Les chaînes

Prolégomènes

- Pour les humains, une **chaîne de caractères** est, comme son nom l'indique, une suite de caractères : on pourrait donc en déduire qu'en Go, elle sera représentée par une tranche de caractères...

- Mais qu'est-ce qu'un "caractère" en Go puisqu'il n'existe pas de type **char** comme dans d'autres langages... ?

- Comme toute information, un caractère est représenté en mémoire par une suite de 1 et de 0 : cette représentation doit être la même dans tous les pays du monde et sur tous les systèmes. En outre, il faut aussi pouvoir coder des symboles comme la ponctuation, des caractères "invisibles" (comme la tabulation ou le retour à la ligne) et les émoji (comme 😀 ou 🕶...)

- Cette norme s'appelle **Unicode** et se décline en plusieurs versions, dont la plus connue est UTF-8 : elle consiste à représenter les caractères par un **entier codé par 1, 2, 3 ou 4 octets selon le cas**.

Chaînes Go : quoi ? les caractères sont des entiers ???

- Qu'est-ce que c'est que cette histoire ? **'a'** n'est pas un entier, c'est une lettre !
  - Certains langages ont choisi de créer un type spécial (**character** ou **char**, par exemple) qui se manipule de façon spéciale avec des opérations spéciales...
  - Go, comme le langage C son proche parent, a décidé qu'un caractère se manipulerait exactement comme son code (c-à-d un entier stocké sur 1 à 4 octets).
  - Il n'y a donc pas de "type caractère" en Go, mais un type **rune,** qui est simplement un synonyme de **int32** (entier sur 4 octets) et on peut donc faire avec une **rune** tout ce qu'on peut faire avec un **int32** (et réciproquement).

Chaînes Go : quoi ? les caractères sont des entiers ???

- Un **littéral caractère** se note entre **apostrophes simples** et sa valeur est donc un entier sur 1 à 4 octets : le code UTF-8 du caractère.
  - le littéral **'a'** est de type **rune** et est équivalent à la valeur **97** (le code UTF-8 de 'a', sur **un** octet)
  - le littéral **'é'** est de type **rune** et est équivalent à la valeur **233** (le code UTF-8 de 'é', sur **deux** octets),
  - le littéral **'界'** est de type **rune** et est équivalent à la valeur **30028** (**trois** octets).
  - le littéral '🥰' est de type **rune** et est équivalent à la valeur **129392** (**quatre** octets)**.**
  - En UTF-8, seuls les codes inférieurs à 128 sont codés sur un octet, les tailles des autres caractères varient de 2 à 4 octets. Voir ce code. Notamment, tous les caractères accentués sont codés sur deux octets.
- La **seule** façon d'afficher un caractère comme une lettre et non comme son code est d'utiliser le spécificateur de format **%c** de **Printf**... Voir ce code.

Les chaînes : résumé

- **Pour les humains**, une chaîne est une suite de **caractères** (chaque caractère est à un emplacement bien déterminé dans la chaîne).

- En UTF-8, les caractères **non accentués**, les symboles de ponctuation, les chiffres, les caractères de contrôle (tabulation et retour à la ligne, par exemple) sont codés sur **un seul octet** (ce qui correspond à l'ancien code ASCII). Les autres sont codés sur **au moins deux octets**.

- Les chaînes étant des tranches particulières, quasiment tout ce que nous avons expliqué pour les tranches reste applicable. Notamment, ce que l'on appelle une "sous-chaîne" est une tranche prise dans une chaîne entre deux indices et l'on peut parcourir une chaîne comme une tranche. Mais... **voir plus loin pour le problème des indices**.

- Une différence importante avec les tranches est que les chaînes sont "**immutables**" : une fois initialisées, elles ne peuvent plus être modifiées. On peut écraser une chaîne par une autre, mais on ne peut pas modifier son **contenu** (les caractères qui la composent)... Voir ce code.

Littéraux chaînes et chaînes brutes

- Les littéraux chaînes se notent entre guillemets (**"..."**), ou entre apostrophes inversées (**`...`**). Ne pas confondre avec les littéraux caractères, qui se notent entre apostrophes (**'...'**).

- Les chaînes entre apostrophes inversées sont des **chaînes brutes** : les symboles qu'elles contiennent ne sont pas interprétés et elles peuvent être écrites sur plusieurs lignes. Nous les utiliserons peu dans ce cours...

```
fmt.Println(`Homme libre,
toujours tu chériras la mer.
La mer est ton miroir\n
etc.`)
```

```
Homme libre,
toujours tu chériras la mer.
La mer est ton miroir\n
etc.
```

Littéraux chaînes et caractères spéciaux

- On peut mettre des apostrophes dans une chaîne entre guillemets (**"Il m'a dit"**). Mais, pour mettre des guillemets dans une chaîne, il faut utiliser une chaîne brute (entre apostrophes inversées) ou protéger ces guillemets :

```
chaine1 = `Il était "une fois"`

chaine2 = "dans la \"ville\" de Foix"
```

- Une chaîne entre guillemets peut contenir n'importe quel caractère, y compris des caractères spéciaux (**\n**, **\t**, ****, …) :
  - **\n** représente un caractère de retour à la ligne (*newline*)
  - **\t** représente un caractère de tabulation (*tab*).
  - **** représente le caractère ****

Opérations sur les chaînes

- Le **seul opérateur** sur les chaînes est l'opérateur **+** qui permet de "concaténer" (mettre bout à bout) deux chaînes (**on ne peut pas utiliser append() sur une chaîne**) :

```
ch1 = "Bonjour"
mess = ch1 + " a tous"      // mess = "Bonjour a tous"
ch2 = "cheval"
ch2 = ch2 + "s"             // ch2 = "chevals"
```

- Le module **strings** dispose d'un grand nombre de fonctions sur les chaînes. Il est fortement conseillé de l'étudier afin de ne pas réinventer la roue... (voir plus loin)

Chaines Go et tranches

- On a évoqué le fait qu'une chaîne étant une suite de caractères, elle pouvait s'apparenter à une tranche de caractères, donc à une tranche de runes puisque les caractères sont représentés par le type **rune** en Go...

- En réalité, un caractère pouvant être codé par un, deux, trois ou 4 octets, Go considère par défaut qu'une chaîne (de type **string**) est une **suite d'octets**. En Go, les octets sont représentés par le type **byte** (un alias de **uint8**)

- Mais, comme on le verra, il existe un moyen simple de lui expliquer qu'on veut considérer une chaîne comme une suite de caractères...

Opérations sur les chaînes

- Une chaîne étant donc **quasiment** une tranche d'octets, on peut lui appliquer la fonction **len(…)** qui renvoie **le nombre d'octets** de la chaîne qu'on lui a passée en paramètre.

- Ainsi, **len("Bonjour")** renverra **7** puisque chaque caractère de cette chaîne est codé sur un seul octet.

- En revanche, **len("était")** renverra **6** et non 5 car le '**é**' est codé sur deux octets…

```
func main() {
	var phrase = "Il était"

	fmt.Printf("len(phrase) = %v\n", len(phrase))                    // 9 octets !!!
}
```

Comparaison de chaînes

- Les chaînes, comme les nombres (et contrairement aux tranches), peuvent être comparées avec les opérateurs de comparaison classiques.

- L'ordre est celui du dictionnaire. Attention, les majuscules sont considérées comme inférieures aux minuscules et les lettres accentuées sont considérées comme supérieures aux non accentuées (ordre du code UTF-8)

```
"Bla" < "blabla"                                    // true
"était" > "fier"                                    // true
nom = "Martin"
if nom == "Martin" {
   fmt.Println("ok")                                // ok
}
```

Saisie de chaîne quelconque

- On a déjà vu qu'on pouvait saisir une chaîne au clavier avec la fonction **fmt.Scan(&chaine)**, *à condition que l'on ne saisisse pas d'espace*.

- Or, la saisie d'une phrase complète, voire d'un nom composé, exige de pouvoir traiter le cas des phrases contenant des espaces.

- Pour ce faire, le plus simple consiste à utiliser un appel **bufio.NewScanner(os.Stdin)** qui, comme on peut le deviner, crée un **Scanner** qui lira ses données sur **os.Stdin** (c'est-à-dire le clavier).

- Pour lancer ce scanner, on utilisera sa fonction **Scan** et, pour récupérer le texte saisi jusqu'au retour à la ligne, on utilisera sa fonction **Text**.

Saisie de chaîne : exemple

```
package main
import (
    "fmt"
    "bufio"       // bufio = Buffered IO (entrées sorties avec tampon)
)

func main() {
   clavier := bufio.NewScanner(os.Stdin)      // On lira sur le clavier

   fmt.Print("Entrez une phrase et terminez avec Entrée : ")
   clavier.Scan()                                    // Effectue une saisie
   phrase := clavier.Text()                          // Récupère le texte saisi

   fmt.Printf("Vous avez saisi : %v\n", phrase)
}
```

Saisie de chaînes : création d'une fonction input()

- Comme on l'a vu en TD, si l'on doit saisir plusieurs fois une chaîne, il peut être souhaitable d'encapsuler tout ce traitement dans une fonction qui imitera la fonction **input("message")** de Python :

```
func input(mess string) string {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Print(mess)
    scanner.Scan()
    return scanner.Text()
}
```

```
func main() {
    phrase := input("Entrez une phrase : ")
    ...
}
```

Autres fonctions utiles sur les chaînes : le module strings

- Le module **strings** (qu'il faut importer) fournit un grand nombre de fonctions utiles sur les chaînes : aidez-vous de l'aide que vous fournit la commande **go doc**.

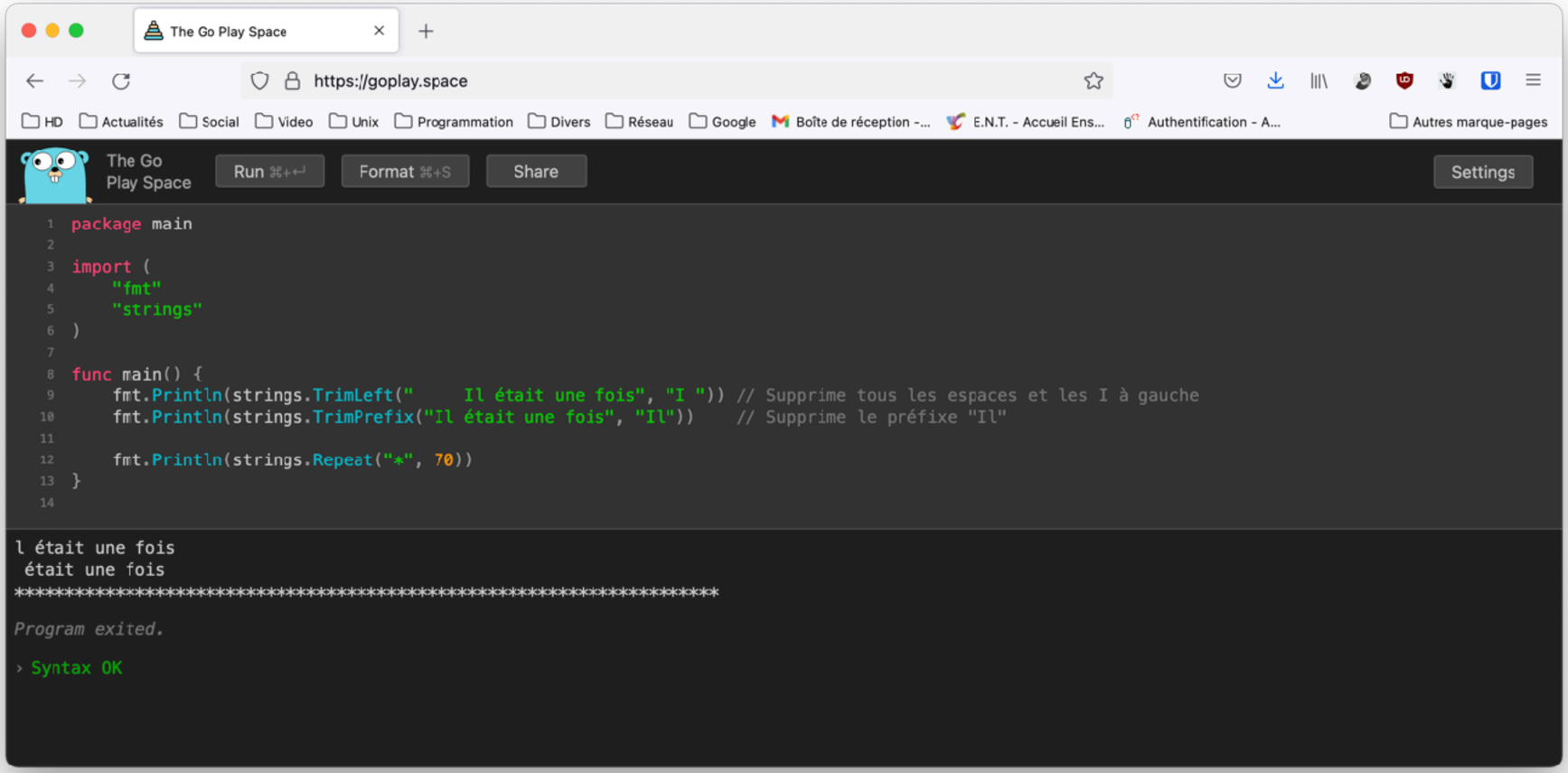
- Comment lire ce qu'affiche l'aide ?

  - la notation **func TrimLeft(s, cutset string) string**, par exemple, signifie que :

    - **TrimLeft** prend deux chaînes en paramètre et renvoie une chaîne (n'oubliez pas qu'on ne peut pas modifier une chaîne en Go…)

    - L'explication précise que la fonction renvoie une copie de **s** où tous les caractères présents dans **cutset** au début de **s** ont été ôtés (notez que l'explication emploie le terme de "Unicode code point" et non de "caractère" pour éviter les ambigüités). Elle renvoie également à la fonction **TrimPrefix**, qui permet de supprimer un préfixe complet…

Autres fonctions utiles : essais avec le Go Play Space



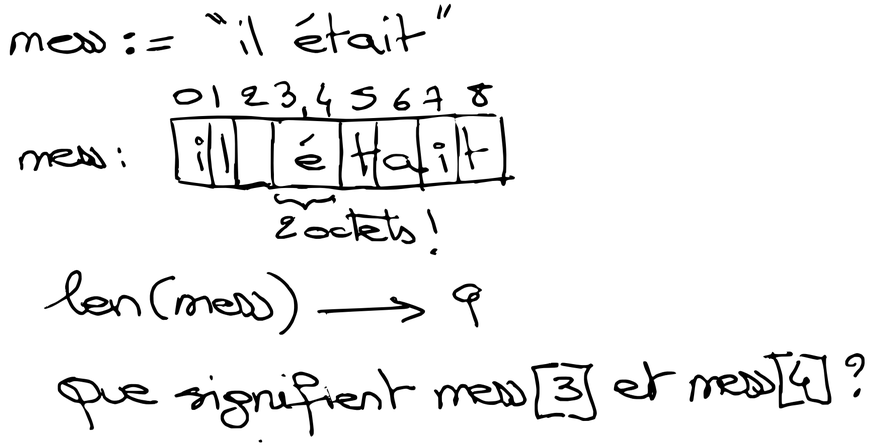
Voir [ce lien]

Au lieu de simplement afficher les résultats des appels de fonctions, on aurait pu les utiliser dans des expressions ou les affecter à des variables de type string

Chaînes Go et tranches de runes

- Nous avons vu que les chaînes Go (**string**) sont considérées comme des **tranches d'octets** particulières : la fonction **len** renvoie donc le nombre d'octets et les indices permettent d'accéder (***en lecture seule***) aux différents octets.

- Le problème est que les caractères sont codés en UTF8, et que leurs codes varient entre 1 et 4 octets… Donc, tout va très bien si la chaîne ne contient que des caractères codés sur un octet, ce qui est le cas des caractères latins non accentués (codés en ASCII)

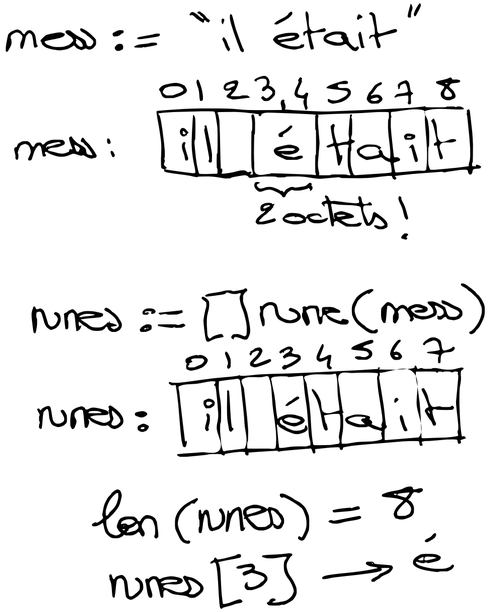


Chaînes Go et tranches de runes

- Dans la vie de tous les jours, on a plutôt tendance à considérer une chaîne comme une **suite de caractères**. En Go, les caractères sont de type **rune** (un alias de **int32**).

- L'idée consiste donc à convertir une chaine Go (**string**) en tranche de runes (**[]rune**).

- Inversement, on peut obtenir une **string** à partir d'une tranche de **rune** en faisant **string(tranche)**



Chaînes Go et tranches de runes

- Cette conversion en tranche présente plusieurs avantages Voir ce code :
  - On peut correctement gérer les différents caractères et les indices correspondent bien à des positions de caractères, pas d'octets. La longueur de la tranche, telle qu'est renvoyée par **len()**, correspond bien au nombre de caractères, pas au nombre d'octets.
  - On peut modifier chaque caractère (alors que le type **string** n'est pas modifiable).
  - On peut aisément passer du type **string** au type **[]rune** (et réciproquement lorsqu'on a besoin des fonctionnalités des chaînes Go et des fonctions du module **strings**…).

Parcours de chaînes

- On retrouve la même distinction que pour les tranches : **parcours partiels** et **parcours complets**, avec les mêmes solutions algorithmiques ("**tant que**" et "**pour chaque**...") mais, ici, il faut en plus gérer le fait que les indices des chaînes sont des **indices d'octets, pas de caractères**...

- En pratique, on n'aura quasiment jamais besoin de parcourir une chaîne octet par octet (sauf si l'on sait, par exemple, que cette chaîne n'est formée que de chiffres) : on n'utilisera donc quasiment jamais les indices d'une chaîne pour la parcourir...

Parcours complet d'une chaîne **rune par rune**

- La forme **for i, car := range chaine {...}** qu'on a déjà vue avec les tranches permet de gérer simplement le parcours complet des chaînes, *caractère par caractère* (et non octet par octet) :
  - à chaque itération, **car** (qui sera alors de type **rune** et non **byte)** contiendra le caractère courant, qu'il soit accentué ou non.
  - à chaque itération, **i** contiendra le numéro du **premier octet** du caractère dans la chaîne (peu utilisé, en pratique).
  - Voir ce lien (notez les valeurs de **i** affichées par la deuxième boucle)

Parcours complet et partiel d'une chaîne rune par rune

- On rappelle qu'**une chaîne n'est pas modifiable** : si l'on veut un parcours complet avec possibilité de modification, il faudra donc passer par une tranche de runes et appliquer ce que l'on a vu précédemment pour les parcours de tranches.

- Pour les parcours partiels, on a vu qu'on doit utiliser un **tant que** avec des indices et donc, là aussi, se ramener à une tranche de runes, sauf cas particuliers.

Parcours de chaînes : résumé

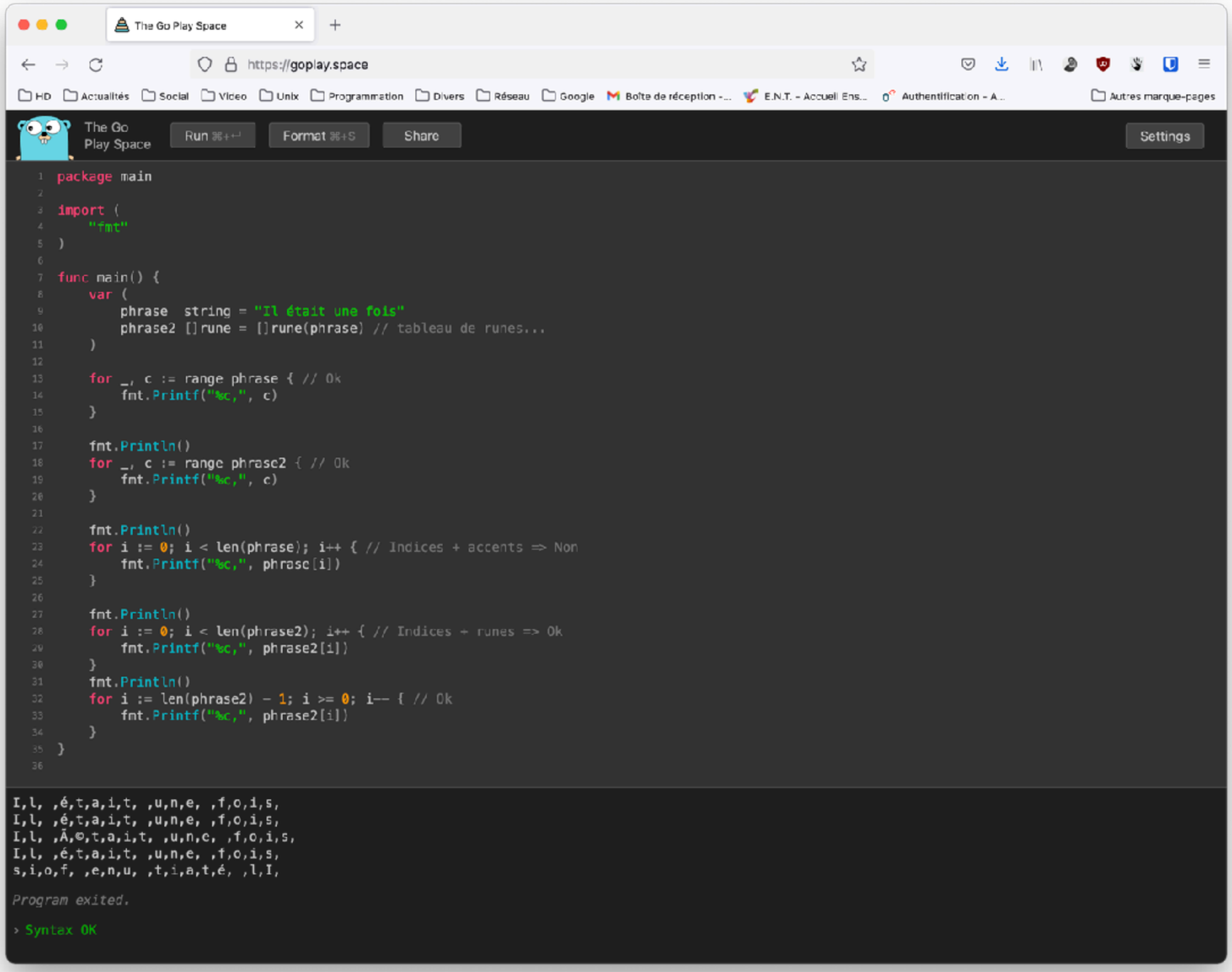
- Pour un *parcours complet de gauche à droite*, **toujours** utiliser la forme **for i, c := range chaine {…}**

- Pour un *parcours partiel* sur une chaîne, passer par une **tranche de runes**, puis utiliser un **for** classique (**Tant Que**) avec un indice correctement initialisé. Attention à tester en premier la fin de la séquence si on la parcourt de gauche à droite (et le début de la séquence si on la parcourt de droite à gauche).

- Faites les feuilles d'exercices pour vous entraîner !

Parcours de chaînes

[Voir ce lien]



Sous-chaînes

- Une sous-chaîne est simplement une portion de chaîne comprise entre deux indices, comme une tranche classique. Mais, encore une fois, **on ne peut pas se fier aux indices du type string** (sauf si l'on est absolument sûr qu'elle ne contiendra que des caractères ASCII -- un code de sécurité sociale ou un code postal, par exemple) : **si l'on veut créer des sous-chaînes, il faut d'abord passer par une tranche de runes**.

- Les chaînes n'étant pas modifiables, les tranches ne le sont pas non plus : une tranche ne fait **qu'extraire** une partie d'une chaîne (on n'a donc pas le problème des tranches de tableaux qui peuvent modifier le tableau initial).

Sous-chaînes : illustration

Soit un numéro INSEE : "1800531120166"

De ce numéro, on peut déduire :

- le sexe : 1 => homme
- l'année de naissance : 80 => 1980
- le mois de naissance : 05 => Mai
- le département de naissance : 31 => Haute-Garonne
- la commune de naissance : 120 => N° de commune en Haute-Garonne
- l'ordre dans le mois de naissance : 166e

Sous-chaînes : illustration avec une chaîne **non accentuée**

```
insee := "1800531120166"    // Uniquement des caractères ASCII...

var sexe byte               // Pour contenir soit le caractère H, soit le caractère F

if insee[0] == '1' {        // insee[0] est de type byte mais '1' est un entier universel
   sexe = 'M'
} else {
   sexe = 'F'
}

anneeNaissance, _  := strconv.Atoi(insee[1:3])          // 80
moisNaissance, _   := strconv.Atoi(insee[3:5])          // 5
deptNaissance      := insee[5:7]                        // "31"
comNaissance       := insee[7:10]                       // "120"
ordreNaissance, _  := strconv.Atoi(insee[10:13] )       //  ou insee[10:]   166
```

Sous-chaînes : autre illustration avec une chaîne **accentuée**

```
chaine := "Il était une fois un palais des congrès abandonné..."  // chaîne générale...

verbe := chaine[3:8]     // raté !!!
fmt.Println(verbe)       // étai

runes := []rune(chaine)      // on passe en tranche de runes pour gérer les indices
verbeOk := runes[3:8]     // ok !!!
verbe = string(verbeOk)    // on peut repasser en chaîne
fmt.Println(verbeOk)       // était
```

Voir ce code

Sous-chaînes : une autre illustration

Le package **strings** fournit une fonction **strings.Index** qui renvoie l'indice de début d'une sous-chaîne dans une chaîne, mais c'est **un indice d'octet** !
L'exemple suivant renverra donc 10 au lieu de 9...

```
fmt.Printf("L'indice de début de 'une' dans 'Il était une fois' = %v\n", strings.Index("Il était une fois", "une"))
```

Voir ce code

Sous-chaînes : une autre illustration

Écrivons une fonction équivalente, mais qui renvoie l'indice du **premier caractère** de la sous-chaîne :

```
// find renvoie l'indice du caractère de début de schaine dans chaine (-1 si non trouvé)
func find(chaine, schaine string) int {
    if len(chaine) == 0 ||  len(schaine) == 0 {     // Gestion d'un mauvais appel...
        return -1
    }
    chaineR, schaineR := []rune(chaine), []rune(schaine)    // On passe par des tranches de runes
    for i := 0; i < len(chaineR)-len(schaineR); i++ {
        if string(chaineR[i:i+len(schaineR)]) == schaine {   // == ne marche pas avec les tranches...
            return i
        }
    }
    return -1
}
```

Voir ce code