Go to *Python*

Nan Messe

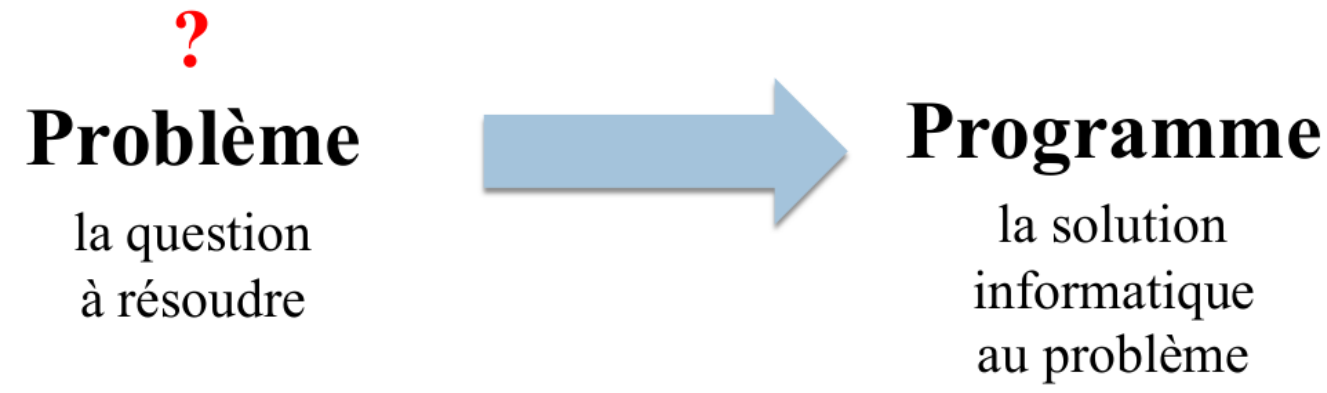
Université Toulouse Jean Jaurès

Sommaire

Sommaire

Rappel des notions

Algorithme

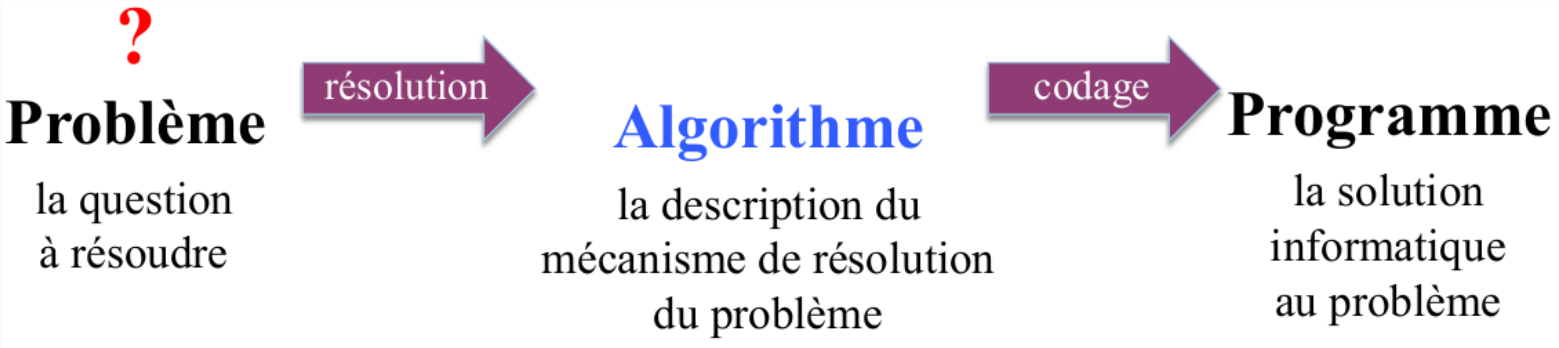


Avant d'écrire un programme :

- reformuler le problème : énoncé précis, sans ambiguïté,
- réfléchir à la (ou les) solution(s),
- écrire la succession d'actions qui fourniront la solution,
- formaliser : opérations logiques, mathématiques, tests, etc..

$\Longrightarrow$ **algorithme**

Algorithme vs. Programme



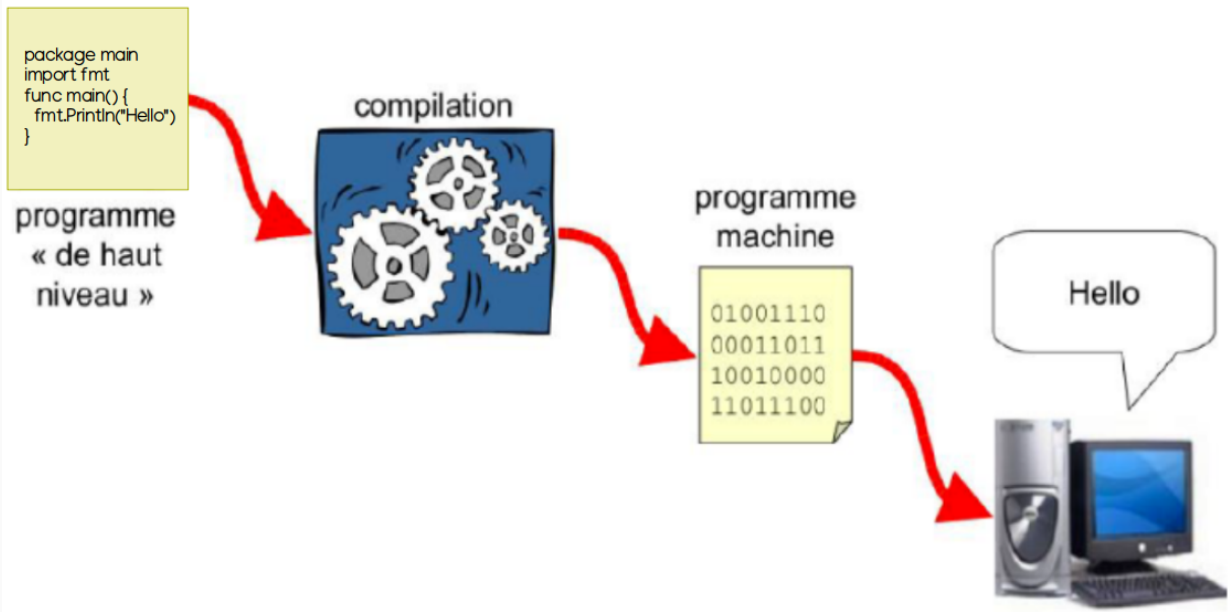
On distingue l'algorithme du programme :

- Un algorithme est une description finie d'un enchaînement d'opérations ou d'instructions élémentaires, organisée dans le but de résoudre un problème ;
- Un programme est une description finie écrite dans un langage compris par la machine : c'est un cas particulier d'algorithme écrit dans un langage de programmation donné.

Compilation vs. Interprétation

Compilateur : un programme qui transforme un programme d'un langage source en un programme d'un langage cible.

- Le langage source est de haut niveau d'abstraction, et facilement compréhensible par l'humain.
- Le langage cible est de plus bas niveau, par exemple un langage d'assemblage ou langage machine, afin de créer un programme exécutable par une machine.



Compilation vs. Interprétation

- L'**interprétation** est un mode d'exécution de programmes dans lequel l'exécution est effectuée au fur et à mesure de l'analyse du code source du programme.
- L'interprétation repose sur l'exécution dynamique du programme par un autre programme (l'**interprète** ou l'**interpréteur**), plutôt que sur sa conversion en un autre langage (par exemple le langage machine) ; elle évite la séparation du temps de conversion et du temps d'exécution, qui sont simultanés.

Compilation vs. Interprétation

Un **compilateur** procède par étapes :

1. analyse lexicale (mots du langage),
2. analyse syntaxique (grammaire),
3. analyse sémantique : les instructions ont-elles un sens ?
4. génération de code.

Le cycle d'un **interpréteur** est le suivant :

1. lire et analyser une instruction (ou expression) ;
2. si l'instruction est syntaxiquement correcte, l'exécuter (ou évaluer l'expression) ;
3. passer à l'instruction suivante.

Script vs. Compilé

On différencie un programme dit script, d'un programme dit compilé :

- un **programme script** est exécuté à partir du fichier source via un interprète de script ;
- un **programme compilé** est exécuté à partir d'un bloc en langage machine issu de la traduction du fichier source.

Introduction

Introduction : *Go*

- *Go* est un langage compilé, qui nécessite de traduire un code source (extension .go) en un code binaire, directement exécutable sur la machine :

```
> go build monfic.go      # compile monfic.go pour produire le fichier exécutable
> go run monfic.go        # compile monfic.go, exécute le programme résultant, puis le supprime
```

- La version binaire, produite par *go build*, peut ensuite s'exécuter sur n'importe quelle autre machine ayant la même architecture et le même système d'exploitation. Cette nouvelle machine n'a pas besoin de compilateur Go ou de quoi que ce soit d'autre.

Remarque :
L'avantage de la compilation est aussi qu'elle permet de détecter des erreurs **avant** de pouvoir exécuter le programme et que le programme produit est bien plus rapide qu'un script interprété.

Introduction : *Python*

- *Python* est un langage interprété qui nécessite la présence d'un interpréteur Python sur la machine pour exécuter un script (extension .py).

```
> python monfig.py      #  exécute le script monfig.
```

- Il faut donc installer un interpréteur Python sur chaque machine ayant besoin d'exécuter le script.

Remarque :
Les erreurs de programmation seront découvertes à mesure que le script s'exécute, ce qui complique la mise au point.

Structure d'un programme

Structure d'un programme *Go*

- Tout programme *Go* comporte au moins une fonction `main()`, qui doit faire partie du paquetage *main* et qui sera automatiquement appelée au lancement du programme :

```
package main

import (                                  // liste des modules nécessaires au programme
    "module1"
    "module2"
    ...
)

func main() {
    ...
}
```

- Exemple :

```
package main
import "fmt"                                    // Si on importe qu'un module, pas besoin de (...)

func main() {
   fmt.Println("Vive le langage Go !")         // La fonction Println fait partie du module fmt
}
```

Structure d'un programme *Python*

- Il n'y a pas de « fonction principale » comme en *Go*, mais rien n'empêche de se forcer à en créer une (qu'il faudra alors appeler explicitement) :

```
# importation des modules nécessaires au programme}
import module1, module2, ...

def main():
    ...

main()  # Pour appeler explicitement le code du programme principal}
```

- Exemple :

```
# Ici, on n'a besoin d'aucun module...}
def main():
    print("Vive le langage Python")

main()
```

Remarque :
En réalité, tout le monde écrirait simplement `print("Vive le langage Python")`...

Commentaires

- En *Go* :

```
// commentaire sur une seule ligne

/* commentaire sur
   plusieurs lignes
*/
```

- En *Python* :

```
# commentaire sur une seule ligne

"""commentaire sur
   plusieurs lignes
"""
```

Types et déclarations

Types et déclarations en *Go*

- Toute variable/constante/paramètre/résultat d'une fonction doit avoir un type (*int*, *float*, *string*, *[]int*, etc.) qui lui est affecté via une déclaration (**typage statique**). Pour les variables (pas les constantes), ce type peut être inféré si on affecte une valeur dès la déclaration (voir :=).
- Les valeurs de types différents (y compris les nombres) sont incompatibles entre elles et il faut passer par des conversions explicites.
- Les noms des variables (et des fonctions) sont de la forme *nomVariable* ou *nomFonction*.
- Une variable déclarée doit être utilisée.

Types et déclarations en *Python*

- Le type d'une variable/paramètre/résultat d'une fonction dépend de la valeur qui lui est affectée (**typage dynamique**), mais *Python* utilise les types *int*, *float*, *bool*, *str*, et *list*.
- Les nombres entiers et flottants sont compatibles entre eux. Pour convertir un flottant ou une chaîne numérique en entier, on utilise l'opération *int(...)*. L'opération *float(...)* permet d'obtenir un flottant à partir d'un entier ou d'une chaîne numérique. Ces opérations échouent si elles n'arrivent pas à convertir la valeur dans le type indiqué.
- Il n'y a pas de déclaration et il n'existe pas d'équivalent des constantes en *Python* : une variable existe dès qu'elle est affectée pour la première fois et elle prend le type de la valeur qui lui est affectée.
- Les noms des variables (et des fonctions) sont de la forme *nom_variable* ou *nom_fonction*.

Délimitation des blocs d'instructions

- En *Go*, les blocs sont définis entre accolades (ce qui n'empêche pas d'indenter).
- En *Python*, les blocs sont définis par leur indentation (ne pas mélanger les espaces et les tabulations). Chaque ensemble d'expressions indentées représente un bloc d'instructions.

Opérateurs

Opérateurs arithmétiques

- Les opérateurs arithmétiques +, -, *, % et / de *Go* se retrouvent à l'identique en *Python*, mais *Python* a deux opérateurs de division :
  - / effectue **toujours** une division flottante et produit **toujours** un résultat flottant, quels que soient les types des opérandes (piège courant...)
  - // effectue une division entière et produit le quotient, dont le type dépend des opérandes.
- *Python* fournit également l'opérateur d'élévation à la puissance **.
- *Python* utilise, comme *Go*, les opérateurs combinés à l'affectation (+=, par exemple). En revanche, les opérateurs ++ et -- n'existent pas en *Python*.

Opérateurs sur les chaînes et opérateurs booléens

- Opérateurs sur les chaînes :
  - Comme en *Go*, l'opérateur de concaténation des chaînes en *Python* est l'opérateur +.
  - *Python* fournit également l'opérateur * qui permet de répéter une chaîne un certain nombre de fois (identique au *strings.Repeat* de *Go*).
- Opérateurs booléens :
  - Les valeurs booléennes *true* et *false* de *Go* sont représentées par *True* et *False* en *Python* (notez la majuscule).
  - Les opérateurs *!*, *&&* et || de *Go* sont représentés, respectivement, par les opérateurs *not*, *and* et *or* en *Python*. Comme eux, ils utilisent une évaluation en court-circuit.

Priorité des opérateurs en *Python*

1. ()
2. Opérateur binaire : **
3. Opérateurs unaires : +, - (ex. -2)
4. Opérateurs binaires :
   - 4.1 *, /, //, %
   - 4.2 + et - exécutés de gauche à droite, comme cela apparaît dans l'expression
   - 4.3 <, >, <=, >=
   - 4.4 ==, !=
5. not
6. and
7. or

Structures de contrôle

Structures de choix en *Go*

```
if cond1 {                    // le { doit être sur la même ligne que if
    ...
} else if cond2 {             // facultatif mais les else doivent être sur la même ligne que }
    ...
} ...
} else {                      // facultatif
    ...
}

switch expr {
    case val1:                // le bloc de val1 commence après le : et se termine avant le case suivant
       ...
    case val2, val3, val4:
       ...
    ...
    default:                  // facultatif
       ...
}

switch {
    case cond1:
       ...
    case cond2:
       ...
    ...
    default:                  // facultatif
       ...
}
```

Structures de choix en *Python*

```
if cond1:
    ...
elif cond2:                    # facultatif
    ...
...
else:                          # facultatif
    ...

match expr:                    # uniquement à partir de Python 3.10
    case val1:
        ...
    case val2:
        ...
    case (val3 | val4):
        ...
    case val5 if val5 < 0:
        ...
    case _:
        ...
```

Boucles en *Go*

```
for cond {
    ...
}

for cpt := deb; cpt < fin; cpt++ {  // cpt est LOCAL au bloc
    ...
}

for cpt := deb; cpt < fin; cpt += pas {
    ...
}

for i, elt := range liste {
    ...
}
```

Boucles en *Python*

- À la différence de *Go*, *Python* dispose du mot-clé *while* pour représenter la boucle *tant que* :

```
while cond:
    ...

for cpt in range(deb, fin):       # s'arrête à fin - 1. cpt n'est PAS locale au bloc
    ...

for cpt in range(deb, fin, pas):
    ...

for elt in liste:
    ...

for i, elt in enumerate(liste):
    ...
```

Itérations en *Python* : boucles while et for

Boucle while

```
n = 0
while (n < 5):
    print(n)
    n = n + 1
```

Boucle for

```
for n in range (5):
    print(n)
```

Entrées/Sorties

Entrées/Sorties en *Go*

- On saisit avec *fmt.Scan* et on affiche avec *fmt.Print*, *fmt.Println* ou *fmt.Printf*.
- Pour saisir des chaines avec des espaces, on utilise les fonctions du module *bufio* (cf. cours de L1)

```
...
var (
    entier int
    chaineSansEspace string
)

fmt.Print("Entrez un entier : ")              // Affichage sans retour à la ligne
fmt.Scan(&entier)

fmt.Print("Entrez une chaîne sans espace : ")
fmt.Scan(&chaineSansEspace)

fmt.Println("Voici un affichage avec retour à la ligne")
fmt.Printf("Vous avez saisi %v et %v\n", entier, chaineSansEspace)
```

Entrées/Sorties en *Python*

- On saisit avec *input* et on affiche avec *print*.
- Il faut éventuellement convertir la chaîne renvoyée par *input*.

```
...
entier = int(input("Entrez un entier :"))    # input renvoie toujours une chaîne de caractères

chaine = input("Entrez une chaine : ")

print("Affichage avec retour à la ligne")
print("Affichage sans retour à la ligne", end="")

print("Vous avez saisi", entier, "et " + chaine)
print(f"Vous avez saisi {entier} et {chaine}")
```

Sous-programmes et portées des variables

Sous-programmes en *Go*

```go
// nomFonction prend deux paramètres de types différents et renvoie une valeur de type typeResultat
func nomFonction(param1 type1, param2 type2) typeResultat {
    var resultat typeResultat
    ...
    return resultat
}

// nomFonction prend deux paramètres de même type et renvoie une valeur de type typeResultat
func nomFonction(param1, param2 type1) typeResultat {
    var resultat typeResultat
    ...
    return resultat
}

// nomFonction ne prend aucun paramètre et renvoie une valeur de type typeResultat
func nomFonction() typeResultat {
    var resultat typeResultat
    ...
    return resultat
}

// nomProc prend deux paramètres de types différents et ne renvoie aucun résultat (procédure)
func nomProc(param1 type1, param2 type2) {
    ...
}

// nomProc ne prend aucun paramètre et ne renvoie aucun résultat (procédure)
func nomProc() {
    ...
}
```

Sous-programmes en *Python*

```python
def nom_fonction(param1, param2):
    """Fonction prenant deux paramètres"""
    ...
    return resultat

def nom_fonction():
    """Fonction ne prenant aucun paramètre"""
    ...
    return resultat

def nom_proc(param1, param2):
    """Procédure prenant deux paramètres"""
    ...
    # Python retourne la valeur None, si pas de return

def nom_proc():
    """Procédure ne prenant aucun paramètre"""
    ...

"""Plus loin, vous appelez le sous-programme en utilisant son nom et des valeurs pour les paramètres"""
nom_fonction(3,5)
```

- En *Go*, le commentaire présentant une fonction est placé **avant** la définition de la fonction et doit commencer par le nom de la fonction.
- En *Python*, le commentaire présentant une fonction est placé **à l'intérieur** de la fonction, généralement entre """ et """.

Spécification des sous-programmes

Contrat : entre celui qui code (implémente) le sous-programme et l'utilisateur qui va l'utiliser

Un **cartouche** :

- Auteur : le nom de celui qui a conçu le code
- Rôle de la fonction : ce qu'elle réalise
- Entrées : ce que l'on donne en entrée de la fonction : paramètres ou arguments
- Résultats : ce que modifie la fonction

```
def est_pair(i):
    """
    Auteur : Pierre Dupont
    Rôle : la fonction est_pair(i) retourne vrai si i est pair et faux sinon
    Entrées : i, entier
    Résultats : retourne True si i est pair et False sinon
    """

    return i %2 == 0
```

Vers une bonne programmation

Un programme sans fonctions :

- Facile pour des problèmes à petite échelle, complexe pour des problèmes plus importants
- Difficulté de garder une trace des détails
- Comment sait-on que la bonne info est fournie à la bonne partie du code ?

Pourquoi utiliser des fonctions ?

- Plus de code n'est pas forcément une bonne chose
- Ce sont des morceaux de code réutilisables
- On mesure les bons programmeurs par la quantité de fonctionnalité dans leur code
- Les fonctions ne sont pas exécutées dans un programme sauf si elles sont appelées dans ce programme
- Mécanisme pour réaliser la **décomposition** et l'**abstraction**

Vers une bonne programmation : Décomposition

Décomposition : Diviser le problème en plusieurs morceaux autonomes

- Division du code en **modules**
  - **autonomes**
  - utilisés pour **subdiviser** le code
  - **réutilisables**
  - permettent d'**organiser** (structurer) le code
  - gardent le code **cohérent**
- Dans ce cours, la décomposition est réalisée au moyen de **fonctions**
- Plus tard, décomposition à l'aide de **classes**

Vers une bonne programmation : Abstraction

Abstraction : Supprimer les détails de la méthode, et ramener le calcul à l'utilisation de cette méthode avec des conditions différentes.

- Penser le morceau de code comme une **boîte noire**
  - On ne peut pas voir les détails
  - On n'a pas besoin de voir les détails
  - On ne veut pas voir les détails
  - On cache les détails fastidieux du code
- On réalise l'abstraction avec la spécification de fonctions (ou docstrings)

Portées des variables/constantes en *Go*

- Une variable ne peut pas être définie en dehors d'une fonction : elle est toujours **locale** à une fonction.
- Une constante peut être définie en dehors de toute fonction : en ce cas, elle est **globale** au fichier.
- Ceci implique qu'une fonction ne peut « voir » que ses paramètres, ses variables/constantes locales et les éventuelles constantes globales.
- La portée d'une variable s'étend de sa déclaration à la fin du bloc dans lequel elle a été définie.

Portée des variables/constantes en *Python*

- Une variable définie dans une fonction est **locale** à cette fonction (comme en *Go*).
- Une variable définie en dehors d'une fonction est accessible **en lecture** en tout point du fichier source (portée lexicale).
- Ceci implique qu'une fonction « voit » ses paramètres, ses variables locales et toutes les variables définies en dehors des fonctions.
- Cela peut poser problème puisque Python n'a pas de fonction principale comme *Go* et les variables du « programme principal » sont donc visibles dans toutes les fonctions du fichier : voir cet exemple.
- C'est la raison pour laquelle il est préférable de simuler une fonction `main` : voir cet exemple (qui ne marche pas... et tant mieux !).

Que ce soit en Go ou en Python, les fonctions ne peuvent pas modifier leurs paramètres (passage par copie), sauf si ces paramètres sont des pointeurs (uniquement en Go) ou des tableaux/tranches/listes.

Chaînes de caractères

Chaînes de caractères en *Go*

- Une chaîne est de type *string* et n'est pas modifiable : il faut passer par une tranche de *rune* pour la modifier.
- L'appel *len(ch)* renvoie le nombre d'octets contenus dans la chaîne *ch*, ce qui ne correspond pas nécessairement au nombre de caractères.
- Les caractères sont représentés par le type *rune* (un entier contenant le code UTF-8 du caractère). Si l'on veut afficher un caractère, il faut utiliser le format *%c* de *fmt.Printf*.
- Un littéral chaîne se note entre guillemets (*"Bonjour"*), un littéral caractère se note entre apostrophes (*'a'*).
- L'opérateur *+* permet de concaténer deux chaînes et le module *strings* propose un grand nombre de fonctions utilitaires sur les chaînes. Voir ce lien.
- On peut parcourir la chaîne *ch* caractère par caractère (rune par rune) en utilisant une boucle *for _ , car := range ch {...}*.

Chaînes de caractères en *Python*

- Une chaîne est de type *str* et n'est pas modifiable : il faut passer par une liste (de type *list*) pour la modifier (voir plus loin).
- L'appel *len(ch)* renvoie le nombre de caractères contenus dans la chaîne *ch*.
- Il n'existe pas de « type caractère » : un caractère est représenté par une chaîne d'un seul caractère.
- Un littéral chaîne se note entre guillemets (*"Bonjour"*) ou entre apostrophes (*'Bonjour'*).
- L'opérateur *+* permet de concaténer deux chaînes, l'opérateur *in* permet de tester qu'une sous-chaîne appartient à une chaîne et le type *str* fournit un grand nombre de méthodes utilitaires sur les chaînes. Voir ce lien.
- On peut parcourir la chaîne *ch* caractère par caractère en utilisant une boucle *for car in ch : ...*
- Comme en *Go*, on peut utiliser des tranches de chaînes pour extraire des sous-chaines.

Tranches et listes

Tranches en *Go* (1/3)

- Une tranche est de type *[]typeElt*, où *typeElt* est le type des éléments : cette tranche ne pourra donc contenir que des éléments de ce type.
- Pour créer une tranche, on part d'une tranche vide (ou non) et on lui ajoute ensuite des éléments avec *append* ou, si on sait d'avance la taille qu'elle devra avoir, on la crée avec *make*, ce qui permet ensuite d'utiliser des indices pour y placer les éléments.
- Pour effectuer un parcours total d'une tranche, on utilise la forme *range* de la boucle *for*.

```
xs := []int{}                              // xs est une tranche d'entiers vide
ys := []string{"Joe", "William"}           // ys est une tranche de deux chaînes
zs := make([]float64, 5)                   // zs est une tranche de 5 flottants
fmt.Printf("%v %v %v\n", len(xs), len(ys), len(zs))   // 0 2 5

xs = append(xs, 10, 100)                   // xs contient maintenant 10 et 100
fmt.Println(len(xs))                       // 2

ys = append(ys, "Jack")                    // ys contient "Joe", "William" et "Jack"
fmt.Println(len(xs))                       // 3

for i, _ := range zs {                     // zs contient 0.0, 1.0, 4.0, 9.0 et 16.0
    zs[i] = float64(i * i)
}
fmt.Println(zs)                            // [0 1 4 9 16]
```

Tranches en *Go* (2/3)

- Les fonctions d'affichage savent afficher une tranche.
- La notation *[deb :fin]* permet d'extraire une sous-tranche (mais **attention**...)

```
zs1, zs2 := zs[0:len(zs)/2], zs[len(zs)/2:len(zs)]
fmt.Printf("%v et %v\n", zs1, zs2)                  // [0 1] et [4 9 16]

zs11, zs21 := zs[:len(zs)/2], zs[len(zs)/2:]        // idem

zs2[1] = 42.0                                       // zs a été modifié aussi !!!
fmt.Printf("%v et %v\n", zs, zs2)                   // [0 1 4 42 16] et [4 42 16]
```

Tranches en *Go* (3/3)

- Une tranche passée en paramètre à une fonction pourra être modifiée par la fonction.

```
func enMaj(tab []string) {
    for i, _ := range tab {
        tab[i] = strings.ToUpper(tab[i])
    }
}
...
enMaj(ys)                   // ys a été modifiée par enMaj
fmt.Println(ys)             // [JOE WILLIAM JACK]
```

- Voir ce lien

Listes en *Python* (1/5)

- Le type *list* permet de regrouper plusieurs valeurs qui seront ensuite accessibles via leurs indices, comme pour les tranches *Go*.
- Bien qu'une liste *Python* puisse contenir des valeurs de types différents, cette pratique est déconseillée : nous n'utiliserons dans ce cours que des listes homogènes, où les éléments seront tous de même type.
- Pour créer une liste, on part d'une liste vide (ou non) et on lui ajoute ensuite des éléments avec *append* (comme en *Go*, à la syntaxe près...).
- On peut concaténer deux listes avec l'opérateur *+*.
- Il n'y a pas d'équivalent au *make* de *Go*, mais on peut initialiser une liste avec un certain nombre d'éléments grâce à l'opérateur ***.
- Comme en *Go* (et la plupart des langages), on accède à un élément d'une liste par son indice et la notation entre crochet. Le premier indice est 0.
- On peut également utiliser des indices négatifs (à partir de -1) pour parcourir la liste de droite à gauche.

Listes en *Python* (2/5)

- Une **liste en extension** est définie par la liste explicite de ses composants (cf. exemples précédents).
- Une **liste en intension** est définie par les propriétés de ses éléments. Sa notation est proche d'une définition mathématique.

```
liste1 = list(range(100)) # liste1 définie en extension
liste_pairs = [x for x in liste1 if x % 2 == 0] # liste_pairs définie en intension
liste2 = [x for x in range(3, 100) if x % 2 == 0 and x ** 2 > 1000] #liste2 définie en intension
```

Listes en *Python* (3/5)

- Pour effectuer un parcours total d'une liste, on utilise une boucle *for* avec l'opérateur d'appartenance *in* pour n'accéder qu'aux éléments, ou avec *enumerate* pour récupérer à la fois les indices et les éléments (comme le *range* de *Go*)
- L'opérateur *in* peut également être utilisé avec un *while* ou un *if* pour tester l'appartenance d'un élément à une liste.

```python
xs = []                     # xs est une liste vide
ys = ["Joe", "William"]     # ys est une liste de deux chaînes
zs = [0.0] * 5              # zs est une liste de 5 flottants
print(f"{len(xs)}, {len(ys)}, {len(zs)}")   # 0, 2, 5
xs.append(10)
xs += [100]                 # xs contient maintenant 10 et 100

ys.append("Jack")           # ys contient "Joe", "William" et "Jack"

for i in range(len(zs)):
    zs[i] = float(i * i)

print(zs)                   # [0.0, 1.0, 4.0, 9.0, 16.0]

liste = list(ys[0])         # liste est modifiable (mutable) : elle contient maintenant ['J', 'o', 'e']
liste[0] = "j"              # remplace l'ancienne valeur de liste[0] par "j"
ys[0] = "".join(liste)
print(ys)                   # ['joe', 'William', 'Jack']
```

Listes en *Python* (4/5)

- La fonction *print* permet d'afficher une liste.
- La notation *[deb :fin]* permet d'extraire une sous-liste distincte de la liste initiale (différence avec *Go*)

```
zs1, zs2 = zs[0:len(zs)//2], zs[len(zs)//2:len(zs)]
print(f"{zs1} et {zs2}")    # [0.0, 1.0] et [4.0, 9.0, 16.0]

zs11, zs21 = zs[:len(zs)//2], zs[len(zs)//2:]
print(f"{zs11} et {zs21}")  # idem

zs2[1] = 42.0               # zs n'a pas été modifiée
print(f"{zs} et {zs2}") #tester vous-même
```

Attention !

- Notez qu'en *Python*, il faut utiliser l'opérateur // pour réaliser une division entière.
- L'opérateur / réalise toujours une division réelle.

Listes en *Python* (5/5)

- Comme en *Go*, une liste passée en paramètre à une fonction pourra être modifiée par la fonction.

```
def en_maj(chaines):
    for i in range(len(chaines)):
        chaines[i] = chaines[i].upper()

...
en_maj(ys)                  #  ys a été modifiée par en_maj...
print(ys)                   # ['JOE', 'WILLIAM', 'JACK']}
```

Lien vers l'exemple complet.

Principaux pièges de *Python*

Principaux pièges de *Python*

- Attention à l'indentation des blocs : ne mélangez pas les espaces (le pep8 en recommande 4) avec les tabulations. Faites notamment attention quand vous copiez/collez du code provenant d'une page web.
  Le pep8 recommande d'utiliser des espaces plutôt que des tabulations, mais vous pouvez configurer votre éditeur pour que la touche `tab` produise 4 espaces (c'est ce que font automatiquement les extensions *Python*).
- Attention à la division : `//` renvoie le quotient de la division entière alors que `/` renvoie la division réelle et renvoie donc toujours un flottant même si les opérandes sont toutes les deux des entiers.
  On rappelle qu'en *Go*, `//` n'est pas un opérateur de division : il sert à introduire un commentaire...
- N'oubliez pas le symbole deux-points (`:`) pour débuter un bloc.
- La fonction *print* produit par défaut un retour à la ligne (comme le *fmt.Println* de *Go*). Pour rester sur la même ligne, ajoutez le paramètre *end=""*.