Structures de données composées :

les tranches et les chaînes

Les tranches

Tableaux, tranches, listes ?

- Selon les langages, on emploie les termes de **tableaux** ou de **listes** pour désigner un **type composé** d'éléments **de même type**, rangés les uns après les autres...

- Généralement, la notion de "**tableau**" fait référence à une collection d'une **taille fixe** (par exemple, un tableau de 10 entiers) alors que la notion de "**liste**" fait référence à une collection dont **la taille peut varier** en cours de programme (on parle aussi de **tableaux dynamiques**, par opposition aux **tableaux statiques**).

- Certains langages n'ont que des tableaux statiques (C, par exemple), d'autres n'ont que des tableaux dynamiques/listes (Python, par exemple). Go a les deux mais il utilise le terme de "**tranche**" pour désigner les listes.

- En pratique, on utilise peu les tableaux en Go car les tranches sont bien plus souples d'emploi et c'est la raison pour laquelle nous ne les présenterons pas dans ce cours.

Les tranches Go

- Une tranche (*slice*) est un « tableau dynamique », c'est-à-dire une collection d'éléments **de même type**. On parlera donc de "tranche d'entiers", de "tranche de chaînes", "tranches de caractères", etc.

- La taille d'une tranche **peut varier en fonction des besoins**. Son type se note **[]type**, où **type** est le type des éléments de la tranche: **[]int**, par exemple, est le type « tranche d'entiers ».

- La façon la plus simple de créer une tranche consiste à utiliser un **littéral tranche** :

```
var daltons = []string{"Joe", "Jack", "William", "Averell"}      // Tranche de chaînes
notes := []int{10, 5, 12, 16, 9}                                 // Tranche d'entiers
```

Les tranches Go : tranches vides et fonction make

- On peut créer une tranche vide (mais il faut préciser son type) :

```
var slice = []int{}   // slice est de type "tranche de int", initialisée avec une tranche vide
slice2 := []int{}     // idem en plus court
```

- Ou créer une tranche avec une taille initiale (ses éléments vaudront le "zéro du type"). Cette taille pourra évoluer par la suite : elle n'est pas fixe...

```
var tranche = make([]int, 10)    // tranche initialisée avec 10 éléments égaux à zéro
slice := make([]bool, 5)         // slice est une tranche de 5 booléens, initialisés à false
```

Les tranches Go : accès à un élément et nombre d'éléments

- Les éléments d'une tranche sont placés consécutivement les uns après les autres.

- Pour accéder à un élément particulier, il faut donc indiquer son emplacement (son "**indice**") dans la tranche : **notes[2]** désigne le **3e** élément de la tranche **notes** (car les indices commencent à 0).

- Pour connaître le nombre d'éléments dans une tranche, on dispose de la fonction **len** : par exemple, **len(notes)** renvoie le nombre d'éléments de **notes** (ce qui signifie que **le dernier indice de notes est égal à len(notes) - 1**...)

Tranches Go : accès à un élément

- Si **notes** est défini comme **notes := []int{10, 5, 12, 16, 9}** :
  - **notes[0]** est égal à 10
  - **notes[3]** est égal à 16
  - **notes[len(notes)-1]** est égal à 9
  - un accès à **notes[7]** provoquera une erreur
- **notes** étant une tranche d'entiers, **tous ses éléments notes[i] sont donc des entiers et se comportent comme tels** : on peut les modifier, les afficher, faire des calculs avec, etc.

Les tranches Go : création à partir d'une autre tranche

- Pour créer une tranche **t2** à partir d'une tranche **t1**, on utilisera la notation **t2 := t1[deb:fin]**, où **deb** et **fin** sont, respectivement, les indices de début (compris) et de fin (exclus) dans **t1**...

- Si deb vaut 0 (on part du début de t1), on peut utilser la notation **t1[:fin]**.

- Si fin vaut le dernier indice de t1, on peut écrire **t1[deb:]**

Les tranches Go : création à partir d'une autre tranche

- **Attention** : la tranche créée à partir d'une autre tranche **partagera ses données** avec celles de la tranche initiale. On peut considérer que la nouvelle tranche est une "fenêtre" sur la tranche initiale.

- Les tranches étant modifiables, la modification des données d'un côté sera donc répercutée de l'autre côté :

```
daltons := []string{"Joe", "Jack", "William", "Averell"}
tranche := daltons[1:3]                                  // ["Jack", "William"]
tab := []int{12, 3, 1, 9, -6}
slice := tab[1:4]                                        //  [3, 1, 9]

slice[2] = 66                                            // slice vaut maintenant [3, 1, 66]
tranche[0] = "Ma"                                        // tranche vaut maintenant ["Ma", "William"]
fmt.Printf("daltons = %v\n", daltons)                    // ["Joe", "Ma", "William", "Averell"]
fmt.Printf("tab = %v\n", tab)                            // [12, 3, 1, 66, -6]
```

Les tranches Go : la fonction **copy**

- La fonction **copy** permet de copier le contenu d'une tranche **sans le partager**, mais il faut auparavant créer la tranche destinataire avec une taille *au moins égale à celle de la tranche source* (voir ce lien) :

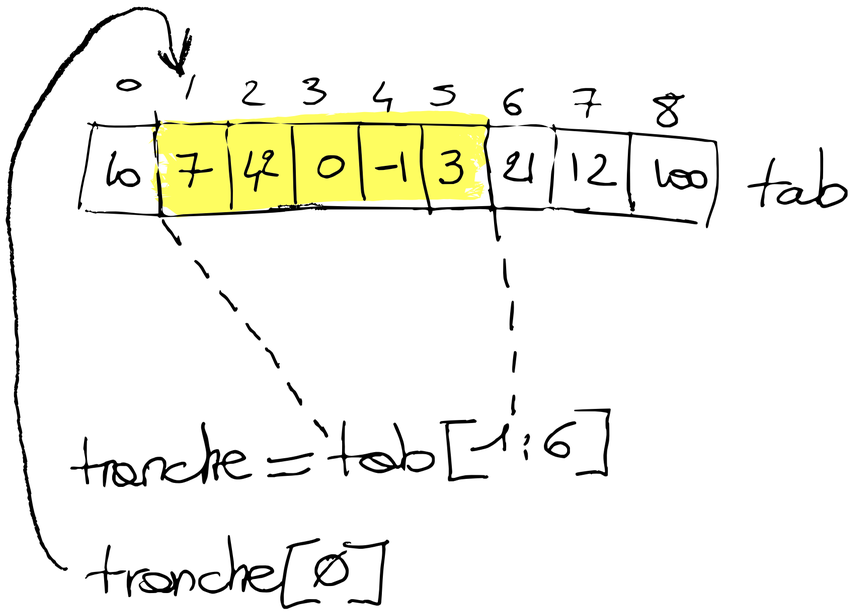
```
daltons := []string{"Joe", "Jack", "William", "Averell"}
bandits := make([]string, len(daltons))     // Création d'une tranche de même taille que daltons

copy(bandits, daltons)                      // Le premier paramètre est le destinataire, comme avec =
bandits[0] = "Ma"
fmt.Printf("daltons = %v\n", daltons)     // ["Joe", "Jack", "William", "Averell"]  daltons n'a pas été modifié...
fmt.Printf("bandits = %v\n", bandits)     // ["Ma", "Jack", "William", "Averell"]
```

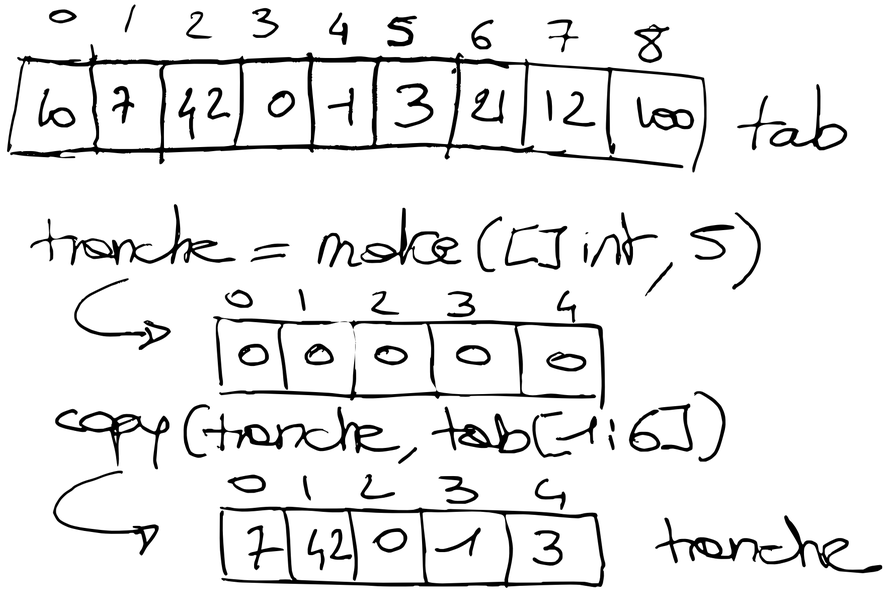
- Il faut évidemment que la tranche destinataire soit du même type que la tranche de départ..

Tranche de tranche : avec et sans copy...

Sans copy



Avec copy



Les tranches Go : la fonction **append**

- On peut ajouter un ou plusieurs éléments à la fois avec **append**. Si l'on veut ajouter le contenu complet d'une autre tranche, il faut utiliser la notation **...** qui "explose" une tranche en plusieurs éléments (voir dernier exemple)

```
daltons := []string{"Joe", "Jack", "William", "Averell"}

daltons = append(daltons, "Ma")    // "Joe", "Jack", "William", "Averell", "Ma"

tab := []int{10, 20, 30, 40}
tab2 := []int{100, 200}

tab3 := append(tab, tab2...)          // 10, 20, 30, 40, 100, 200

tab4 := []int{}                       // tab4 est une tranche vide
tab4 = append(tab4, 42, 12)           // 42, 12

tab5 := make([]int, 3)                // tab5 vaut initialement 0, 0, 0
tab5 = append(tab5, 42, 12)           // 0, 0, 0, 42, 12
```

Les tranches Go : la fonction **append**

- Si l'on ne veut pas qu'**append** modifie aussi la tranche initiale, il suffit de copier celle-ci dans une nouvelle tranche, qu'il faudra créer d'abord avec la bonne taille. Voir ce lien.

```
tab := []int{12, 3, 1, 66, -6}
slice := make([]int, len(tab[1:4]))      // Crée une NOUVELLE tranche slice de 3 éléments
copy(slice, tab[1:4])                    // COPIE les éléments d'indice 1 à 3 de tab dans slice
```

- Résumé pour le partage entre tranches :
  - Si vous créez une tranche B à partir d'une autre tranche A et que vous comptez ensuite modifier B *tout en continuant à vous servir de A*, utilisez **make + copy**…
  - Si vous ne comptez plus vous servir de A après avoir créé B ou si vous ne comptez pas modifier B, la forme classique **tab[deb:fin]** suffit.

Les tranches Go : diminution d'une tranche et comparaisons

- Pour réduire une tranche, il suffit de lui affecter une tranche plus petite :

```
slice2 := []int{10, 9, 8, 7, 6, 5, 4}
slice2 = slice2[:5]         // [10, 9, 8, 7, 6]

slice2 := []int{10, 9, 8, 7, 6, 5, 4}
slice2 = slice2[3:]          // [7, 6, 5, 4]

slice2 := []int{10, 9, 8, 7, 6, 5, 4}
slice2 = append(slice2[:3], slice2[6:]...)         // [10 9 8 4]
```

- On ne peut pas comparer deux tranches entre elles. La seule comparaison autorisée sur une tranche est sa comparaison avec **nil** (pour savoir si la tranche est "vide")

Les tranches Go : optimisation

- Si vous connaissez d'avance le nombre d'éléments minimum que vous voudrez stocker dans une tranche, il est préférable de réserver d'avance ce nombre d'emplacements avec **make()** lorsque vous créez la tranche : cela vous permettra ensuite d'utiliser une simple notation entre crochets pour placer vos éléments (**append()** restera disponible pour ajouter des éléments supplémentaires)

- En revanche, la création d'une tranche initialement vide et l'ajout d'éléments par des appels répétés à **append()** est bien plus coûteux en occupation mémoire et en temps d'exécution.

Les tranches Go : optimisation

- Comparons ces deux extraits de code :

```
const NBELTS = ...    // un grand nombre
ints := []int{}       // ints est une tranche vide

for i := 0; i < NBELTS; i++ {
    ints = append(ints, i)
}
```

```
const NBELTS = ...  // un grand nombre
ints := make([]int, NBELTS)  // ints est une tranche de NBELTS entiers

for i := 0; i < NB_ELTS; i++ {
    ints[i] = i
}
```

- Les deux réalisent la même opération : remplir une grosse tranche d'entiers avec des valeurs allant de 0 à NBELTS - 1.

- Mais le second consomme **5 fois moins de mémoire** et s'exécute **8 fois plus vite**...

Parcours de tranches

- Il existe deux sortes de parcours :
  - Les **parcours partiels** où l'on ne parcourt qu'une partie de la tranche (jusqu'à trouver un élément particulier, par exemple)
  - Les **parcours complets** où l'on parcourt la tranche du début à la fin (ou de la fin au début).

Parcours complet

- Go fournit une forme particulière de boucle pour parcourir une tranche du début à la fin : **for i, elt := range tranche {...}**, qui correspond à l'énoncé algorithmique "*Pour chaque élément de tranche Faire*"

```
slice := []int{10, 9, 8, 7, 6, 5, 4}

for i, elt := range slice {
    fmt.Printf("%v est à l'indice %v", elt, i)
}
```

- À chaque tour de boucle, **i** vaudra l'indice courant (en partant de 0) et **elt** contiendra une **copie** de l'élément à cet indice dans **slice** : ceci signifie que si vous modifiez **elt** dans la boucle, **cela ne modifiera pas slice...**

Parcours complets avec et sans modification

- Si vous souhaitez **modifier** un ou plusieurs éléments au cours du parcours, il **faut** passer par l'indice. En ce cas, vous n'avez plus besoin de **elt** :

```
slice := []int{10, 9, 8, 7, 6, 5, 4}
for i, _ := range slice {
    if slice[i] % 2 == 0 {
        slice[i] += 1
    }
}
```

- Inversement, si vous souhaitez simplement **consulter** les éléments sans utiliser leur indice, vous n'avez plus besoin de **i** :

```
for _, elt := range slice {
    if elt % 2 == 0 {
        ...
    }
}
```

Parcours complet "restreint"

- Un cas particulier de parcours complet est celui où l'on souhaite parcourir tous les éléments **compris entre deux indices**... En ce cas, il suffit de prendre une tranche entre ces deux indices et appliquer ce que nous venons de voir :

```
slice := []int{10, 9, 8, 7, 6, 5, 4}
for i, _ := range slice[1:5] {
    if slice[i] % 2 == 0 {
        slice[i] += 1
    }
}                                   // slice vaudra 10, 9, 9, 7, 7, 5, 4
```

Parcours partiel

- Ici, on ne souhaite plus parcourir une tranche du début à la fin, ni d'un indice donné à un autre : on veut la parcourir "tant qu'une certaine condition est vraie".

- En ce cas, on utilisera une boucle **Tant Que**, mais ce sera à nous de faire progresser le parcours en jouant sur l'indice et, **surtout**, à vérifier qu'on s'arrête bien à la fin de la tranche...

```
slice := []int{10, 20, 8, 7, 6, 5, 4}
i = 0
for i < len(slice) && slice[i] % 2 == 0 { // Attention à l'ordre !!!
    fmt.Printf("%v ", slice[i])
    i++
}                                          // Affichera 10, 20, 8
```

Fonctions renvoyant des tranches

- Une fonction Go peut avoir des paramètres tranches et renvoyer une tranche, comme n'importe quel autre type :

```
func pairs(nbres []int) []int {
    res := []int{}
    for _, nb := range nbres {
        if nb % 2 == 0 {
            res = append(res, nb)
        }
    }
    return res
}
```

Tranches passées en paramètres

- Nous avons vu que les fonctions utilisaient leurs paramètres pour produire un résultat mais qu'**elles ne pouvaient pas modifier ces paramètres** puisqu'ils étaient passés "par valeur" (ou "par copie").

- Dans le cas des tranches, c'est un peu différent : **on peut modifier son contenu** (c'est ce qu'on appelle un passage "par référence").

- Ce mode de passage permet donc d'écrire des sous-programmes qui modifient les tranches qui leur sont passées en paramètres.

- Attention à ne pas modifier un paramètre tranche accidentellemement : toute modification de cette tranche dans le sous-programme sera répercutée sur la tranche originale !

Tranches passées en paramètres

- Supposons que nous voulions écrire un sous-programme qui multiplie par deux tous les éléments d'une tranche d'entiers.

- On a plusieurs possibilités :

    - Écrire une **fonction** qui renvoie un résultat sans modifier la tranche originale.

    - Écrire une **procédure** (c-à-d, une fonction qui ne renvoie rien, mais qui modifie "sur place" la tranche originale).

  - La version fonction est "plus propre", mais moins efficace que la version procédurale puisqu'elle crée une deuxième tranche...

  - Comme on l'a déjà vu, un appel de fonction est une **expression,** contrairement à l'appel d'une procédure qui s'apparente plus à une **instruction**...

Tranches passées en paramètres

- Comparez les versions fonction et procédure...

- Remarquez que la version fonction peut être utilisée partout où une tranche d'entiers est attendue (ici, dans un Println).

- Remarquez que la version procédure doit faire l'objet d'une instruction à part, qui ne peut pas être intégrée dans une expression puisqu'elle ne renvoie rien.

- Remarquez aussi que, contrairement à la fonction, la procédure a modifié la tranche originale qui est donc désormais perdue...

Exemple de parcours total

```
func moyenne(notes []float64) (float64, error) {
    // Traitement du cas où le tableau est vide : on ne peut pas diviser par zéro
    if len(notes) == 0 {
        return 0.0, errors.New("le tableau de notes est vide")
    }
    // Sinon, on fait la somme des notes et on la divise par le nombre de notes
    somme := 0.0
    for _, note := range notes {
        somme += note
    }
    return somme / float64(len(notes)), nil
}
```

Exemple de parcours partiel

- On veut rechercher l'indice de la première occurrence d'un élément dans une tranche.
- Si l'élément ne se trouve pas dans la tranche, on renvoie une valeur qui ne peut pas être un indice : -1.
- Ici, on s'arrête dès qu'on a trouvé l'élément mais, si l'élément n'est pas dans la tranche, il faut s'arrêter après avoir testé le dernier élément.
- C'est donc un parcours partiel, donc un Tant Que...
- Voir ce code

Exemples d'exercices sur les tranches

- Afficher tous les nombres pairs d'une tranche d'entiers
- Inverser une tranche d'entiers
- Faire le ET d'une tranche de booléens
- Renvoyer une tranche ne contenant que les éléments supérieurs à une valeur donnée dans une tranche d'entiers
- Rechercher l'indice d'un élément particulier, à partir du début ou de la fin
- etc.