Conteneurs de données

Introduction :

Le chapitre consacré aux structures de données aborde successivement différentes structures permettant d'organiser des données pour tenter de répondre au mieux aux problématiques du monde réel. Ce cours présente les principaux conteneurs de données dont nous étudierons les méthodes.

Dans une première partie nous aborderons les principaux conteneurs généralistes que sont les listes, les dictionnaires, les tuples et les sets. Dans une deuxième partie nous étudierons les structures de type piles et files. Enfin, nous verrons dans une troisième partie l'importance de choisir une structure de données adaptée à la situation à modéliser.

1 Conteneurs de données généralistes

Dans cette première partie nous présentons les principales structures de données telles qu'elles sont implémentées dans le langage Python.

Chaque type de conteneur de données dispose d'un certain nombre de méthodes pour en permettre l'utilisation. Sans prétendre à l'exhaustivité, du fait du nombre parfois important de ces méthodes, on présentera dans le cadre de ce cours les plus caractéristiques d'entre elles pour chaque type de conteneur.

Rappel

- La fonction native `dir()` fournit la liste des méthodes associées à un conteneur ou à un type de conteneur.
- La fonction native `help()` permet d'obtenir une aide intégrée sur les méthodes associées à ce conteneur ou type de conteneur.

Exemple

> Pour connaître la liste des méthodes associées au type de conteneur de données LIST, on passera la commande Python suivante : `dir(list)`. Pour en obtenir l'aide intégrée, on passera la commande `help(list)`.

a. Listes

Définition

> **Liste :**
>
> En Python, une liste est une structure de données dont les éléments individuels sont accessibles à partir de leur position indicielle. C'est une structure très couramment employée pour de nombreux usages.

Les listes sont **ordonnées** et **mutables**. Elles peuvent contenir toutes sortes d'objets, y compris d'autres listes imbriquées et d'autres structures de données. Elles peuvent en outre contenir des doublons.

La **création** d'une liste peut s'effectuer de plusieurs façons :

```
# liste vide
liste = list()
liste = [] # notation courte
```

```
# liste non vide
liste = ['Ada', 'Alan', 'Alice']
```

L'**ajout** d'élément(s) en fin de liste s'effectue avec la méthode `append()` ou la notation courte équivalente :

```
# ajout d'un élément
liste.append('Lisa')
liste += ['Paul'] # notation courte
print(liste)
# affiche ['Ada', 'Alan', 'Alice', 'Lisa', 'Paul']
```

→ L'ajout est automatiquement effectué en fin de liste.


Astuce

> Tant qu'elle n'a pas fait l'objet d'un **tri**, une liste reflète l'ordre d'insertion des éléments qui la composent.

Chaque élément est accessible par sa position indicielle.


Rappel

> La numérotation des indices commence à 0.

```
print(liste[2])
# affiche Alice
```

Les listes étant mutables, il est possible d'en **modifier** des éléments existants.

```
liste[2] = 'Guido'
```

Il est également possible d'**insérer** des éléments au sein des listes, ailleurs qu'à la fin de celle-ci, en employant la méthode `insert()`.

```
liste.insert(2, 'Elsa') #
```

→ Le nombre indique à quelle position indicielle de la liste effectuer l'insertion de l'élément.

```
print(liste)
# affiche ['Ada', 'Alan', 'Elsa', 'Guido', 'Lisa', 'Paul']
```

Une liste peut être **étendue** par l'ajout d'un contenu d'une autre liste ou d'un autre itérable, avec la méthode `extend()`.

```
rajouts = ['Juliette', 'Sylvain', 'Zoé']
liste.extend(rajouts)
```

La notation courte équivalente est la suivante :

```
liste += rajouts

print(liste)
# affiche ['Ada', 'Alan', 'Elsa', 'Guido', 'Lisa', 'Paul', 'Juliette', 'Sylvain', 'Zoé']
```

Attention

> Il ne faut pas confondre l'extension d'une liste par une autre avec l'ajout d'un élément en notation courte.
>
> - Ajout d'un élément à une liste : `liste += [element]`
> - Extension d'une liste par une autre liste : `liste += autre_liste`

L'erreur classique consiste à mélanger ces deux syntaxes, comme illustré ci-après.

Exemple

> Voici ce qui se produit quand on confond les syntaxes courtes de l'ajout et de l'extension de liste :
>
> ```
> liste = ['Lisa', 'Paul']
> liste += 'Sylvain' #
> ```
> Au lieu de `liste += ['Sylvain']`.
>
> Ceci ne produira pas le résultat escompté :
>
> ```
> print(liste)
> # affiche ['Lisa', 'Paul', 'S', 'y', 'l', 'v', 'a', 'i', 'n']
> ```

Les listes permettent d'en **extraire** des éléments. L'extrait du dernier élément est une opération courante, rendue possible par la méthode `pop()`.

```
# extraction d'un élément
liste = ['Ada', 'Guido', 'Alan', 'Alice']
extrait = liste.pop()

print(extrait)
# affiche Alice
```

```
print(liste)
# affiche ['Ada', 'Guido', 'Alan']
```

La méthode `pop()` extrait par défaut le dernier élément de la liste. Si on précise un numéro d'index, `pop()` extrait l'élément situé à la position correspondante.

```
extrait = liste.pop(0) # extrait le premier élément de la liste

print(extrait)
# affiche Ada

print(liste)
# affiche ['Guido', 'Alan']
```

À retenir

> L'appel de la méthode `pop()` retourne un élément et le supprime de la liste. Il génère une erreur de type `IndexError` si la liste est vide.

Les listes permettent également la **suppression** d'un élément en fonction de sa valeur, avec la méthode `remove()`.

```
liste.remove('Alan')
print(liste)
# affiche ['Guido']
```

La méthode `remove()` retire la première occurrence rencontrée.
Si la valeur n'est pas présente, une erreur de type `ValueError` est générée.

Les listes étant ordonnées, elles peuvent être triés en place, c'est-à-dire que la liste triée remplace la liste d'origine, avec la méthode `sort()`. L'ordre de la liste peut être facilement inversé avec la méthode `reverse()`.

b. Dictionnaires

Définition

Dictionnaire :

En Python, les dictionnaires sont des conteneurs de données permettant de stocker des paires de clés et de valeurs associées à ces clés. Tout comme les listes, ils sont des structures de données très couramment employées.

La notion d'ordre est inhérente aux listes que nous avons étudiées précédemment, car c'est la position de la donnée dans la liste qui permet de l'identifier et d'y accéder.
Avec les dictionnaires, c'est la **clé** qui ouvre l'accès à la donnée, l'ordre importe peu.


Astuce

Remarque :

Historiquement les dictionnaires Python n'étaient d'ailleurs pas ordonnés. Ils le sont devenus dans les versions récentes du langage (depuis Python 3.6).

Les dictionnaires sont **mutables**, comme les listes, et eux aussi peuvent contenir toutes sortes d'objets et des structures de données imbriquées, notamment d'autres dictionnaires et des listes.
Ils ne peuvent, en revanche, pas contenir de doublons de clés, mais une même **valeur** peut être affectée à des clés distinctes.

La **création** d'un dictionnaire peut s'effectuer de plusieurs façons :

```
# dictionnaire vide
meteo = dict()
meteo = {} # notation courte
```

```
# dictionnaire non vide
meteo = {'Toulouse': 15, 'Strasbourg': 16}
```

L'**ajout** d'un élément s'effectue en indiquant sa clé entre crochets et en lui affectant sa valeur.

```
# ajout d'un élément
meteo['Cherbourg'] = 13
```

L'**accès** à un élément du dictionnaire s'effectue principalement par sa clé, avec la même notation que pour l'ajout.

```
# accès à un élément
print(meteo['Toulouse'])
# affiche 15
```

Il en est de même pour la **modification** d'un élément.

```
# modification d'un élément
meteo['Toulouse'] = 17
```

Le **retrait** d'un élément s'effectue avec la méthode `pop()`. Contrairement à son emploi sur une liste, la méthode impose pour un dictionnaire de spécifier la clé concernée.

```
resultat = meteo.pop('Cherbourg')
print(resultat)
# affiche 13
```

Le retrait d'un élément inexistant se solde par une erreur de type `KeyError`.

Un dictionnaire peut être **complété et mis à jour** à partir d'un autre dictionnaire avec la méthode `update()`, selon les modalités suivantes :

- les clés absentes du dictionnaire sont ajoutées avec les valeurs correspondantes ;
- les valeurs des clés déjà présentes dans le dictionnaire d'origine sont remplacées par celles du dictionnaire ajouté.

```
meteo = {'Toulouse': 15, 'Strasbourg': 16, 'Cherbourg': 13}
ajouts = {'Paris': 13, 'Toulouse': 17}
meteo.update(ajouts)
```

```
print(meteo)
# affiche {'Toulouse': 17, 'Strasbourg': 16, 'Cherbourg': 13, 'Paris': 13}
```

Depuis la version 3.6 de Python, les dictionnaires sont nativement ordonnés : ils le sont sur la base de l'ordre d'insertion des éléments. Les dictionnaires des versions récentes du langage conservent ainsi cette trace de l'ordre d'insertion.
Les versions antérieures pouvaient déjà disposer de dictionnaires ordonnés (`OrderedDict`) via le module `collections` de la bibliothèque standard.

c. Tuples

Définition

> **Tuple :**
>
> Les tuples sont des séquences ordonnées et non mutables. Ils peuvent contenir toutes sortes d'éléments de différents types. Les tuples peuvent contenir des doublons.

La création d'un tuple peut s'effectuer par énumération ou à partir d'un itérable.

```
# création d'un tuple
eleves = ('Alice', 'Alan', 'Lisa')
print(eleves)
```

Astuce

> - Les parenthèses sont optionnelles pour la déclaration des tuples.
> - La déclaration par énumération d'un tuple comportant un seul élément nécessite l'ajout d'une virgule.

```
equipement = ('ordinateur',)
print(type(equipement))
# affiche
```

Les tuples étant **immutables**, il n'est pas possible d'ajouter, de modifier ou de supprimer des éléments. Ceux-ci sont accessibles en lecture par la notation indicielle, comme pour les listes.

```
print(eleves)
# affiche ('Alice', 'Alan', 'Lisa')

print(eleves[2])
# affiche Lisa
```

Les tuples possèdent également une méthode `count()` permettant de compter le nombre d'occurrences d'une valeur au sein d'un tuple.

```
print(eleves.count('Alice'))
# affiche 1
```

Rappel

> Le module collections de la bibliothèque standard de Python comporte une sous-classe `namedtuples` de tuples dont les valeurs sont nommées.

d. Sets

Définition

> **Set :**
>
> Les sets, ou ensembles, sont des conteneurs de données non ordonnés dont les éléments sont uniques. Les sets sont mutables et peuvent contenir des éléments hétérogènes, mais ne peuvent pas contenir d'éléments mutables tels que listes, dictionnaires ou sets.

La **création** d'un set s'effectue de la façon suivante :

```
villes = set() # set vide
villes = {'Toulouse', 'Strasbourg', 'Paris', 'Brest'}
```

L'**ajout** d'éléments à un set s'effectue avec la méthode `add()`.

```
villes.add('Cherbourg') # ajout d'un élément
```

```
print(villes)
# affiche {'Strasbourg', 'Toulouse', 'Paris', 'Brest', 'Cherbourg'}
```

Un set ne peut pas comporter de doublon, mais les tentatives d'ajout sont seulement infructueuses, elles ne produisent pas d'erreurs.

```
villes.add('Toulouse')
```

```
print(villes)
# affiche {'Strasbourg', 'Toulouse', 'Paris', 'Brest', 'Cherbourg'}
```

Un élément présent dans le set peut être **retiré** sur la base de sa valeur.

```
villes.remove('Strasbourg')
```

```
print(villes)
# affiche {'Toulouse', 'Paris', 'Brest', 'Cherbourg'}
```

Les sets disposent d'un jeu complet de méthodes ensemblistes spécifiques permettant d'obtenir très facilement :

- l'**union** entre deux sets avec la méthode `union()` ;
- l'**intersection** entre deux sets avec la méthode `intersection()` ;
- la **différence** entre deux sets avec la méthode `difference()` ;
- la **différence symétrique** entre deux sets avec la méthode `symmetric_difference()`.

Les sets permettent aussi de déterminer :

- si un set est un sous-ensemble d'un autre set avec la méthode `issubset()` ;
- si un set est un sur-ensemble d'un autre set avec la méthode `issuperset()`.

Nous avons présenté les conteneurs généralistes de type liste, dictionnaire, tuple et set. Nous allons maintenant nous intéresser à des conteneurs plus spécialisés, avec les piles et les files.

2 | Piles et files

a. Types de piles

Définition

> **Pile :**
>
> Une pile est une structure de donnée destinée à stocker des données et à les restituer en fonction de leur ordre d'ajout.

Il existe schématiquement deux types de piles :

- celles qui restituent en premier lieu les dernières données ajoutées ;
- celles qui restituent en premier lieu les données ajoutées en premier.

Ces deux types de piles sont appelées **LIFO** et **FIFO**, en référence aux sigles résumant leur fonctionnement.

1 Pile LIFO :

LIFO signifie *Last In, First Out*. Le sigle est francisé en DEPS pour « Dernier Entré, Premier Sorti ». Une pile de type LIFO est parfois désignée par sa dénomination anglaise *stack*.

Exemple

> L'exemple classique pour l'expliquer est celui de la pile d'assiettes. Lorsqu'on souhaite prendre une assiette, on prend celle située en haut de la pile. C'est la dernière à avoir été ajoutée qui est accessible. Ce sera donc la première à être retirée de la pile.

→ Un algorithme d'exploration de labyrinthe s'appuie sur une pile pour mémoriser (empiler) ses déplacements. S'il se heurte à une impasse, il peut ainsi revenir sur les derniers déplacements effectués (dépiler) depuis la dernière intersection rencontrée, et explorer ensuite les autres chemins non encore empruntés.

2 Pile FIFO :

FIFO signifie *First In, First Out*. Le sigle est francisé en PEPS pour « Premier Entré, Premier Sorti ». Les piles de type FIFO sont communément appelées « files » ou « files d'attente » (*queues* en anglais).

Exemple

> L'exemple classique pour la figurer est d'ailleurs celui d'une file d'attente au guichet, pour prendre un billet de train ou d'avion. Les personnes accèdent au guichet dans leur ordre d'arrivée.

→ Un serveur web ou un serveur d'impression traite les requêtes reçues de manière analogue en fonction de leur ordre d'arrivée dans une file d'attente.

Astuce

> Quand on parle uniquement de piles FIFO, on utilise de préférence le terme « file », mais quand on parle simultanément des deux types FIFO et LIFO, on parle de « piles ».

b. Méthodes des piles

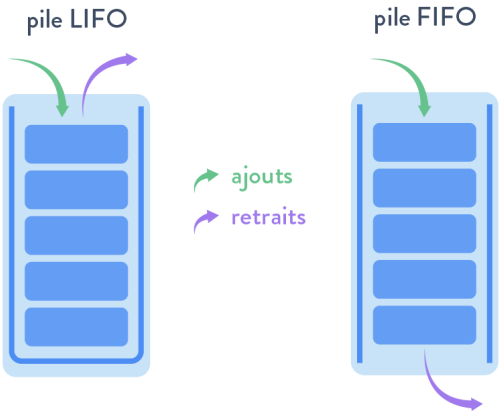
À retenir

> Les méthodes principales des piles sont l'**empilage** et le **dépilage**.
>
> - Empiler consiste à ajouter un élément à la pile.
> - Dépiler consiste à retirer un élément à la pile.

Astuce

> Le vocabulaire est parfois adapté au type de pile :
>
> - « empiler » et « dépiler » pour une pile LIFO ;
> - « enfiler » et « défiler » pour une pile FIFO.

Dans une pile LIFO, le point d'entrée de la pile est aussi le point de sortie. Dans une pile FIFO, le point d'entrée et le point de sortie sont aux extrémités opposées de la pile.



Les piles et les files sont des structures de données abstraites, qu'il est possible d'implémenter de différentes manières en fonction des langages. Avec Python nous implémenterons successivement des piles avec les listes natives, puis avec un type spécialisé disponible au sein de la bibliothèque standard.

C. Implémentation avec listes natives

Les listes natives de Python que nous avons étudiées en première partie nous permettent techniquement d'implémenter des piles LIFO et des piles FIFO.

1 Implémentation d'une pile LIFO

```
pile = ['a', 'b', 'c']
print(pile)
print('ajout nouvel élément')
pile.append('d') # empilage
print(pile)
```

```
while pile:
    print('dépilage de ', pile.pop())
    print('pile contient maintenant', pile)
```

Le code produit successivement les affichages suivants :

```
['a', 'b', 'c']
ajout nouvel élément
['a', 'b', 'c', 'd']
dépilage de d
pile contient maintenant ['a', 'b', 'c']
dépilage de c
pile contient maintenant ['a', 'b']
dépilage de b
pile contient maintenant ['a']
dépilage de a
pile contient maintenant []
```

2 Implémentation d'une pile FIFO

```
pile = ['c', 'b', 'a']
print(pile)
print('ajout nouvel élément')
pile.insert(0, 'd') # enfilage
print(pile)

while pile:
    print('défilage de ', pile.pop())
    print('pile contient maintenant', pile)
```

Le code produit successivement les affichages suivants :

```
['c', 'b', 'a']
['d', 'c', 'b', 'a']
dépilage de a
pile contient maintenant ['d', 'c', 'b']
dépilage de b
pile contient maintenant ['d', 'c']
dépilage de c
pile contient maintenant ['d']
```

```
dépilage de d
pile contient maintenant []
```

Toutefois les listes natives ne sont pas optimisées pour de tels traitements. Si l'empilage et le dépilage avec `pop()` et `append()` sont rapides car effectuées en fin de liste, les insertions en début de liste, nécessaires pour une file, sont lentes : elles obligent à décaler chaque fois en mémoire tous les éléments suivants composant la liste.

On aurait pu faire l'inverse en utilisant `append()` pour ajouter en fin de liste à la place d'`insert(0)` qui insérait en début de liste, mais le retrait d'une file devant s'effectuer à l'extrémité opposée, on aurait alors dû utiliser `pop(0)` pour les sorties de file, avec le même problème de décalage en mémoire des éléments suivants, et donc de performance.

d. Implémentation spécialisée

Il existe des conteneurs spécialisés pour **implémenter les files** avec le type `deque` du module `collections` de la bibliothèque standard. Ce type de conteneur a été conçu pour permettre l'ajout et le retrait d'éléments de manière optimisée à chaque extrémité du conteneur.

```
from collections import deque

file_attente = deque(['Alice', 'Charles'])
print(file_attente)
# affiche deque(['Alice', 'Charles'])
```

Le type `deque` de la bibliothèque `collections` implémente, en plus des méthodes `pop()` et `append()` des listes natives, les méthodes `appendleft()` et `popleft()`.
Ces dernières permettent toutes sortes d'usage de manière optimisée.

Pour l'implémentation d'une file d'attente, on peut choisir d'effectuer :

- les entrées (enfilage) avec `append()` ;
- les sorties (défilage) avec `popleft()`.

Si l'on souhaite ajouter Paul à la file d'attente :

```
file_attente.append('Paul')

print(file_attente)
# affiche deque(['Alice', 'Charles', 'Paul'])
```

La sortie de la file d'attente s'effectue de la manière suivante :

```
premier_sorti = file_attente.popleft()

print(premier_sorti)
# affiche Alice
```

On aurait pu symétriquement réaliser notre implémentation en remplaçant `append()` par `appendleft()` et `popleft()` par `pop()`. Dans les deux cas, l'entrée et la sortie s'effectuent aux extrémités opposées de la file d'attente.

Outre les piles, le module `collections` de la bibliothèque standard contient de nombreux autres types de conteneurs spécialisés pour étendre, modifier ou compléter les caractéristiques des conteneurs natifs.

3 Critères d'emploi des conteneurs de données

a. Modélisation

Définition

> **Modélisation :**
>
> La modélisation consiste à transposer un problème du monde réel en problématique traitable par un algorithme, lequel manipule des données.

Si certaines données sont liées entre elles, il est souhaitable que leur modélisation reflète ce lien pour éviter d'entrainer des incohérences. Illustrons-le avec l'exemple d'un groupe d'élèves dont on connaît les prénoms et les âges. On choisit dans un premier temps de stocker de manière parallèle ces informations dans deux listes distinctes.

```
prenoms = ['Ada', 'Alan', 'Alice', 'Bob']
ages = [17, 22, 21, 19]

for prenom, age in zip(prenoms, ages):
    print(prenom, 'a', age, 'ans')

# affiche
Ada a 17 ans
Alan a 22 ans
Alice a 21 ans
Bob a 19 ans
```

Les données concernant le nom et l'âge de l'élève sont stockées dans des conteneurs différents, mais nous pouvons **itérer** en même temps sur les deux listes.

Nous le faisons ici avec la fonction native `zip()` qui itère simultanément sur les éléments individuels de chaque itérable passé en argument.

 Astuce

> Si les listes sont de longueurs inégales, l'itération cesse une fois la fin de l'itérable le plus court atteinte.

La concordance entre les informations sur le prénom et l'âge d'un élève repose uniquement sur leur position dans chaque liste. La moindre erreur lors d'un ajout ou d'une suppression pourrait introduire un décalage, rendant les données incohérentes.

```
del prenoms[1] # suppression de l'entrée correspondant à Alan dans la liste des prénoms

for prenom, age in zip(prenoms, ages):
    print(prenom, 'a', age, 'ans')
```

Le code produit l'affichage suivant :

```
Ada a 17 ans
Alice a 22 ans
Bob a 21 ans
```

Les âges d'Alice et Bob ne sont plus bons car l'âge d'Alan n'a pas été supprimé alors que son prénom l'a été de la liste des élèves. Les deux listes étant indépendantes, on peut techniquement modifier l'une sans modifier l'autre, alors que les données qui les composent sont liées dans le monde réel.

Le recours à un dictionnaire s'avère donc plus adapté au cas présent :

```
eleves = {'Ada': 17, 'Alan': 22, 'Alice': 21, 'Bob': 19}

for prenom, age in eleves.items():
    print(prenom, 'a', age, 'ans')
```

Le code produit l'affichage suivant :

```
Ada a 17 ans
Alan a 22 ans
Alice a 21 ans
Bob a 19 ans
```

La suppression d'un élève (clé du dictionnaire) entraîne automatiquement la suppression de son âge (valeur associée à la clé).

```
del(eleves['Alan'])

print(eleves)
# affiche {'Ada': 17, 'Alice': 21, 'Bob': 19}
```

On pourrait éventuellement utiliser une liste de tuples à la place d'un dictionnaire :

```
eleves = [('Ada', 17),
          ('Alan', 22), ('Alice', 21), ('Bob', 19)]

for prenom, age in eleves:
    print(prenom, 'a', age, 'ans')
```

Le code produit l'affichage suivant :

```
Ada a 17 ans
Alan a 22 ans
```

```
Alice a 21 ans
Bob a 19 ans
```

Le recours à des tuples crée une association entre l'élève et son âge. La suppression d'un élément de la liste des élèves fait disparaître le tuple correspondant, contenant à la fois son prénom et son âge.

```
del(eleves[1])
```

```
print(eleves)
# affiche [('Ada', 17), ('Alice', 21), ('Bob', 19)]
```

Toutefois le caractère immutable des tuples nous empêche de modifier l'âge d'un élève.

```
print(eleves[0][1])
# affiche 17 (âge d'Alice)
```

```
eleves[0][1] = 18
# affiche TypeError: 'tuple' object does not support item assignment
```

Avec un dictionnaire, la modification d'âge ne poserait en revanche aucun problème puisque les données sont mutables.

b. Impact du conteneur sur la complexité algorithmique

La complexité d'un algorithme donné peut varier en fonction du type de conteneur de données utilisé. Considérons un algorithme très simple, composé d'une boucle parcourant séquentiellement tous les éléments pour vérifier s'ils sont ou non présents dans un conteneur.

```
for element in sequence:
    if element in conteneur:
        print(element, 'trouvé dans le conteneur')
```

Quelle est la complexité temporelle de cet algorithme ? La boucle `for` implique une complexité au moins linéaire, notée $O(n)$; mais la complexité globale de l'algorithme dépend aussi des traitements effectués à l'intérieur de la boucle.

Or, il est impossible de l'indiquer précisément sans connaître la nature du conteneur utilisé dans le test conditionnel. En effet, la complexité temporelle du test d'appartenance `element in conteneur` dépend de la nature de ce conteneur.

Ainsi, si le conteneur est une liste, le test d'appartenance va entraîner la comparaison de l'élément recherché avec chacun des éléments de la liste jusqu'à le trouver. Cette opération est de complexité linéaire soit $O(n)$.
En revanche ce même test d'appartenance est plus immédiat si le conteneur est un dictionnaire. La présence de la clé est évaluée en $O(1)$, la complexité du test d'appartenance est donc constante, quelle que soit la taille du dictionnaire.

Cela signifie que la complexité globale de l'algorithme sera linéaire ou quadratique, et que cette différence de performance, potentiellement très importante, repose uniquement sur le type de conteneur choisi, le code restant par ailleurs strictement identique.

> Conclusion :
>
> Nous avons présenté les principaux conteneurs de données que sont les listes, les dictionnaires, puis les tuples et les sets. Nous nous sommes ensuite intéressé·e·s aux conteneurs spécialisés que sont les piles et les files, dont nous avons montré différentes implémentations. Nous avons caractérisé ces conteneurs par leurs propriétés et les principales méthodes qu'ils proposent. Nous avons ensuite montré l'importance du choix d'un conteneur adapté à la problématique à traiter.