier `borne`.

On peut **supprimer** le dernier élément d'une liste `lst` avec l'expression `lst.pop()`.

```
>>> lst = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst
>>> lst.pop() # Retire le dernier élément de lst et le renvoie
2
>>> lst # On vérifie que le dernier élément de lst a bien été retiré
[2, 1, 3, 1, -1, 5, 2]
```

On peut effectuer des **extractions de tranche** d'une liste `lst` avec les différentes expressions

`lst[a:]`, `lst[:b]`, `lst[a:b]`, `lst[a::p]`, `lst[:b:p]`, `lst[::p]` ou `lst[a:b:p]`.

- Le paramètre `a` permet d'indiquer un indice de début (inclus) pour la tranche. Si le paramètre `a` n'est pas précisé, la tranche commence au début de la liste `lst`.
- Le paramètre `b` permet d'indiquer un indice de fin (exclu) pour la tranche. Si le paramètre `b` n'est pas précisé, la tranche se termine à la fin de la liste `lst`.
- Le paramètre `p` permet d'indiquer un pas pour la tranche, i.e. de préciser l'écart pour deux indices consécutifs de la tranche. Si le paramètre `p` n'est pas précisé, il est égal à 1 par défaut.

Exemple 9 : Voici quelques exemples de tranches pour un tuple.

```
>>> lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # On définit une liste lst
>>> lst[7:] # Liste des éléments de lst d'indice >= 7
[7, 8, 9]
>>> lst[2:7] # Liste des éléments de lst d'indice >= 2 et < 7
[2, 3, 4, 5, 6]
>>> lst[2:8:2] # Liste des éléments de lst d'indice >= 2 et < 8 avec un pas de 2
[2, 4, 6]
>>> lst[::-1] # Liste des éléments de lst avec un pas de -1
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

IV.B - Opérations arithmétiques sur les listes

On peut fabriquer une nouvelle liste en **concaténant** deux listes `lst1` et `lst2` avec l'expression `lst1 + lst2`.

```
>>> lst1 = [2, 1, 3, 1, -1, 5, 2] # On définit une liste lst1
>>> lst2 = [1, 2, 3, 4] # On définit une liste lst2
>>> lst1 + lst2 # Concaténation des listes lst1 et lst2
[2, 1, 3, 1, -1, 5, 2, 1, 2, 3, 4]
```

Étant donnés un entier positif `n` et une liste `lst`, on peut effectuer la **multiplication** de `n` et de `lst` en utilisant l'expression `n * lst` : cette opération concatène `n` fois `lst` avec elle-même.

```
>>> lst = [1, "abc", 3.14] # Définition d'une liste
>>> 3 * lst
[1, 'abc', 3.14, 1, 'abc', 3.14, 1, 'abc', 3.14]
```

Remarque 4 : Cette nouvelle opération peut s'avérer très utile pour initialiser certaines listes facilement, comme sur l'exemple ci-dessous.

```
>>> lst = 10 * [0] # Définition d'une liste
>>> lst
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

IV.C - Conversion de type

En Python, il est possible de faire certaines conversions de type. Il faut employer ces conversions avec des précautions : il n'est pas toujours simple d'en prédire le comportement.

Exemple 10 : Voici quelques exemples de conversions de type.

```
>>> chaine = "les"
>>> list(chaine)
['l', 'e', 's']
```

```
>>> lst = ['l', 'e', 's']
>>> str(lst)
"['l', 'e', 's']"
```

```
>>> t = (1, 2, 3)
>>> list(t)
[1, 2, 3]
```

```
>>> lst = [1, 2, 3]
>>> tuple(lst)
(1, 2, 3)
```

Dans certains cas, la conversion n'est pas possible.

```
>>> a = 123
>>> list(a)
TypeError: 'int' object is not iterable
```

IV.D - Copier une liste

On peut générer une **copie** d'une liste `lst` avec l'expression `lst.copy()` ou `lst[:]`.

```
>>> lst1 = [1, 2, 3] # On définit une liste lst1
>>> lst2 = lst1.copy() # La liste lst2 est une copie de lst1
>>> lst2[2] = 4 # On modifie une valeur de lst2
>>> lst1, lst2 # La liste lst1 n'est pas modifiée tandis que lst2 est modifiée
([1, 2, 3], [1, 2, 4])
```

ATTENTION : Il y a plusieurs subtilités dans la manipulation des listes et de leurs copies.

- Si `lst1` est une liste, l'instruction `lst2 = lst1` ne génère pas une copie de `lst1` : les variables `lst1` et `lst2` pointent vers le même objet en mémoire.

```
>>> lst1 = [1, 2, 3] # On définit une liste lst1
>>> lst2 = lst1 # lst2 n'est pas une copie de lst1 : c'est le même objet
>>> lst2[2] = 4 # On modifie une valeur de lst2
>>> lst1, lst2 # Les deux listes ont été modifiées
([1, 2, 4], [1, 2, 4)]
```

- L'utilisation de `lst.copy()` a ses limites : l'ordinateur fabrique une nouvelle liste dont les éléments pointent vers le même objet en mémoire que dans la liste de départ. Ce fonctionnement peut poser des problèmes lorsqu'on manipule notamment des listes de listes. On dit que la copie est **superficielle**.

```
>>> lst1 = [[1,2], [1,2,3]] # On définit une liste lst1
>>> lst2 = lst1.copy() # La liste lst2 est une copie superficielle de lst1
>>> lst2.append(4)
>>> lst1, lst2 # La liste lst1 n'est pas modifiée tandis que lst2 est modifiée
([[1, 2], [1, 2, 3]], [[1, 2], [1, 2, 3], 4])

>>> lst1[0].append(5) # On ajoute un élément dans la première liste de lst1
>>> lst1, lst2 # Le premier élément de lst1 et lst2 ont été modifiés
([[1, 2, 5], [1, 2, 3]], [[1, 2, 5], [1, 2, 3], 4])
```

- Pour contourner la difficulté précédente, il est possible d'utiliser une librairie de Python pour effectuer une copie **profonde** de la liste, i.e. une copie de la liste contenant des copies de tous les objets de la liste de départ.

Partie V Les dictionnaires

V.A - Généralités

Les **dictionnaires** sont représentés en Python par le type `dict`. Un dictionnaire présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments par le biais d'un indice, on y accède par le biais d'une clé. Les clés du dictionnaire sont des objets qui doivent être des objets immuables : `str`, `int`, `tuple`, `float`,...

La création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1,\dots,c_n$ sont des **clés** (nécessairement deux à deux distinctes) et $v_1,\dots,v_n$ les **valeurs** qui leurs sont associées.

Exemple 11 : Les instructions ci-dessous définissent deux chaînes de caractères.

```
# Définition d'un dictionnaire avec quatre paires clé: valeur
dico = {"cle1": "valeur1", -8: [1,3], (3,5): "pcsi", 0.5: 12}

# Définition du dictionnaire vide
dico_vide = {}
```

Pour récupérer la **longueur** d'un dictionnaire on utilise la fonction `len`.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> len(dico) # Longueur du dictionnaire dico
3
```

Les dictionnaires sont des objets **itérables** : on peut les parcourir avec une boucle `for` comme ci-dessous.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> for element in dico:
...         print(element)
a
3
c
```

On peut **accéder** aux éléments d'un dictionnaire en utilisant ses clés.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> dico["c"] # Élément dont la clé est "c"
'd'
>>> dico[3] # Élément dont la clé est 3
2
>>> dico[0] # Renvoie une erreur car dico n'a pas de clé 0
KeyError: 0
```

Les dictionnaires sont des objets **mutables** : on peut **modifier** les éléments d'un dictionnaire, comme on peut le vérifier sur l'exemple ci-dessous.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> dico["a"] = 3.14 # On modifie la valeur de l'élément de clé "a"
>>> dico
{'a': 3.14, 3: 2, 'c': 'd'}
```

En utilisant la même syntaxe, on peut également **ajouter** un élément dans un dictionnaire.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> dico[9] = 3.14 # On ajoute la clé 9 avec la valeur 3.14
>>> dico
{'a': 1, 3: 2, 'c': 'd', 9: 3.14}
```

On peut **supprimer** un élément d'un dictionnaire `dico`.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> del dico[3] # On supprime l'élément de clé 3
>>> dico # On vérifie que l'élément de clé 3 dans dico a bien été supprimé
{"a": 1, "c": "d"}
>>> del dico[2] # Renvoie une erreur car dico n'a pas d'élément de clé 2
KeyError: 2
```

On peut **tester la présence** d'une clé dans un dictionnaire avec le mot-clé `in`.

```
>>> dico = {"a": 1, 3: 2, "c": "d"} # On définit un dictionnaire dico
>>> "a" in dico # On teste si "a" est une clé de dico
True
>>> 5 in dico # On teste si 5 est une clé de dico
False
```

Question 21 : Écrire une fonction `car_max(chaine:str) -> str` qui renvoie un caractère ayant le nombre maximal d'occurrences dans la chaîne de caractères `chaine`. Par exemple, l'exécution de `car_max("ananas")` renvoie le caractère `"a"`. On pourra utiliser un dictionnaire pour compter le nombre d'occurrences de chaque caractère.

Question 22 : Écrire une fonction `nb_elements(lst:list) -> int` qui prend en argument une liste d'entiers `lst` et renvoie le nombre d'éléments distincts dans la liste `lst`. Par exemple, si `lst=[1, 0, 0, 1, 2, 1, 1, 2]`, l'exécution de `nb_elements(lst)` renvoie `3`. On pourra utiliser un dictionnaire mémorisant les éléments déjà rencontrés.

V.B - Conversion de type

En Python, il est possible de faire certaines conversions de type. Il faut employer ces conversions avec des précautions : il n'est pas toujours simple d'en prédire le comportement.

Exemple 12 : Voici quelques exemples de conversions de type.

```
>>> dico = {"a": 1, 2: 3}
>>> str(dico)
"{'a': 1, 2: 3}"
```

```
>>> dico = {"a": 1, 2: 3}
>>> tuple(dico)
('a', 2)
```

```
>>> dico = {"a": 1, 2: 3}
>>> list(dico)
['a', 2]
```

Dans l'autre sens, la conversion n'est pas possible en général.

```
>>> t = (1, 2, 3)
>>> dict(t)
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

V.C - Copier un dictionnaire

On peut générer une **copie** d'un dictionnaire `dico` avec l'expression `dico.copy()`.

```
>>> dico1 = {"a": 1, "b": 2} # On définit un dictionnaire dico1
>>> dico2 = dico1.copy() # Le dictionnaire dico2 est une copie de dico1
>>> dico2["b"] = 3 # On modifie dico2
>>> dico1, dico2 # dico1 n'est pas modifié tandis que dico2 est modifié
({'a': 1, 'b': 2}, {'a': 1, 'b': 3})
```

ATTENTION : Les subtilités sur les copies de listes restent valables pour les copies de dictionnaires : les notions de copie **superficielle** et de copie **profonde** restent valables dans le contexte des dictionnaires.

Partie VI Une première approche de la complexité temporelle

Dans cette partie, on étudie une approche simplifiée de la notion de **complexité temporelle dans le pire des cas** d'une fonction (qui sera approfondie au second semestre). La notion de complexité temporelle permet d'évaluer l'efficacité d'un programme (i.e. de quantifier sa vitesse d'exécution) indépendamment de la machine sur lequel il est exécuté. Elle permet également de comparer l'efficacité de différents algorithmes résolvant le même problème.

On dit qu'une fonction prenant en entrée une structure de données `seq` est :

- de **complexité constante** s'il existe une constante $A \in \mathbb{R}$ ne dépendant pas de `seq` telle que la fonction accède à ou modifie moins de A fois un élément de `seq`,
- de **complexité logarithmique** s'il existe un couple $(A, B) \in \mathbb{R}^2$ ne dépendant pas de `seq` tel que l'algorithme de la fonction accède à ou modifie moins de $A\log(n) + B$ fois un élément de `seq` où n est la longueur de `seq`,
- de **complexité linéaire** s'il existe un couple $(A, B) \in \mathbb{R}^2$ ne dépendant pas de `seq` tel que l'algorithme de la fonction accède à ou modifie moins de $An + B$ fois un élément de `seq` où n est la longueur de `seq`,
- de **complexité quadratique** s'il existe un couple $(A, B) \in \mathbb{R}^2$ ne dépendant pas de `seq` tel que l'algorithme de la fonction accède à ou modifie moins de $An^2 + B$ fois un élément de `seq` où n est la longueur de `seq`.

Exemple 13 : On considère la fonction `prod(lst:list) -> int` définie ci-dessous qui prend en argument une liste d'entiers `lst` et qui renvoie le produit des éléments de `lst`.

```
def prod(lst:list) -> int:
        res = 1
        for x in lst:
                res *= x
        return(res)
```

En notant n le nombre d'éléments de la liste `lst`, on constate que la fonction `prod` accède n fois à des éléments de la liste `lst`, donc la fonction `prod` est de complexité linéaire.

Question 23 : Déterminer le type de complexité des fonctions suivantes définies dans les parties précédentes.

- Partie II : `compter`, `indice`.
- Partie III : `est_croissant`, `ecart_minimal`, `recherche`, `recherche_dicho`.
- Partie IV : `parcours`, `tri_bulles`.