# MIA0301T : Algorithmique et programmation

## Les collections Python

5 octobre 2025

Université Jean Jaurès, Toulouse

# Sommaire

# Sommaire

# Types composés et séquences

# Types composés et séquences : définitions

- Un *type composé* est un type dont les valeurs sont formées de plusieurs valeurs. Le type *str* qui permet de représenter les chaînes et le type *list* permettant de représenter des listes sont donc des types composés.
- En *Python*, un *type séquence* est un type composé qui dispose des opérateurs *in*, *not in*, *+*, ***, des fonctions *len(...)*, *min(...)*, *max(...)*, de l'indexation et des tranches. Les types *str* et *list* sont donc des types séquences.
- Tous les types composés ne sont pas des types séquence : les dictionnaires (*dict*) et les ensembles (*set*) ne sont pas des types séquences.
- On peut assimiler un type séquence à un type composé où le placement des différents éléments a son importance.
- Un type séquence est un *type itérable*, puisqu'on peut parcourir ses éléments séquentiellement.

# Types composés et séquences : définitions

- Un *type composé* est un type dont les valeurs sont formées de plusieurs valeurs. Le type *str* qui permet de représenter les chaînes et le type *list* permettant de représenter des listes sont donc des types composés.
- En *Python*, un *type séquence* est un type composé qui dispose des opérateurs *in*, *not in*, *+*, ***, des fonctions *len(...)*, *min(...)*, *max(...)*, de l'indexation et des tranches. Les types *str* et *list* sont donc des types séquences.
- Tous les types composés ne sont pas des types séquence : les dictionnaires (*dict*) et les ensembles (*set*) ne sont pas des types séquences.
- On peut assimiler un type séquence à un type composé où le placement des différents éléments a son importance.
- Un type séquence est un *type itérable*, puisqu'on peut parcourir ses éléments séquentiellement.

# Types composés et séquences : définitions

- Un *type composé* est un type dont les valeurs sont formées de plusieurs valeurs. Le type *str* qui permet de représenter les chaînes et le type *list* permettant de représenter des listes sont donc des types composés.
- En *Python*, un *type séquence* est un type composé qui dispose des opérateurs *in*, *not in*, *+*, ***, des fonctions *len(...)*, *min(...)*, *max(...)*, de l'indexation et des tranches. Les types *str* et *list* sont donc des types séquences.
- Tous les types composés ne sont pas des types séquence : les dictionnaires (*dict*) et les ensembles (*set*) ne sont pas des types séquences.
- On peut assimiler un type séquence à un type composé où le placement des différents éléments a son importance.
- Un type séquence est un *type itérable*, puisqu'on peut parcourir ses éléments séquentiellement.

# Types composés et séquences : définitions

- Un *type composé* est un type dont les valeurs sont formées de plusieurs valeurs. Le type *str* qui permet de représenter les chaînes et le type *list* permettant de représenter des listes sont donc des types composés.
- En *Python*, un *type séquence* est un type composé qui dispose des opérateurs *in*, *not in*, *+*, ***, des fonctions *len(...)*, *min(...)*, *max(...)*, de l'indexation et des tranches. Les types *str* et *list* sont donc des types séquences.
- Tous les types composés ne sont pas des types séquence : les dictionnaires (*dict*) et les ensembles (*set*) ne sont pas des types séquences.
- On peut assimiler un type séquence à un type composé où le placement des différents éléments a son importance.
- Un type séquence est un *type itérable*, puisqu'on peut parcourir ses éléments séquentiellement.

# Types composés et séquences : définitions

- Un *type composé* est un type dont les valeurs sont formées de plusieurs valeurs. Le type *str* qui permet de représenter les chaînes et le type *list* permettant de représenter des listes sont donc des types composés.
- En *Python*, un *type séquence* est un type composé qui dispose des opérateurs *in*, *not in*, *+*, ***, des fonctions *len(...)*, *min(...)*, *max(...)*, de l'indexation et des tranches. Les types *str* et *list* sont donc des types séquences.
- Tous les types composés ne sont pas des types séquence : les dictionnaires (*dict*) et les ensembles (*set*) ne sont pas des types séquences.
- On peut assimiler un type séquence à un type composé où le placement des différents éléments a son importance.
- Un type séquence est un *type itérable*, puisqu'on peut parcourir ses éléments séquentiellement.

# Types composés et séquences : parcours

- Les séquences peuvent être parcourues de façon simple avec un *for* :

```
mess = "Bonjour à tous"
for car in mess:                # Possible parce que mess est une séquence
    print(car)
```

- Ou classiquement, en passant par les indices :

```
mess = "Bonjour à tous"
for i in range(0, len(mess)):
    print(mess[i])
```

# Types composés et séquences : parcours

- Les séquences peuvent être parcourues de façon simple avec un *for* :

```
mess = "Bonjour à tous"
for car in mess:                # Possible parce que mess est une séquence
    print(car)
```

- Ou classiquement, en passant par les indices :

```
mess = "Bonjour à tous"
for i in range(0, len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
for i in range(len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
i = 0
while i < len(mess):      # pas indiqué ici car le nombre d'itérations est connu d'avance
    print(mess[i])
    i+= 1
```

# Types composés et séquences : parcours

- Les séquences peuvent être parcourues de façon simple avec un *for* :

```
mess = "Bonjour à tous"
for car in mess:                # Possible parce que mess est une séquence
    print(car)
```

- Ou classiquement, en passant par les indices :

```
mess = "Bonjour à tous"
for i in range(0, len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
for i in range(len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
i = 0
while i < len(mess):      # pas indiqué ici car le nombre d'itérations est connu d'avance
    print(mess[i])
    i+= 1
```

# Types composés et séquences : parcours

- Les séquences peuvent être parcourues de façon simple avec un *for* :

```
mess = "Bonjour à tous"
for car in mess:                # Possible parce que mess est une séquence
    print(car)
```

- Ou classiquement, en passant par les indices :

```
mess = "Bonjour à tous"
for i in range(0, len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
for i in range(len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
i = 0
while i < len(mess):      # pas indiqué ici car le nombre d'itérations est connu d'avance
    print(mess[i])
    i+= 1
```

# Types composés et séquences : parcours

- Les séquences peuvent être parcourues de façon simple avec un *for* :

```
mess = "Bonjour à tous"
for car in mess:                # Possible parce que mess est une séquence
    print(car)
```

- Ou classiquement, en passant par les indices :

```
mess = "Bonjour à tous"
for i in range(0, len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
for i in range(len(mess)):
    print(mess[i])
```

```
mess = "Bonjour à tous"
i = 0
while i < len(mess):      # pas indiqué ici car le nombre d'itérations est connu d'avance
    print(mess[i])
    i+= 1
```

# Types composés et séquences : parcours

- Le parcours d'une séquence en *Python* avec un *for* doit privilégier la forme simple, sans indice.
- Si l'on ne souhaite pas parcourir toute la séquence, on utilise les tranches :

# Types composés et séquences : parcours

- Le parcours d'une séquence en *Python* avec un *for* doit privilégier la forme simple, sans indice.
- Si l'on ne souhaite pas parcourir toute la séquence, on utilise les tranches :

```
mess = "Bonjour à tous"
for c in mess[3::7]:          # c in "jour"
    ...

for c in mess[::3]:           # c in "Bnorâtu"
    ...
```

# Types composés et séquences : parcours

- Le parcours d'une séquence en *Python* avec un *for* doit privilégier la forme simple, sans indice.
- Si l'on ne souhaite pas parcourir toute la séquence, on utilise les tranches :

```
mess = "Bonjour à tous"
for c in mess[3::7]:          # c in "jour"
    ...

for c in mess[::3]:           # c in "Bnorâtu"
    ...
```

# Types composés et séquences : parcours

- Le parcours d'une séquence en *Python* avec un *for* doit privilégier la forme simple, sans indice.
- Si l'on ne souhaite pas parcourir toute la séquence, on utilise les tranches :

```
mess = "Bonjour à tous"
for c in mess[3:7]:          # c in "jour"
    ...

for c in mess[::2]:          # c in "Bnoràtu"
    ...
```

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

- La fonction *reversed()* permet de parcourir une séquence dans l'ordre inverse :

```
for c in reversed(mess):      # même effet que mess[::-1]
    print(c)
```

**Remarque**

reversed(mess) ne produit pas une chaîne mais un itérateur (une séquence parcourable avec for)

mess[::-1] produit une chaîne

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

- La fonction *reversed()* permet de parcourir une séquence dans l'ordre inverse :

```
for c in reversed(mess):      # même effet que mess[::-1]
    print(c)
```

**Remarque**

reversed(mess) ne produit pas une chaîne mais un itérateur (une séquence parcourable avec for)

mess[::-1] produit une chaîne

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

- La fonction *reversed()* permet de parcourir une séquence dans l'ordre inverse :

```
for c in reversed(mess):      # même effet que mess[::-1]
    print(c)
```

**Remarque**

*reversed(mess)* ne produit pas une chaîne mais un itérateur (une séquence parcourable avec *for*)

*mess[::-1]* produit une chaîne

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

- La fonction *reversed()* permet de parcourir une séquence dans l'ordre inverse :

```
for c in reversed(mess):      # même effet que mess[::-1]
    print(c)
```

**Remarque**

- *reversed(mess)* ne produit pas une chaîne mais un *itérateur* (une séquence parcourable avec *in*).
- *mess[ : :-1]* produit une chaîne.

# Types composés et séquences : parcours

- La fonction *enumerate()* permet d'obtenir à la fois l'indice d'un élément et son contenu :

```
for i, c in enumerate(mess):
    print(f"{c} est à l'indice {i}")
```

- La fonction *reversed()* permet de parcourir une séquence dans l'ordre inverse :

```
for c in reversed(mess):      # même effet que mess[::-1]
    print(c)
```

**Remarque**

- *reversed(mess)* ne produit pas une chaîne mais un *itérateur* (une séquence parcourable avec *in*).
- *mess[ : :-1]* produit une chaîne.

# Types composés et séquences : parcours

- En algorithmique, la boucle *Tant que* est privilégiée lorsque le parcours d'une séquence est lié à une condition et que l'on ne sait pas d'avance le nombre d'itérations à effectuer :

```
mess = "Bonjour à tous"
i = 0
while i < len(mess) and mess[i] != " ":
    print(mess[i])
    i += 1
```

- Notez que l'ordre des tests a une importance... Ici, il **faut** tester la valeur de *i* **avant** d'accéder à *mess[i]* (on rappelle que les opérateurs *and* et *or* fonctionnent en court-circuit)

# Types composés et séquences : parcours

- En algorithmique, la boucle *Tant que* est privilégiée lorsque le parcours d'une séquence est lié à une condition et que l'on ne sait pas d'avance le nombre d'itérations à effectuer :

```
mess = "Bonjour à tous"
i = 0
while i < len(mess) and mess[i] != " ":
    print(mess[i])
    i += 1
```

- Notez que l'ordre des tests a une importance... Ici, il **faut** tester la valeur de *i* **avant** d'accéder à *mess[i]* (on rappelle que les opérateurs *and* et *or* fonctionnent en court-circuit)

# Les tuples

# Les tuples

- Les tuples (qui n'existent pas en *Go*) sont des *types séquences immutables*. Ils permettent de regrouper plusieurs valeurs de type quelconque (simple ou composé).
- Un tuple est représenté par ses valeurs séparées par des virgules. Généralement, on l'entoure de parenthèses (mais elles ne sont obligatoires que dans certaines circonstances).
- Un tuple étant une séquence, ses différentes valeurs sont accessibles via leurs indices et l'on peut tester l'appartenance d'une valeur à un tuple avec l'opérateur *in*.

# Les tuples

- Les tuples (qui n'existent pas en *Go*) sont des *types séquences immutables*. Ils permettent de regrouper plusieurs valeurs de type quelconque (simple ou composé).
- Un tuple est représenté par ses valeurs séparées par des virgules. Généralement, on l'entoure de parenthèses (mais elles ne sont obligatoires que dans certaines circonstances).
- Un tuple étant une séquence, ses différentes valeurs sont accessibles via leurs indices et l'on peut tester l'appartenance d'une valeur à un tuple avec l'opérateur *in*.

# Les tuples

- Les tuples (qui n'existent pas en *Go*) sont des *types séquences immutables*. Ils permettent de regrouper plusieurs valeurs de type quelconque (simple ou composé).
- Un tuple est représenté par ses valeurs séparées par des virgules. Généralement, on l'entoure de parenthèses (mais elles ne sont obligatoires que dans certaines circonstances).
- Un tuple étant une séquence, ses différentes valeurs sont accessibles via leurs indices et l'on peut tester l'appartenance d'une valeur à un tuple avec l'opérateur *in*.

# Utilisation des tuples

- Généralement, on utilise un tuple pour rassembler des valeurs différentes afin de former une valeur composée : une adresse, par exemple, formée de la rue, du code postal et de la ville pourrait être représentée par :

```
adresse = ("21 rue de la pompe", "31000", "Toulouse")
```

- On utilise généralement le "dépaquetage" (unpacking) pour isoler les différentes valeurs d'un tuple :

```
rue, cp, ville = adresse        # idem (rue, cp, ville) = adresse
```

- Le plus souvent, les parenthèses d'un tuple ne sont pas nécessaires :

```
adresse           = "20, rue de la Pompe", "31000", "Toulouse"
nouvelle_adresse  = "22, rue de la Pompe", cp, ville
val1, val2        = 10, 100
```

# Utilisation des tuples

- Généralement, on utilise un tuple pour rassembler des valeurs différentes afin de former une valeur composée : une adresse, par exemple, formée de la rue, du code postal et de la ville pourrait être représentée par :

```
adresse = ("21 rue de la pompe", "31000", "Toulouse")
```

- On utilise généralement le "dépaquetage" (unpacking) pour isoler les différentes valeurs d'un tuple :

```
rue, cp, ville = adresse        # idem (rue, cp, ville) = adresse
```

- Le plus souvent, les parenthèses d'un tuple ne sont pas nécessaires :

```
adresse           = "20, rue de la Pompe", "31000", "Toulouse"
nouvelle_adresse  = "22, rue de la Pompe", cp, ville
val1, val2        = 10, 100
```

# Utilisation des tuples

- Généralement, on utilise un tuple pour rassembler des valeurs différentes afin de former une valeur composée : une adresse, par exemple, formée de la rue, du code postal et de la ville pourrait être représentée par :

```
adresse = ("21 rue de la pompe", "31000", "Toulouse")
```

- On utilise généralement le "dépaquetage" (unpacking) pour isoler les différentes valeurs d'un tuple :

```
rue, cp, ville = adresse        # idem (rue, cp, ville) = adresse
```

- Le plus souvent, les parenthèses d'un tuple ne sont pas nécessaires :

```
adresse           = "20, rue de la Pompe", "31000", "Toulouse"
nouvelle_adresse  = "22, rue de la Pompe", cp, ville
val1, val2        = 10, 100
```

# Utilisation des tuples

- Les tuples sont **immutables**, comme les chaînes de caractères : une fois initialisés, on ne peut plus modifier les éléments individuellement, même si ces éléments sont eux-mêmes modifiables.

```
moi = "Jacoboni", "Eric", 1.88
moi[2] = 1.89                          # TypeError: 'tuple' object does not support item assignment
nom, prenom, taille = moi
taille = 1.89
moi = nom, prenom, taille              # OK car réaffectation complète
print(moi)                             # Affiche ('Jacoboni', 'Eric', 1.89)
```

# Utilisation des tuples

- Les parenthèses sont obligatoires pour représenter un tuple vide et un tuple d'un seul élément doit finir par une virgule.

```
tuple_vide = ()
tuple_mono = "bonjour",          # Noter la virgule !!!
mauvais_tuple = ("salut")       # Ce n'est pas un tuple, car c'est égal à "salut"...
```

- En pratique, les tuples sont utilisés pour fabriquer d'autres types ou pour récupérer les résultats de fonctions qui renvoient plusieurs valeurs.

```
quotient, reste = divmod(5, 2)      # divmod renvoie un tuple...
```

## Remarque

Les fonctions Python ne renvoient en réalité qu'une seule valeur. Mais bien sûr, parfois cette valeur peut être un tuple. Lorsque l'on passe comme valeur de retour plusieurs valeurs, la fonction prédéfinie `divmod(a, b)` par exemple, est écrite un peu de la façon suivante :

# Utilisation des tuples

- Les parenthèses sont obligatoires pour représenter un tuple vide et un tuple d'un seul élément doit finir par une virgule.

```
tuple_vide = ()
tuple_mono = "bonjour",          # Noter la virgule !!!
mauvais_tuple = ("salut")       # Ce n'est pas un tuple, car c'est égal à "salut"...
```

- En pratique, les tuples sont utilisés pour fabriquer d'autres types ou pour récupérer les résultats de fonctions qui renvoient plusieurs valeurs.

```
quotient, reste = divmod(5, 2)      # divmod renvoie un tuple...
```

**Remarque**

Les fonctions Python ne renvoient en fait qu'une seule valeur. Mais lorsque l'on écrit cette valeur peut être un tuple. C'est ce qui se passe lorsque l'on renvoie plusieurs valeurs. Les fonctions prédéfinies divmod() par exemple, renvoie un tuple à deux

# Utilisation des tuples

- Les parenthèses sont obligatoires pour représenter un tuple vide et un tuple d'un seul élément doit finir par une virgule.

```
tuple_vide = ()
tuple_mono = "bonjour",          # Noter la virgule !!!
mauvais_tuple = ("salut")       # Ce n'est pas un tuple, car c'est égal à "salut"...
```

- En pratique, les tuples sont utilisés pour fabriquer d'autres types ou pour récupérer les résultats de fonctions qui renvoient plusieurs valeurs.

```
quotient, reste = divmod(5, 2)      # divmod renvoie un tuple...
```

**Remarque**

Les fonctions Python ne renvoient en fait qu'une seule valeur. Mais lorsque l'on écrit cette valeur peut être un tuple. C'est ce qui se passe lorsque l'on renvoie plusieurs valeurs. Les fonctions prédéfinies divmod() par exemple, renvoie un tuple à deux

# Utilisation des tuples

- Les parenthèses sont obligatoires pour représenter un tuple vide et un tuple d'un seul élément doit finir par une virgule.

```
tuple_vide = ()
tuple_mono = "bonjour",          # Noter la virgule !!!
mauvais_tuple = ("salut")       # Ce n'est pas un tuple, car c'est égal à "salut"...
```

- En pratique, les tuples sont utilisés pour fabriquer d'autres types ou pour récupérer les résultats de fonctions qui renvoient plusieurs valeurs.

```
quotient, reste = divmod(5, 2)       # divmod renvoie un tuple...
```

## Remarque

- Les fonctions *Python* ne renvoient en réalité qu'une seule valeur. Mais, dans certains cas, cette valeur peut être un tuple. . .(donc tout se passe comme si on renvoyait plusieurs valeurs). La fonction prédéfinie *divmod*, par exemple, exécute un code équivalent à :

```
def divmod(dividende, diviseur):
    return dividende // diviseur, dividende % diviseur
```

# Algorithmes classiques sur les tableaux/listes

# Algorithmes classiques sur les tableaux/listes

- Recherches (linéaires et dichotomique)
- Tris par sélection, par permutation (tri à bulle), par partition (rapide) et par fusion.
- Pour tous ces algorithmes, les listes sont censées contenir des éléments d'un type ordonné (doté d'une relation d'ordre).

**Remarque**

*Les termes « liste » et « tableau » seront employés ici de façon interchangeable, car il n'y a pas de différence entre eux en Python. En Go, on utilise le terme de « tranche ».*

*Certains langages font la différence entre listes et tableaux, mais ça ne change rien aux algorithmes présentés ici.*

# Algorithmes classiques sur les tableaux/listes

- Recherches (linéaires et dichotomique)
- Tris par sélection, par permutation (tri à bulle), par partition (rapide) et par fusion.
- Pour tous ces algorithmes, les listes sont censées contenir des éléments d'un type ordonné (doté d'une relation d'ordre).

**Remarque**

*Les termes « liste » et « tableau » seront employés ici de façon interchangeable, car il n'y a pas de différence entre eux en Python. En Go, on utilise le terme de « tranche ».*

*Certains langages font la différence entre listes et tableaux, mais ça ne change rien aux algorithmes présentés ici.*

# Algorithmes classiques sur les tableaux/listes

- Recherches (linéaires et dichotomique)
- Tris par sélection, par permutation (tri à bulle), par partition (rapide) et par fusion.
- Pour tous ces algorithmes, les listes sont censées contenir des éléments d'un type ordonné (doté d'une relation d'ordre).

**Remarque**

*Les termes « liste » et « tableau » seront employés ici de façon interchangeable, car il n'y a pas de différence entre eux en Python. En Go, on utilise le terme de « tranche ».*

*Certains langages font la différence entre listes et tableaux, mais ça ne change rien aux algorithmes présentés ici.*

# Algorithmes classiques sur les tableaux/listes

- Recherches (linéaires et dichotomique)
- Tris par sélection, par permutation (tri à bulle), par partition (rapide) et par fusion.
- Pour tous ces algorithmes, les listes sont censées contenir des éléments d'un type ordonné (doté d'une relation d'ordre).

**Remarque**

*Les termes « liste » et « tableau » seront employés ici de façon interchangeable, car il n'y a pas de différence entre eux en Python. En Go, on utilise le terme de « tranche ».*

*Certains langages font la différence entre listes et tableaux, mais ça ne change rien aux algorithmes présentés ici.*

# Algorithmes classiques sur les tableaux/listes

- Recherches (linéaires et dichotomique)
- Tris par sélection, par permutation (tri à bulle), par partition (rapide) et par fusion.
- Pour tous ces algorithmes, les listes sont censées contenir des éléments d'un type ordonné (doté d'une relation d'ordre).

**Remarque**

*Les termes « liste » et « tableau » seront employés ici de façon interchangeable, car il n'y a pas de différence entre eux en Python. En Go, on utilise le terme de « tranche ».*

*Certains langages font la différence entre listes et tableaux, mais ça ne change rien aux algorithmes présentés ici.*

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches

- On peut distinguer deux types de recherches :
  - Les recherches linéaires
  - Les recherches non linéaires dans une liste **déjà triée** (recherche dichotomique)
- Parmi les recherches linéaires, on peut distinguer :
  - Les recherches dans une liste non nécessairement triée
  - Les recherches dans une liste triée
- Dans tous les cas, que doit renvoyer une fonction de recherche ?

# Recherches linéaires

- Une recherche est linéaire si elle consiste à parcourir la liste depuis le début jusqu'à trouver l'élément recherché ou atteindre la fin sans l'avoir trouvé.
- On comprend que ce type de recherche dépend de la longueur de la liste : au pire, il faudra $n$ étapes pour avoir la réponse ($n$ étant le nombre d'éléments de la liste).
- La recherche dans une liste non triée est obligée de parcourir tout la liste avant de pouvoir décider que l'élément ne s'y trouve pas.
- La recherche dans une liste triée peut constater que l'élément n'appartient pas à la liste dès qu'elle rencontre un élément supérieur à l'élément recherché.

# Recherches linéaires

- Une recherche est linéaire si elle consiste à parcourir la liste depuis le début jusqu'à trouver l'élément recherché ou atteindre la fin sans l'avoir trouvé.
- On comprend que ce type de recherche dépend de la longueur de la liste : au pire, il faudra $n$ étapes pour avoir la réponse ($n$ étant le nombre d'éléments de la liste).
- La recherche dans une liste non triée est obligée de parcourir tout la liste avant de pouvoir décider que l'élément ne s'y trouve pas.
- La recherche dans une liste triée peut constater que l'élément n'appartient pas à la liste dès qu'elle rencontre un élément supérieur à l'élément recherché.

# Recherches linéaires

- Une recherche est linéaire si elle consiste à parcourir la liste depuis le début jusqu'à trouver l'élément recherché ou atteindre la fin sans l'avoir trouvé.
- On comprend que ce type de recherche dépend de la longueur de la liste : au pire, il faudra $n$ étapes pour avoir la réponse ($n$ étant le nombre d'éléments de la liste).
- La recherche dans une liste non triée est obligée de parcourir tout la liste avant de pouvoir décider que l'élément ne s'y trouve pas.
- La recherche dans une liste triée peut constater que l'élément n'appartient pas à la liste dès qu'elle rencontre un élément supérieur à l'élément recherché.

# Recherches linéaires

- Une recherche est linéaire si elle consiste à parcourir la liste depuis le début jusqu'à trouver l'élément recherché ou atteindre la fin sans l'avoir trouvé.
- On comprend que ce type de recherche dépend de la longueur de la liste : au pire, il faudra $n$ étapes pour avoir la réponse ($n$ étant le nombre d'éléments de la liste).
- La recherche dans une liste non triée est obligée de parcourir tout la liste avant de pouvoir décider que l'élément ne s'y trouve pas.
- La recherche dans une liste triée peut constater que l'élément n'appartient pas à la liste dès qu'elle rencontre un élément supérieur à l'élément recherché.

# Recherche linéaire dans une liste quelconque

- Que la liste soit triée ou non, une recherche linéaire consiste à balayer de la gauche vers la droite jusqu'à trouver l'élément ou atteindre la fin de la liste.

```
def recherche_lineaire(elt, liste):
  # TODO
```

# Recherche linéaire dans une liste triée

- Si la liste est triée par ordre croissant/décroissant, on peut optimiser la recherche car si l'on trouve un élément plus grand/petit que l'élément recherché, c'est qu'il ne s'y trouve pas : on peut donc s'arrêter avant d'atteindre la fin de la liste.

```
def recherche_lineaire_triee(elt, liste):
  # TODO
```

# Recherches non linéaires

- Ce type de recherche consiste à réduire l'espace de recherche à chaque itération. Pour que cela fonctionne, il faut que les éléments soient rangés dans un certain ordre.
- Pour une liste triée, la méthode la plus connue est la *recherche dichotomique*, qui consiste à diviser par deux l'espace de recherche. La recherche s'effectue alors au pire en $\log n$ étapes, ce qui est inférieur au $n$ de la recherche séquentielle...
- Il existe aussi des méthodes de recherche utilisant des arbres binaires (voir l'UE MIC0301T).

# Recherches non linéaires

- Ce type de recherche consiste à réduire l'espace de recherche à chaque itération. Pour que cela fonctionne, il faut que les éléments soient rangés dans un certain ordre.
- Pour une liste triée, la méthode la plus connue est la *recherche dichotomique*, qui consiste à diviser par deux l'espace de recherche. La recherche s'effectue alors au pire en $\log n$ étapes, ce qui est inférieur au $n$ de la recherche séquentielle...
- Il existe aussi des méthodes de recherche utilisant des arbres binaires (voir l'UE MIC0301T).

# Recherches non linéaires

- Ce type de recherche consiste à réduire l'espace de recherche à chaque itération. Pour que cela fonctionne, il faut que les éléments soient rangés dans un certain ordre.
- Pour une liste triée, la méthode la plus connue est la *recherche dichotomique*, qui consiste à diviser par deux l'espace de recherche. La recherche s'effectue alors au pire en $\log n$ étapes, ce qui est inférieur au $n$ de la recherche séquentielle...
- Il existe aussi des méthodes de recherche utilisant des arbres binaires (voir l'UE MIC0301T).

# Recherche dichotomique : principe

- On prend l'élément du milieu de la liste triée et on le compare à la valeur recherchée :
  - S'il est égal, c'est terminé et on renvoie sa position
  - S'il est plus petit, on réitère l'opération sur la partie droite de la liste
  - S'il est plus grand, on réitère l'opération sur la partie gauche.
- C'est donc un algorithme récursif que nous présenterons plus tard, mais que nous pouvons aussi écrire de façon itérative.

# Recherche dichotomique : principe

- On prend l'élément du milieu de la liste triée et on le compare à la valeur recherchée :
  - S'il est égal, c'est terminé et on renvoie sa position
  - S'il est plus petit, on réitère l'opération sur la partie droite de la liste
  - S'il est plus grand, on réitère l'opération sur la partie gauche.
- C'est donc un algorithme récursif que nous présenterons plus tard, mais que nous pouvons aussi écrire de façon itérative.

# Recherche dichotomique : principe

- On prend l'élément du milieu de la liste triée et on le compare à la valeur recherchée :
  - S'il est égal, c'est terminé et on renvoie sa position
  - S'il est plus petit, on réitère l'opération sur la partie droite de la liste
  - S'il est plus grand, on réitère l'opération sur la partie gauche.
- C'est donc un algorithme récursif que nous présenterons plus tard, mais que nous pouvons aussi écrire de façon itérative.

# Recherche dichotomique : principe

- On prend l'élément du milieu de la liste triée et on le compare à la valeur recherchée :
  - S'il est égal, c'est terminé et on renvoie sa position
  - S'il est plus petit, on réitère l'opération sur la partie droite de la liste
  - S'il est plus grand, on réitère l'opération sur la partie gauche.
- C'est donc un algorithme récursif que nous présenterons plus tard, mais que nous pouvons aussi écrire de façon itérative.

# Recherche dichotomique : principe

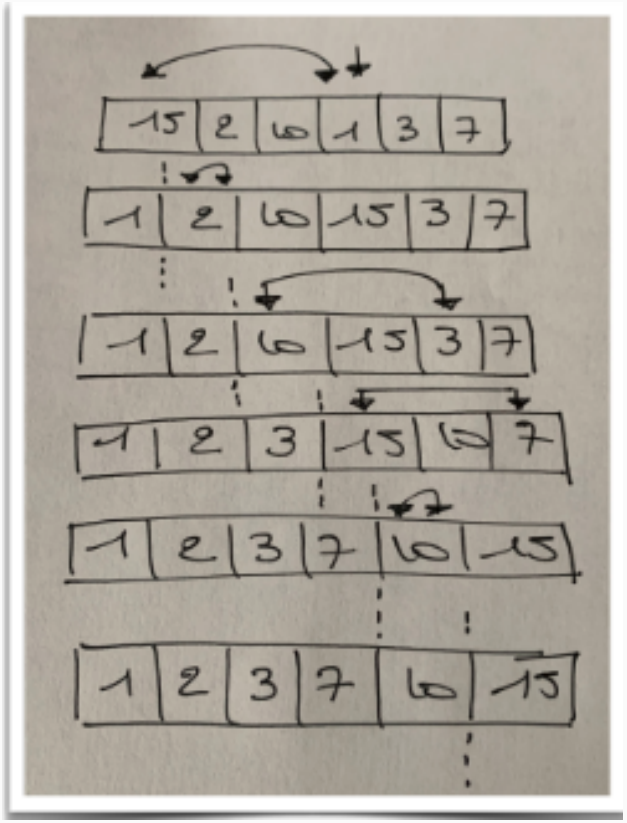
- On prend l'élément du milieu de la liste triée et on le compare à la valeur recherchée :
  - S'il est égal, c'est terminé et on renvoie sa position
  - S'il est plus petit, on réitère l'opération sur la partie droite de la liste
  - S'il est plus grand, on réitère l'opération sur la partie gauche.
- C'est donc un algorithme récursif que nous présenterons plus tard, mais que nous pouvons aussi écrire de façon itérative.

# Recherche dichotomique : implémentation itérative

```
def recherche_dicho(elt, liste):
  # TODO
```

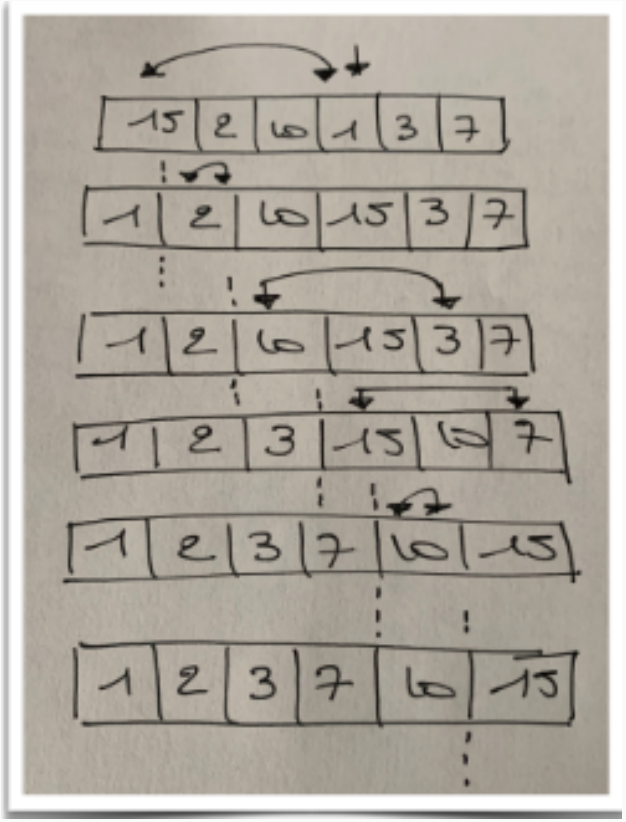
# Tri par sélection : principe

- On recherche le plus petit élément du tableau et on l'échange avec le premier élément.
- Puis on recherche le second plus petit et on l'échange avec le second élément
- etc. jusqu'à ce que le tableau soit complètement trié.



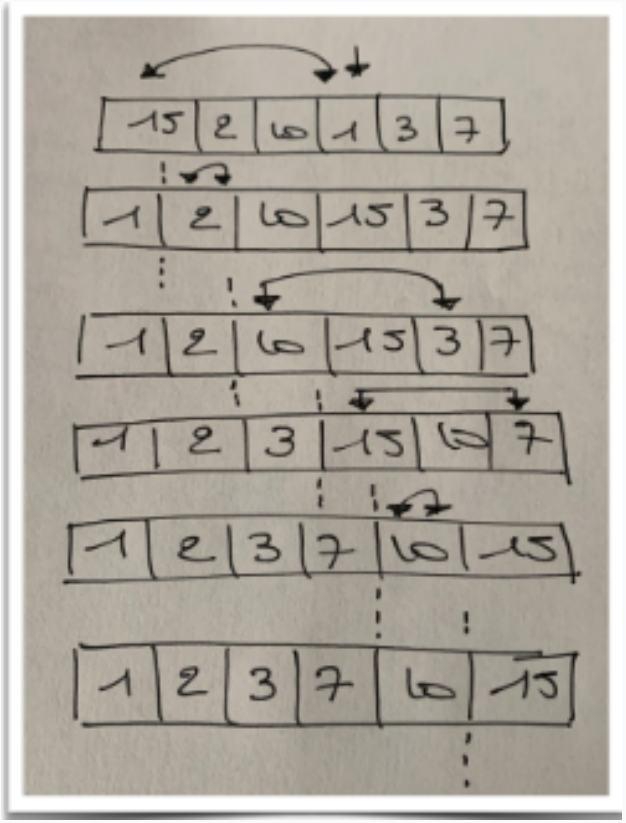
# Tri par sélection : principe

- On recherche le plus petit élément du tableau et on l'échange avec le premier élément.
- Puis on recherche le second plus petit et on l'échange avec le second élément
- etc. jusqu'à ce que le tableau soit complètement trié.



# Tri par sélection : principe

- On recherche le plus petit élément du tableau et on l'échange avec le premier élément.
- Puis on recherche le second plus petit et on l'échange avec le second élément
- etc. jusqu'à ce que le tableau soit complètement trié.



# Tri par sélection : principe

- À chaque étape, on recherche le plus petit élément de l'indice $i + 1$ à la fin du tableau et on échange cet élément avec l'élément à l'indice $i$. Initialement, $i$ vaut 0
- L'algorithme s'arrête lorsque $i$ est l'indice de l'avant-dernière case.

```
def tri_selection(liste):
    # TODO
```

# Tri par sélection : principe

- À chaque étape, on recherche le plus petit élément de l'indice $i + 1$ à la fin du tableau et on échange cet élément avec l'élément à l'indice $i$. Initialement, $i$ vaut 0
- L'algorithme s'arrête lorsque $i$ est l'indice de l'avant-dernière case.

```
def tri_selection(liste):
    # TODO
```

# Tri par sélection : principe

- À chaque étape, on recherche le plus petit élément de l'indice $i + 1$ à la fin du tableau et on échange cet élément avec l'élément à l'indice $i$. Initialement, $i$ vaut 0
- L'algorithme s'arrête lorsque $i$ est l'indice de l'avant-dernière case.

```
def tri_selection(liste):
    # TODO
```

# Tri par sélection : principe

- À chaque étape, on recherche le plus petit élément de l'indice $i + 1$ à la fin du tableau et on échange cet élément avec l'élément à l'indice $i$. Initialement, $i$ vaut 0
- L'algorithme s'arrête lorsque $i$ est l'indice de l'avant-dernière case.

```
def tri_selection(liste):
  # TODO
```

# Tri par permutation (tri à bulle) : principe

- On compare deux à deux les éléments du tableau en les échangeant si nécessaire et on recommence tant qu'il y a eu permutation.
- À la fin du premier parcours, le plus grand élément est donc placé à la fin du tableau, à la fin du second, le second plus grand est avant-dernier, etc.
- On peut donc optimiser cet algorithme en réduisant la longueur du tableau à chaque parcours.

```
def tri_bulle(liste):
    # TODO
```

# Tri par permutation (tri à bulle) : principe

- On compare deux à deux les éléments du tableau en les échangeant si nécessaire et on recommence tant qu'il y a eu permutation.
- À la fin du premier parcours, le plus grand élément est donc placé à la fin du tableau, à la fin du second, le second plus grand est avant-dernier, etc.
- On peut donc optimiser cet algorithme en réduisant la longueur du tableau à chaque parcours.

```
def tri_bulle(liste):
    # TODO
```

# Tri par permutation (tri à bulle) : principe

- On compare deux à deux les éléments du tableau en les échangeant si nécessaire et on recommence tant qu'il y a eu permutation.
- À la fin du premier parcours, le plus grand élément est donc placé à la fin du tableau, à la fin du second, le second plus grand est avant-dernier, etc.
- On peut donc optimiser cet algorithme en réduisant la longueur du tableau à chaque parcours.

```
def tri_bulle(liste):
    # TODO
```

# Tri par permutation (tri à bulle) : principe

- On compare deux à deux les éléments du tableau en les échangeant si nécessaire et on recommence tant qu'il y a eu permutation.
- À la fin du premier parcours, le plus grand élément est donc placé à la fin du tableau, à la fin du second, le second plus grand est avant-dernier, etc.
- On peut donc optimiser cet algorithme en réduisant la longueur du tableau à chaque parcours.

```
def tri_bulle(liste):
    # TODO
```

# Tri par permutation (tri à bulle) : principe

- On compare deux à deux les éléments du tableau en les échangeant si nécessaire et on recommence tant qu'il y a eu permutation.
- À la fin du premier parcours, le plus grand élément est donc placé à la fin du tableau, à la fin du second, le second plus grand est avant-dernier, etc.
- On peut donc optimiser cet algorithme en réduisant la longueur du tableau à chaque parcours.

```
def tri_bulle(liste):
  # TODO
```

# Tri par permutation optimisé : principe

- Comme on l'a vu précédemment, au premier parcours, l'élément le plus grand est placé à la fin du tableau
- Au second parcours, le second plus grand est à l'avant-dernière position, etc.
- On peut donc réduire d'une case le parcours à chaque itération.

```
def tri_bulle_optimise(liste):
    # TODO
```

# Tri par permutation optimisé : principe

- Comme on l'a vu précédemment, au premier parcours, l'élément le plus grand est placé à la fin du tableau
- Au second parcours, le second plus grand est à l'avant-dernière position, etc.
- On peut donc réduire d'une case le parcours à chaque itération.

```
def tri_bulle_optimise(liste):
    # TODO
```

# Tri par permutation optimisé : principe

- Comme on l'a vu précédemment, au premier parcours, l'élément le plus grand est placé à la fin du tableau
- Au second parcours, le second plus grand est à l'avant-dernière position, etc.
- On peut donc réduire d'une case le parcours à chaque itération.

```
def tri_bulle_optimise(liste):
    # TODO
```

# Tri par permutation optimisé : principe

- Comme on l'a vu précédemment, au premier parcours, l'élément le plus grand est placé à la fin du tableau
- Au second parcours, le second plus grand est à l'avant-dernière position, etc.
- On peut donc réduire d'une case le parcours à chaque itération.

```
def tri_bulle_optimise(liste):
    # TODO
```

# Tri par permutation optimisé : principe

- Comme on l'a vu précédemment, au premier parcours, l'élément le plus grand est placé à la fin du tableau
- Au second parcours, le second plus grand est à l'avant-dernière position, etc.
- On peut donc réduire d'une case le parcours à chaque itération.

```
def tri_bulle_optimise(liste):
  # TODO
```

# Tri par fusion : principe

- On découpe le tableau en deux parties égales, on trie ces deux parties et on les fusionne.
- C'est donc typiquement un tri récursif que nous étudierons plus tard.
- Un algorithme intéressant est celui de la fusion de deux tableaux triés : nous allons le présenter ici sous sa version itérative et nous l'étudierons plus loin sous sa forme récursive.

**Remarque**

*Ce tri est particulièrement adapté pour les tris externes, dans lesquels il faut trier un volume de données trop important pour tenir en mémoire. On est donc obligé de découper ce volume en plusieurs tronçons, les trier et les fusionner.*

# Tri par fusion : principe

- On découpe le tableau en deux parties égales, on trie ces deux parties et on les fusionne.
- C'est donc typiquement un tri récursif que nous étudierons plus tard.
- Un algorithme intéressant est celui de la fusion de deux tableaux triés : nous allons le présenter ici sous sa version itérative et nous l'étudierons plus loin sous sa forme récursive.

**Remarque**

*Ce tri est particulièrement adapté pour les tris externes, dans lesquels il faut trier un volume de données trop important pour tenir en mémoire. On est donc obligé de découper ce volume en plusieurs tronçons, les trier et les fusionner.*

# Tri par fusion : principe

- On découpe le tableau en deux parties égales, on trie ces deux parties et on les fusionne.
- C'est donc typiquement un tri récursif que nous étudierons plus tard.
- Un algorithme intéressant est celui de la fusion de deux tableaux triés : nous allons le présenter ici sous sa version itérative et nous l'étudierons plus loin sous sa forme récursive.

**Remarque**

*Ce tri est particulièrement adapté pour les tris externes, dans lesquels il faut trier un volume de données trop important pour tenir en mémoire. On est donc obligé de découper ce volume en plusieurs tronçons, les trier et les fusionner.*

# Tri par fusion : principe

- On découpe le tableau en deux parties égales, on trie ces deux parties et on les fusionne.
- C'est donc typiquement un tri récursif que nous étudierons plus tard.
- Un algorithme intéressant est celui de la fusion de deux tableaux triés : nous allons le présenter ici sous sa version itérative et nous l'étudierons plus loin sous sa forme récursive.

**Remarque**

*Ce tri est particulièrement adapté pour les tris externes, dans lesquels il faut trier un volume de données trop important pour tenir en mémoire. On est donc obligé de découper ce volume en plusieurs tronçons, les trier et les fusionner.*

# Tri par fusion : principe

- On découpe le tableau en deux parties égales, on trie ces deux parties et on les fusionne.
- C'est donc typiquement un tri récursif que nous étudierons plus tard.
- Un algorithme intéressant est celui de la fusion de deux tableaux triés : nous allons le présenter ici sous sa version itérative et nous l'étudierons plus loin sous sa forme récursive.

**Remarque**

*Ce tri est particulièrement adapté pour les tris externes, dans lesquels il faut trier un volume de données trop important pour tenir en mémoire. On est donc obligé de découper ce volume en plusieurs tronçons, les trier et les fusionner.*

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Fusion de deux tableaux triés : principe

1. On parcourt les deux tableaux élément par élément en recopiant à chaque fois le plus petit élément dans le tableau résultat.
2. Lorsque l'un des deux tableaux a été totalement parcouru, on ajoute ce qui reste du second à la fin du tableau résultat.
3. Il faut donc un indice par tableau parcouru. Le tableau résultat partira d'un tableau vide qui sera mis à jour par des appels à *append* et *+=*.
4. Notez aussi que les tranches permettent de simplifier la recopie finale de l'étape 2.

```
def fusion(tab1, tab2):
    """Renvoie le résultat de la fusion de tab1 et tab2 (supposés triés dans le même ordre)"""
    # TODO
```

# Tri rapide de Hoare : principe

- On prend une valeur du tableau (le "pivot"). Il peut s'agir de la première valeur, de la valeur située au milieu, ...
- On crée deux tableaux : les éléments plus petits ou égaux au pivot, les éléments plus grands que le pivot. C'est donc un tri par partition.
- On trie ces deux tableaux.
- On concatène le tableau de plus petits, le pivot et le tableau des plus grands.
- C'est donc là encore un algorithme récursif que nous étudierons plus tard (on peut l'écrire de façon impérative mais c'est un exercice peu naturel).

# Tri rapide de Hoare : principe

- On prend une valeur du tableau (le "pivot"). Il peut s'agir de la première valeur, de la valeur située au milieu, ...
- On crée deux tableaux : les éléments plus petits ou égaux au pivot, les éléments plus grands que le pivot. C'est donc un tri par partition.
- On trie ces deux tableaux.
- On concatène le tableau de plus petits, le pivot et le tableau des plus grands.
- C'est donc là encore un algorithme récursif que nous étudierons plus tard (on peut l'écrire de façon impérative mais c'est un exercice peu naturel).

# Tri rapide de Hoare : principe

- On prend une valeur du tableau (le "pivot"). Il peut s'agir de la première valeur, de la valeur située au milieu, ...
- On crée deux tableaux : les éléments plus petits ou égaux au pivot, les éléments plus grands que le pivot. C'est donc un tri par partition.
- On trie ces deux tableaux.
- On concatène le tableau de plus petits, le pivot et le tableau des plus grands.
- C'est donc là encore un algorithme récursif que nous étudierons plus tard (on peut l'écrire de façon impérative mais c'est un exercice peu naturel).

# Tri rapide de Hoare : principe

- On prend une valeur du tableau (le "pivot"). Il peut s'agir de la première valeur, de la valeur située au milieu, ...
- On crée deux tableaux : les éléments plus petits ou égaux au pivot, les éléments plus grands que le pivot. C'est donc un tri par partition.
- On trie ces deux tableaux.
- On concatène le tableau de plus petits, le pivot et le tableau des plus grands.
- C'est donc là encore un algorithme récursif que nous étudierons plus tard (on peut l'écrire de façon impérative mais c'est un exercice peu naturel).

# Tri rapide de Hoare : principe

- On prend une valeur du tableau (le "pivot"). Il peut s'agir de la première valeur, de la valeur située au milieu, ...
- On crée deux tableaux : les éléments plus petits ou égaux au pivot, les éléments plus grands que le pivot. C'est donc un tri par partition.
- On trie ces deux tableaux.
- On concatène le tableau de plus petits, le pivot et le tableau des plus grands.
- C'est donc là encore un algorithme récursif que nous étudierons plus tard (on peut l'écrire de façon impérative mais c'est un exercice peu naturel).

# Dictionnaires

# Les dictionnaires

- Les *dictionnaires* (également appelés *hachages* ou *tableaux associatifs*) sont des types composés qui ne sont **pas** des séquences : leurs élements ne sont pas rangés séquentiellement, comme dans les chaînes, les listes ou les tuples.
- Un dictionnaire peut être vu comme une liste de valeurs dont les indices ne sont plus des entiers mais sont d'un type **immutable** quelconque : des chaînes ou des tuples par exemple. Ces indices sont appelés *clés*.
- Un dictionnaire est donc une liste d'association clés/valeurs. La place d'une valeur dans un dictionnaire est indéterminée et une clé est **unique** (comme dans une table SQL).

**Remarque**

*En réalité, les clés ne doivent pas seulement être immutables : elles doivent également être « hachables », c'est-à-dire disposer d'une méthode hash(), ce qui est le cas des chaînes et des tuples.*

*En L3, vous verrez comment utiliser d'autres types de clés et comment implémenter cette méthode hash().*

# Les dictionnaires

- Les *dictionnaires* (également appelés *hachages* ou *tableaux associatifs*) sont des types composés qui ne sont **pas** des séquences : leurs élements ne sont pas rangés séquentiellement, comme dans les chaînes, les listes ou les tuples.
- Un dictionnaire peut être vu comme une liste de valeurs dont les indices ne sont plus des entiers mais sont d'un type **immutable** quelconque : des chaînes ou des tuples par exemple. Ces indices sont appelés *clés*.
- Un dictionnaire est donc une liste d'association clés/valeurs. La place d'une valeur dans un dictionnaire est indéterminée et une clé est **unique** (comme dans une table SQL).

**Remarque**

*En réalité, les clés ne doivent pas seulement être immutables : elles doivent également être « hachables », c'est-à-dire disposer d'une méthode hash(), ce qui est le cas des chaînes et des tuples.*

*En L3, vous verrez comment utiliser d'autres types de clés et comment implémenter cette méthode hash().*

# Les dictionnaires

- Les *dictionnaires* (également appelés *hachages* ou *tableaux associatifs*) sont des types composés qui ne sont **pas** des séquences : leurs élements ne sont pas rangés séquentiellement, comme dans les chaînes, les listes ou les tuples.
- Un dictionnaire peut être vu comme une liste de valeurs dont les indices ne sont plus des entiers mais sont d'un type **immutable** quelconque : des chaînes ou des tuples par exemple. Ces indices sont appelés *clés*.
- Un dictionnaire est donc une liste d'association clés/valeurs. La place d'une valeur dans un dictionnaire est indéterminée et une clé est **unique** (comme dans une table SQL).

**Remarque**

*En réalité, les clés ne doivent pas seulement être immutables : elles doivent également être « hachables », c'est-à-dire disposer d'une méthode `hash()`, ce qui est le cas des chaînes et des tuples.*

*En L3, vous verrez comment utiliser d'autres types de clés et comment implémenter cette méthode `hash()`.*

# Les dictionnaires

- Les *dictionnaires* (également appelés *hachages* ou *tableaux associatifs*) sont des types composés qui ne sont **pas** des séquences : leurs élements ne sont pas rangés séquentiellement, comme dans les chaînes, les listes ou les tuples.
- Un dictionnaire peut être vu comme une liste de valeurs dont les indices ne sont plus des entiers mais sont d'un type **immutable** quelconque : des chaînes ou des tuples par exemple. Ces indices sont appelés *clés*.
- Un dictionnaire est donc une liste d'association clés/valeurs. La place d'une valeur dans un dictionnaire est indéterminée et une clé est **unique** (comme dans une table SQL).

**Remarque**

*En réalité, les clés ne doivent pas seulement être immutables : elles doivent également être « hachables », c'est-à-dire disposer d'une méthode `hash()`, ce qui est le cas des chaînes et des tuples.*

*En L3, vous verrez comment utiliser d'autres types de clés et comment implémenter cette méthode `hash()`.*

# Les dictionnaires

- Les *dictionnaires* (également appelés *hachages* ou *tableaux associatifs*) sont des types composés qui ne sont **pas** des séquences : leurs élements ne sont pas rangés séquentiellement, comme dans les chaînes, les listes ou les tuples.
- Un dictionnaire peut être vu comme une liste de valeurs dont les indices ne sont plus des entiers mais sont d'un type **immutable** quelconque : des chaînes ou des tuples par exemple. Ces indices sont appelés *clés*.
- Un dictionnaire est donc une liste d'association clés/valeurs. La place d'une valeur dans un dictionnaire est indéterminée et une clé est **unique** (comme dans une table SQL).

**Remarque**

*En réalité, les clés ne doivent pas seulement être immutables : elles doivent également être « hachables », c'est-à-dire disposer d'une méthode hash(), ce qui est le cas des chaînes et des tuples.*

*En L3, vous verrez comment utiliser d'autres types de clés et comment implémenter cette méthode hash().*

# Utilisation des dictionnaires

- Les dictionnaires sont particulièrement utiles pour accéder directement à des valeurs lorsque l'on connaît leur clé. Ils sont du type *dict*.
- La recherche d'un élément dans un dictionnaire est immédiate : si la clé est présente, on accède directement à la valeur. On a vu que la recherche d'un élément dans une liste, en revanche, dépend de la longueur de la liste puisqu'il faut parcourir toute la liste si l'élément ne s'y trouve pas.
- On utilise un dictionnaire à chaque fois que l'on veut stocker des valeurs sans se soucier de leurs emplacements : tout ce qui compte est que les valeurs soient stockées et pouvoir les retrouver le plus rapidement possible.

# Utilisation des dictionnaires

- Les dictionnaires sont particulièrement utiles pour accéder directement à des valeurs lorsque l'on connaît leur clé. Ils sont du type *dict*.
- La recherche d'un élément dans un dictionnaire est immédiate : si la clé est présente, on accède directement à la valeur. On a vu que la recherche d'un élément dans une liste, en revanche, dépend de la longueur de la liste puisqu'il faut parcourir toute la liste si l'élément ne s'y trouve pas.
- On utilise un dictionnaire à chaque fois que l'on veut stocker des valeurs sans se soucier de leurs emplacements : tout ce qui compte est que les valeurs soient stockées et pouvoir les retrouver le plus rapidement possible.

# Utilisation des dictionnaires

- Les dictionnaires sont particulièrement utiles pour accéder directement à des valeurs lorsque l'on connaît leur clé. Ils sont du type *dict*.
- La recherche d'un élément dans un dictionnaire est immédiate : si la clé est présente, on accède directement à la valeur. On a vu que la recherche d'un élément dans une liste, en revanche, dépend de la longueur de la liste puisqu'il faut parcourir toute la liste si l'élément ne s'y trouve pas.
- On utilise un dictionnaire à chaque fois que l'on veut stocker des valeurs sans se soucier de leurs emplacements : tout ce qui compte est que les valeurs soient stockées et pouvoir les retrouver le plus rapidement possible.

# Utilisation des dictionnaires : accès aux valeurs

- Le dictionnaire vide se note *{}*
- L'accès aux valeurs utilise la notation entre crochets, comme les listes, sauf que l'indice est remplacé par la clé : *personne["Nom"]* renvoie la valeur associée à la clé *Nom* pour la variable *personne*.
- Si l'on tente de lire la valeur associée à une clé inexistante, Python déclenche l'erreur *KeyError* (il faut donc toujours tester l'existence d'une clé avant d'y accéder...).
- Si l'on tente d'écrire une valeur associée à une clé inexistante, Python crée une nouvelle association pour cette clé. Si la clé existait déjà, son ancienne valeur est remplacée par la nouvelle.

# Utilisation des dictionnaires : accès aux valeurs

- Le dictionnaire vide se note *{}*
- L'accès aux valeurs utilise la notation entre crochets, comme les listes, sauf que l'indice est remplacé par la clé : *personne["Nom"]* renvoie la valeur associée à la clé *Nom* pour la variable *personne*.
- Si l'on tente de lire la valeur associée à une clé inexistante, Python déclenche l'erreur *KeyError* (il faut donc toujours tester l'existence d'une clé avant d'y accéder...).
- Si l'on tente d'écrire une valeur associée à une clé inexistante, Python crée une nouvelle association pour cette clé. Si la clé existait déjà, son ancienne valeur est remplacée par la nouvelle.

# Utilisation des dictionnaires : accès aux valeurs

- Le dictionnaire vide se note *{}*
- L'accès aux valeurs utilise la notation entre crochets, comme les listes, sauf que l'indice est remplacé par la clé : *personne["Nom"]* renvoie la valeur associée à la clé *Nom* pour la variable *personne*.
- Si l'on tente de lire la valeur associée à une clé inexistante, Python déclenche l'erreur *KeyError* (il faut donc toujours tester l'existence d'une clé avant d'y accéder...).
- Si l'on tente d'écrire une valeur associée à une clé inexistante, Python crée une nouvelle association pour cette clé. Si la clé existait déjà, son ancienne valeur est remplacée par la nouvelle.

# Utilisation des dictionnaires : accès aux valeurs

- Le dictionnaire vide se note *{}*
- L'accès aux valeurs utilise la notation entre crochets, comme les listes, sauf que l'indice est remplacé par la clé : *personne["Nom"]* renvoie la valeur associée à la clé *Nom* pour la variable *personne*.
- Si l'on tente de lire la valeur associée à une clé inexistante, Python déclenche l'erreur *KeyError* (il faut donc toujours tester l'existence d'une clé avant d'y accéder...).
- Si l'on tente d'écrire une valeur associée à une clé inexistante, Python crée une nouvelle association pour cette clé. Si la clé existait déjà, son ancienne valeur est remplacée par la nouvelle.

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : opérations

- La fonction *len()* renvoie le nombre d'entrées du dictionnaire qui lui est passé en paramètre.
- Les méthodes *keys()*, *values()* et *items()* renvoient les listes des clés, des valeurs et des couples clé/valeur d'un dictionnaire. Le résultat n'est pas trié, mais on peut le trier avec *sorted()* (voir exemples).
- La méthode *get(clé, valdef)* renvoie la valeur associée à *clé*, ou *valdef* si clé n'est pas dans le dictionnaire.
- La fonction *del()* supprime l'entrée ayant la clé indiquée dans un dictionnaire.
- Comme pour les listes, on peut créer des dictionnaires en intension.

**Remarque**

*En réalité, keys(), values() et items() ne renvoient pas des listes mais des* ***itérateurs*** *qui peuvent être parcourus avec une boucle for . . . in . . .*

*Pour obtenir des listes, il faut les convertir avec list() (voir transparent suivant)*

# Utilisation des dictionnaires : exemples

```
en_to_fr = {}                               # création d'un dico vide
en_to_fr['red'] = 'rouge'                   # nouvelles associations clé -> valeur
en_to_fr|'blue'] = 'bleu'
en_to_fr['green'] = 'vert'

print("red is", en_to_fr['red'])            # Affiche 'red is rouge'

fr_to_en = { 'rouge': 'red', 'vert': 'green', 'bleu': 'blue'}
len(fr_to_en)                               # 3

list(en_to_fr)                              # ['blue', 'red', 'green']
list(en_to_fr.keys())                       # idem
list(en_to_fr.values())                     # ['bleu', 'rouge', 'vert']
list(en_to_fr.items())                      # [('blue', 'bleu'), ('red', 'rouge'), ('green', 'vert')]

for color in en_to_fr.keys():
    print(en_to_fr[color], end=', ')        # Affiche 'rouge, bleu, vert,'

for color in en_to_fr:
    print(en_to_fr[color], end=', ')        # idem

for color, couleur in en_to_fr.items():
    print(f"{color} en français se dit {couleur}")
```

# Utilisation des dictionnaires : exemples

```
en_to_fr['purple']                          # KeyError : 'purple'

if 'purple' in en_to_fr:
    print(en_to_fr['purple'])               # N'affichera donc rien...

print(en_to_fr.get('purple', 'inconnu'))    # Affichera 'inconnu'

for color in sorted(en_to_fr):              # Tri sur les clés
    print(en_to_fr[color], end=', ')        # Affichera 'bleu, vert, rouge,'

del en_to_fr['blue']                        # Supprime une entrée (si elle existe)

liste = [1, 2, 3, 4]
dico_carres = { cle: cle**2 for cle in liste }              # {1: 1, 2: 4, 3: 9, 4: 16}
dico_cubes = { cle: cle**3 for cle in liste if cle > 2 }    # {3: 27, 4: 64}
```

# Cas d'utilisation typique : compter les mots d'une phrase

- On veut compter le nombre d'occurrences de chaque mot d'une phrase saisie au clavier.
- On utilise un dictionnaire où les clés seront les mots et les valeurs seront les nombres d'occurrences de ces mots.
- À chaque fois qu'on rencontre un mot dans la phrase, on incrémente son compteur associé.
- À la fin, on affiche les occurrences des mots triés par ordre alphabétique.

# Cas d'utilisation typique : compter les mots d'une phrase

- On veut compter le nombre d'occurrences de chaque mot d'une phrase saisie au clavier.
- On utilise un dictionnaire où les clés seront les mots et les valeurs seront les nombres d'occurrences de ces mots.
- À chaque fois qu'on rencontre un mot dans la phrase, on incrémente son compteur associé.
- À la fin, on affiche les occurrences des mots triés par ordre alphabétique.

# Cas d'utilisation typique : compter les mots d'une phrase

- On veut compter le nombre d'occurrences de chaque mot d'une phrase saisie au clavier.
- On utilise un dictionnaire où les clés seront les mots et les valeurs seront les nombres d'occurrences de ces mots.
- À chaque fois qu'on rencontre un mot dans la phrase, on incrémente son compteur associé.
- À la fin, on affiche les occurrences des mots triés par ordre alphabétique.

# Cas d'utilisation typique : compter les mots d'une phrase

- On veut compter le nombre d'occurrences de chaque mot d'une phrase saisie au clavier.
- On utilise un dictionnaire où les clés seront les mots et les valeurs seront les nombres d'occurrences de ces mots.
- À chaque fois qu'on rencontre un mot dans la phrase, on incrémente son compteur associé.
- À la fin, on affiche les occurrences des mots triés par ordre alphabétique.

# Cas d'utilisation typique : compter les mots d'une phrase

```python
import re                                          # Pour les expressions régulières

def compter_lettres(phrase):
    """Renvoie un dictionnaire contenant les occurrences de chaque mot de la phrase"""
    occurrences = {}                               # Création du dictionnaire

    for mot in re.split('\W+', phrase):            # Découpe phrase sur les espaces
        mot = mot.lower()
        occurrences[mot] = occurrences.get(mot, 0) + 1

    return occurrences

# Prog principal
phrase = input("Entrez une phrase : ")             # To be or not to be, that is the question

compteurs_mots = compter_lettres(phrase)
for mot in sorted(compteurs_mots.keys()):          # On veut trier sur les mots
    print(f"'{mot}' apparaît {compteurs_mots[mot]} fois")
```

# Ensembles

# Les ensembles

- Un ensemble est une collection de données **non ordonnées et non dupliquées**.
- Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être **hachables et immutables** (ce qui est le cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires, ni des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- On utilise un ensemble lorsque l'on veut stocker des valeurs uniques et que l'on souhaite simplement savoir si une valeur appartient à cet ensemble.

# Les ensembles

- Un ensemble est une collection de données **non ordonnées et non dupliquées**.
- Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être **hachables et immutables** (ce qui est le cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires, ni des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- On utilise un ensemble lorsque l'on veut stocker des valeurs uniques et que l'on souhaite simplement savoir si une valeur appartient à cet ensemble.

# Les ensembles

- Un ensemble est une collection de données **non ordonnées et non dupliquées**.
- Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être **hachables et immutables** (ce qui est le cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires, ni des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- On utilise un ensemble lorsque l'on veut stocker des valeurs uniques et que l'on souhaite simplement savoir si une valeur appartient à cet ensemble.

# Les ensembles

- Un ensemble est une collection de données **non ordonnées et non dupliquées**.
- Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être **hachables et immutables** (ce qui est le cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires, ni des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- On utilise un ensemble lorsque l'on veut stocker des valeurs uniques et que l'on souhaite simplement savoir si une valeur appartient à cet ensemble.

# Les ensembles : utilisation

- En Python, les ensembles sont du type *set*, on ajoute un élément avec la méthode *add()*, on en supprime avec les méthodes *remove()* ou *discard()*, on teste l'appartenance avec les opérateurs *in* ou *not in*.
- Les opérations d'union, d'intersection et de différence symétrique sont, respectivement, assurées par les opérateurs |, & et ^ (ou par les méthodes *union()*, *intersection()* et *symmetric_difference()*).
- La différence ensembliste est implémentée par l'opérateur - ou la méthode *difference()* (voir *help(set)* et la doc en ligne pour les autres opérations...)

# Les ensembles : utilisation

- En Python, les ensembles sont du type *set*, on ajoute un élément avec la méthode *add()*, on en supprime avec les méthodes *remove()* ou *discard()*, on teste l'appartenance avec les opérateurs *in* ou *not in*.
- Les opérations d'union, d'intersection et de différence symétrique sont, respectivement, assurées par les opérateurs |, *&* et ^ (ou par les méthodes *union()*, *intersection()* et *symmetric_difference()*).
- La différence ensembliste est implémentée par l'opérateur - ou la méthode *difference()* (voir *help(set)* et la doc en ligne pour les autres opérations...)

# Les ensembles : utilisation

- En Python, les ensembles sont du type *set*, on ajoute un élément avec la méthode *add()*, on en supprime avec les méthodes *remove()* ou *discard()*, on teste l'appartenance avec les opérateurs *in* ou *not in*.
- Les opérations d'union, d'intersection et de différence symétrique sont, respectivement, assurées par les opérateurs |, *&* et ^ (ou par les méthodes *union()*, *intersection()* et *symmetric_difference()*).
- La différence ensembliste est implémentée par l'opérateur - ou la méthode *difference()* (voir *help(set)* et la doc en ligne pour les autres opérations...)

# Les ensembles : autres opérations

- Comme pour les autres collections, la fonction *len()* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments (qui sont dans un ordre quelconque car un ensemble n'est pas une séquence... mais on peut trier avec *sorted()*, comme on l'a vu pour les dictionnaires).
- L'ensemble vide se note *set()*.
- On peut construire un ensemble non vide à partir de n'importe quelle séquence (les éléments dupliqués seront supprimés)
- Comme pour les listes et les dictionnaires, on peut créer des ensembles en intension.

# Les ensembles : autres opérations

- Comme pour les autres collections, la fonction *len()* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments (qui sont dans un ordre quelconque car un ensemble n'est pas une séquence... mais on peut trier avec *sorted()*, comme on l'a vu pour les dictionnaires).
- L'ensemble vide se note *set()*.
- On peut construire un ensemble non vide à partir de n'importe quelle séquence (les éléments dupliqués seront supprimés)
- Comme pour les listes et les dictionnaires, on peut créer des ensembles en intension.

# Les ensembles : autres opérations

- Comme pour les autres collections, la fonction *len()* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments (qui sont dans un ordre quelconque car un ensemble n'est pas une séquence... mais on peut trier avec *sorted()*, comme on l'a vu pour les dictionnaires).
- L'ensemble vide se note *set()*.
- On peut construire un ensemble non vide à partir de n'importe quelle séquence (les éléments dupliqués seront supprimés)
- Comme pour les listes et les dictionnaires, on peut créer des ensembles en intension.

# Les ensembles : autres opérations

- Comme pour les autres collections, la fonction *len()* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments (qui sont dans un ordre quelconque car un ensemble n'est pas une séquence... mais on peut trier avec *sorted()*, comme on l'a vu pour les dictionnaires).
- L'ensemble vide se note *set()*.
- On peut construire un ensemble non vide à partir de n'importe quelle séquence (les éléments dupliqués seront supprimés)
- Comme pour les listes et les dictionnaires, on peut créer des ensembles en intension.

# Les ensembles : autres opérations

```
s = set([1, 3, 5, 7])
t = set([1, 2, 3, 4, 6, 8])
s.union(t)                      # set([1, 2, 3, 4, 5, 6, 7, 8])
s | t                           # idem
s & t                           # set([1, 3])
s - t                           # set([5, 7])
s ^ t                           # set([2, 4, 5, 6, 7, 8])
s.issubset(set(range(1,10)))    # True
s.add(3)                        # s non modifié
s.remove(2)                     # KeyError
if 2 in s: s.remove(2)          # s non modifié
s.discard(2)                    # pas d'erreur et s non modifié

# On exploite le fait que les ensembles sont implémentés à l'aide de dict :
u = {1, 3, 4, 12}                           # set([4, 3, 12, 1])
u = { e for e in range(1,20) if e % 2 == 0 }  # ensemble en intension

# set appliqué à un dictionnaire renvoie l'ensemble de ses clés :
moi = {'prénom': 'Eric', 'nom': 'Jacoboni', 'age': 20}
champs = set(moi)               # set(['age', 'nom', 'prénom'])
```

# Les ensembles : tests d'appartenance

- Les ensembles étant implémentés par des dictionnaires, ils sont particulièrement adaptés aux tests d'appartenance :

```
> python3 -m timeit -s 'li = list(range(100))' '"x" in li'
100000 loops, best of 3: 2.17 usec per loop
> python3 -m timeit -s 'li = set(range(100))' '"x" in li'
10000000 loops, best of 3: 0.0305 usec per loop
```

- Même le cas le plus favorable des listes est à peine meilleur que les ensembles :

```
> python3 -m timeit -s 'li = list(range(100))' '0 in li'
10000000 loops, best of 3: 0.0243 usec per loop
> python3 -m timeit -s 'li = set(range(100))' '0 in li'
10000000 loops, best of 3: 0.0313 usec per loop
```

# Les ensembles : tests d'appartenance

- Les ensembles étant implémentés par des dictionnaires, ils sont particulièrement adaptés aux tests d'appartenance :

```
> python3 -m timeit -s 'li = list(range(100))' '"x" in li'
100000 loops, best of 3: 2.17 usec per loop
> python3 -m timeit -s 'li = set(range(100))' '"x" in li'
10000000 loops, best of 3: 0.0305 usec per loop
```

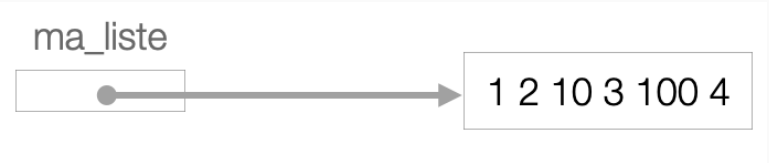
- Même le cas le plus favorable des listes est à peine meilleur que les ensembles :

```
> python3 -m timeit -s 'li = list(range(100))' '0 in li'
10000000 loops, best of 3: 0.0243 usec per loop
> python3 -m timeit -s 'li = set(range(100))' '0 in li'
10000000 loops, best of 3: 0.0319 usec per loop
```

# Paramètres collections

# Paramètres collections

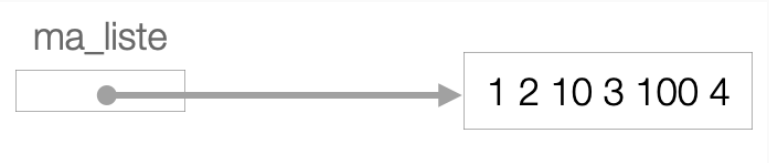
- Les variables collection étant des objets, ceci a des conséquences lorsqu'on les passe en paramètre à un sous-programme.
- En réalité, une « variable » collection ne contient pas la collection proprement dite, mais une *référence* vers la véritable collection qui, elle, est stockée ailleurs.




- On a vu qu'un sous-programme Python ne peut pas modifier le contenu du paramètre effectif (passage par copie) mais, si c'est un objet modifiable, il peut modifier l'objet en passant par sa référence qui a été passée en paramètre.

# Paramètres collections

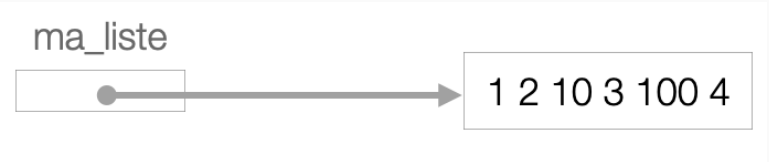
- Les variables collection étant des objets, ceci a des conséquences lorsqu'on les passe en paramètre à un sous-programme.
- En réalité, une « variable » collection ne contient pas la collection proprement dite, mais une *référence* vers la véritable collection qui, elle, est stockée ailleurs.




- On a vu qu'un sous-programme Python ne peut pas modifier le contenu du paramètre effectif (passage par copie) mais, si c'est un objet modifiable, il peut modifier l'objet en passant par sa référence qui a été passée en paramètre.

# Paramètres collections

- Les variables collection étant des objets, ceci a des conséquences lorsqu'on les passe en paramètre à un sous-programme.
- En réalité, une « variable » collection ne contient pas la collection proprement dite, mais une *référence* vers la véritable collection qui, elle, est stockée ailleurs.




- On a vu qu'un sous-programme Python ne peut pas modifier le contenu du paramètre effectif (passage par copie) mais, si c'est un objet modifiable, il peut modifier l'objet en passant par sa référence qui a été passée en paramètre.

# Paramètres collections : exemple

```
def echanger(a, b):
    a, b = b, a

def echanger_bis(tab):
    tab[0], tab[1] = tab[1], tab[0]

def main():
    val1, val2 = 10, 100
    vals = [10, 100]
    vals_bis = (10, 100)

    echanger(val1, val2)
    echanger_bis(vals)
    echanger_bis(vals_bis)
    print(val1, val2)                    # 10 100 => val1 et val2 n'ont pas été modifiées
    print(vals)                          # [100, 10] => la liste pointée par vals a été modifée
    print(vals_bis)                      # (10, 100) => le tuple pointé par vals_bis n'a pas été modifé

main()
```