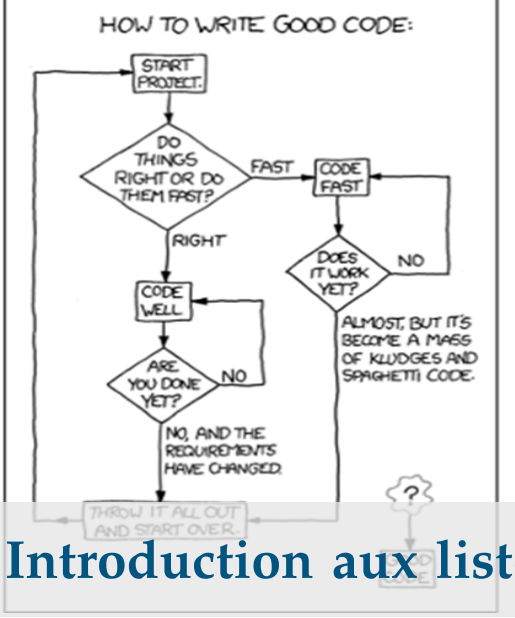

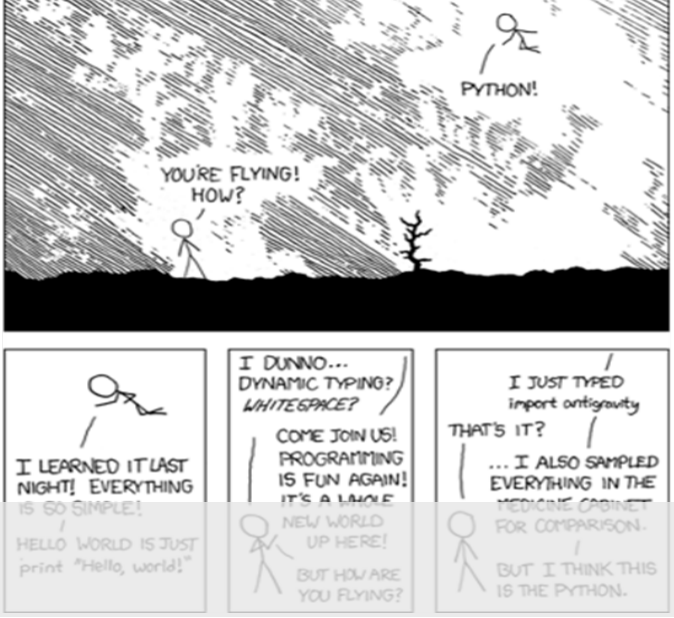

3 Introduction aux listes

3.1 Tableaux et listes

3.1 Tableaux et listes 1
3.2 Les listes 2
3.3 Chaînes et type `str` en `python` 6

- Choisir un type de variable.
- Concevoir un algorithme utilisant une structure conditionnelle (Si), une structure itérative (while), une structure itérative (for).
- Manipuler des listes.

3.1.1 Deux structures de données en informatique.

Les données manipulées en informatique sont organisées, que ce soit dans la mémoire où dans la manière d'y accéder, de les manipuler. Une telle organisation porte le nom de **structure de données**, en voici deux grandes.

Définition – Tableaux

Les données sont stockées dans des cases contiguës de la mémoire de l'ordinateur, chaque emplacement étant souvent indicé par un entier. A priori, il n'est pas possible d'en rajouter autant que possible : cette place a été pré-allouée. Pour accéder au contenu de l'emplacement numéro k, la machine a seulement besoin de connaître l'adresse de la première case mémoire et de la largeur de chaque case. L'accès en lecture et en écriture à une donnée (à partir du numéro de son emplacement) se fait donc en temps constant (on dit $\mathcal{O}(1)$). L'ajout d'une nouvelle donnée à un tableau peut alors être problématique !

Définition – Liste (chaînée)

Les données ne sont pas stockées de manière organisée dans la mémoire, mais de chaque emplacement on pointe l'emplacement suivant. Ainsi, l'accès (en lecture ou en écriture) ne se fait plus en temps constant, mais l'ajout d'une nouvelle donnée se fait simplement en temps constant.

3.1.2 Et en `python`...

Les objets de type `list` en Python sont des tableaux : c'est une dénomination fâcheuse car, partout ailleurs en informatique, le terme **liste** désigne en fait une liste chaînée. Prenez donc l'habitude de dire « tableau », ou « liste Python » et non « liste » quand vous parlez de cette structure.

Cependant, il y a une raison à cette dénomination : ces tableaux en Python possèdent presque la propriété des listes, dans le sens où l'ajout d'un nouvel élément se fait en temps constant **amorti**. Cela signifie que la plupart de ces ajouts se font en temps

constants car en fait Python a « gardé des cellules en réserve », mais parfois l'ajout d'un élément force la création d'un nouveau tableau avec plus de cellules de réserve, et la copie de l'ancien tableau dans le nouveau. Cette opération est coûteuse, mais elle est rare. En moyenne, chaque ajout à un coût constant.

On parle alors de **tableaux dynamiques**.

3.2 Les listes

3.2.1 Déclaration et initialisation

```
# Déclaration d'une liste vide
L = []
# Déclaration d'une liste avec des éléments connus
L = [10,-12,20,54,4]
```

3.2.2 Accès et modification aux éléments d'une liste

Les éléments d'une liste sont indexés. Cela signifie qu'ils sont numérotés par un numéro. Le premier élément de la liste est porte le numéro 0.

L = [	10,	-12,	20,	54,	4]
	↑	↑	↑	↑	↑
Indice	0	1	2	3	4

On peut alors accéder aux éléments par leurs indices.

```
>>> L[0] = 0
>>> print(L)
    [0,-12,20,54,4]
```

Attention à utiliser un indice existant !

```
>>> L[12]
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    IndexError: list index out of range
```

Les indices appartiennent à $[\![0, \texttt{len(t)}[\![$. Mais on peut aussi compter les éléments à partir de la fin, en utilisant des indices négatifs (plutôt déconseillé pour le moment, à part pour avoir accès au dernier élément)

```
L[-1] # dernier élément
L[-2] # avant-dernier
```

Attention, là aussi, à ne pas sortir du tableau !

```
>>> t[-4]
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    IndexError: list index out of range
```

3.2.3 Ajout et suppression des éléments d'une liste

On peut ajouter un élément à une liste. On utilise en priorité la méthode `append`. Avec l'utilisation de cette méthode l'élément est ajouté à la « droite » de la liste.

```
>>> L = [10,-12,20,54,4]
>>> L.append(12)
>>> print(L)
    [10,-12,20,54,4,12]
```

On peut aussi concaténer des listes :

```
>>> L = [10,-12,20,54,4]
>>> L = L + [1,2,3]
>>> print(L)
    [10,-12,20,54,4,1,2,3]
```

Enfin, on peut insérer un élément à un index donné. Tous les les éléments suivant voient donc leur index décalé vers la droite.

```
>>> L = [10,-12,20,54,4]
>>> L.insert(1,12) # Insertion de la valeur 12 au rang 1
>>> print(L)
    [10,1,-12,20,54,4]
```

Pour supprimer un élément, on peut utiliser son index (`del`) ou sa valeur (`L.remove()`). Attention, après suppression d'un élément tous les éléments suviants sont décalés vers la guache.

```
>>> L = [10,-12,20,54,4]
>>> L.remove(-12)
>>> print(L)
    [10,20,54,4]
>>> del L[0]
>>> print(L)
    [20,54,4]
```

Parcours de listes

Maîtriser le parcours d'une liste est essentiel.

Il est possible de parcourir les listes par valeurs.

```
L = [3,1,2,5,4,6]
for v in L :
    print(v)
```

On peut aussi faire un parcours par index (ou par indice).

```
L = [3,1,2,5,4,6]
for i in range(len(L)) :
    print(L[i])
```

Quelques calculs

La longueur se calcule avec la fonction `len`. Pour des listes Python de nombres, le maximum se calcule avec la fonction `max`, le minimum avec `min` et la somme des éléments de la liste avec `sum`.

```
len(t)
max(t)
min(t)
sum(t)
```

On peut aussi tester l'appartenance d'un élément dans une liste Python avec `in`.

```
42 in t
[] in [[]]
```

Enfin, on peut recopier plusieurs fois un **même** tableau avec l'opérateur $*$.

```
4*t
t*4
```

3.2.4 Tranchage – Slicing

On peut directement accéder à une sous-liste Python (ou tranche —**slice** en anglais— c'est-à-dire bloc de cases consécutives) d'une liste, c'est ce que l'on appelle le tranchage (*slicing*).

On utilise les syntaxes :

- `u[i:j]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`;
- `u[i:]` pour accéder à la tranche allant de l'indice `i` à la fin de la liste `u`;
- `u[:j]` pour accéder à la tranche allant du début de la liste `u` à l'indice `j-1`;
- `u[:]` pour accéder à la tranche complète (toute la liste `u`);
- `u[i:j:p]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`, par pas de `p`.

```
u
u[2:5]
u[2:]
u[:5]
u[1:8:2]
```

3.2.5 Autres méthodes

Python est un langage **orienté objet** : en `python`, tout est un **objet**, et ces objets sont regroupées en **classes**. Une **méthode** est une fonction qui s'applique aux objets d'une classe particulière. Si `method` est une méthode de la classe `c`, et si `a` est un objet de cette classe, alors on applique `method` à `a` avec la syntaxe suivante : `a.method()`, les parenthèses pouvant contenir des paramètres.

Les méthodes rassemblées dans le tableau 3.1 ne sont pas exigibles.

3.2.6 Tableaux multidimensionnels

On peut représenter une matrice avec des listes Python, par exemple en la décrivant ligne par ligne. Par exemple, on peut représenter la matrice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ par

```
M = [[1,2,3],[4,5,6],[7,8,9]]
```

les aux

Méthode	Description
`append(x)`	Ajoute `x` en fin de liste
`extend(L)`	Concatène la liste `L` en fin de liste
`insert(i,x)`	Insère `x` à la position `i`
`remove(x)`	Supprime la première occurence de `x` (erreur si impossible)
`pop(i)`	Supprime l'élément en position `i` (si vide, le dernier)
`index(x)`	Renvoie l'indice de la première occurence de `x` (erreur si impossible)
`count(x)`	Renvoie le nombre d'occurences de `x`
`sort(cmp,key,rev)`	Trie la liste (nombreuses options)
`reverse()`	Renverse la liste

On accède à la première ligne deuxième colonne par `M[0][1]`. Cependant, cela ne permet pas d'effectuer les opérations classiques sur les matrices. On préférera utiliser les possibilités de la bibliothèque de calcul numérique `numpy`.

3.2.7 Alias

Parfois, lorsque l'on manipule des tableaux, on observe un phénomène étrange qui laisse pantois : c'est l'**aliasing**. C'est une notion assez subtile, mais qu'il est bon de connaître pour éviter les problèmes qui en découlent, ou au moins avoir une parade lorsqu'on s'y trouve confronté.

Commençons par un exemple où tout se passe intuitivement :

```
1 x = 3
2 y = x
3 x = 42
```

`y` vaut alors 3.

Avec des listes Python, cela ne fonctionne pas exactement de cette façon :

```
1 x = [0]*5
2 y = x
3 x[0] = 42
```

`y` vaut alors `[42,0,0,0,0]`.

Tâchons de donner une explication à ce phénomène.

- Pour les objets de type **non mutable** (on ne peut pas les modifier, comme les types `int`, `float`, `bool`, `str`, `tuple`), une instruction du type `x = y` crée une nouvelle variable : la variable `x` pointe vers une case contenant la valeur de `y`, et l'on dit que `x` et `y` sont des **alias** de cette valeur. Réattribuer à `x` ou `y` une nouvelle valeur casse cet alias : les deux variables ne pointent plus vers la même valeur.
- Pour les objets de type **mutable**, les choses sont plus compliquées : on peut modifier leur contenu sans les réassigner. C'est le cas des listes `python`. Après une instruction du type `x = y` , les variables `x` et `y` pointent vers le même objet : ce sont des **alias** de cet objet là aussi. Mais si l'on modifie le contenu de `x`, sans réaffecter `x`, l'alias n'est pas cassé et l'on modifie à la fois `x` et `y`.

La fonction `id()`, qui affiche pour chaque objet son « numéro d'identité », permet de mettre cela en évidence :

```
x = 3
y = x
id(x),id(y)
x = 42
id(x),id(y)
```

```
x = [0]*5
y = x
id(x),id(y)
x[0] = 42
id(x),id(y)
```

Si l'on veut que x et y ne soient plus des alias, on pourra plutôt utiliser la méthode `copy()`.

```
x = [0]*5
y = x.copy()
id(x),id(y)
x[0] = 42
y
```

Il existe d'autres manières pour des tableaux de casser cet alias : `y = list(x)`, `y = x[:]` ou encore `y = x+[]`.

Cependant, en insérant des tableaux dans d'autres tableaux, on a une notion de « profondeur ». Les techniques données ci-dessus ne permettent de rompre un alias qu'au niveau de l'enveloppe externe. Il existe la fonction `deepcopy` de la bibliothèque `copy`, qui effectue une copie totale d'un objet, en profondeur comme son nom l'indique. Cela dit il est peu probable que nous en ayons besoin.

3.3 Chaînes et type `str` en `python`

3.3.1 Définition

Pour représenter des textes, on utilise la structure de données de « chaîne de caractères » (*string* en anglais). En `python`, cela correspond aux objets de type `str`.

Ces objets sont non-mutables, c'est-à-dire qu'une fois créés, on ne peut pas les modifier. Comme il ne peut y avoir des problèmes d'alias (par exemple, comme pour les listes `python`), nous ne détaillerons pas ici leur représentation en mémoire.

3.3.2 Création et lecture

On définit une chaîne de caractère en entourant ces caractères par des guillemets simples, doubles, ou trois guillemets simples ou doubles. L'utilisation de guillemets simples permet d'utiliser des guillemets doubles dans la chaîne de caractère, et vice-versa.

```
'Hallo Welt'
'It is "Hello World" !'
"Ja, aber 'not in germany' ..."
```

On peut signaler une tabulation par `\t` et créer une nouvelle ligne par `\n`.

```
s = u"\t Il faut une science politique nouvelle à un monde tout nouveau.\n
    [...]"
print(s)
```

Remarque

La présence de u devant les guillements permet de s'assurer de prendre les comptes les caractères spéciaux liés à l'encodage utf-8.

On accède alors aux caractères d'une chaîne en donnant son indice, comme pour les listes.

```
s[0]
s[-1]
```

Remarquons que le tranchage fonctionne similairement aux listes.

```
s[14:21]
s[:11]
s[37:]
```

On peut aussi créer une chaîne de caractères à partir de nombres ou de booléens avec la fonction `str`.

```
str(42)
str(-3.5)
str(4==4.)
```

Enfin, rappelons que ce type est non mutable.

```
>>> s[0] = "a"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s[0] = "a"
TypeError: 'str' object does not support item assignment
```

3.3.3 Opérations sur les chaînes

Comme pour les listes, on peut concaténer deux chaînes de caractères avec l'opérateur +.

```
'Hello'+'World'
```

On peut aussi dupliquer une chaîne de caractères plusieurs fois avec l'opérateur $*$.

```
print('GA-BU-ZO-MEU\n'*5)
```

La fonction `len` permet là encore de calculer la longueur d'une liste.

```
len(s)
```

On peut aussi chercher la présence d'un motif dans une chaîne avec `in`, son absence avec `not in`

```
'Science' in s
'science' not in s
```

3.3.4 Chaînes et méthodes

De nombreuses méthodes existent sur les chaînes, voici les plus utiles.

Méthode	Description
`count(m)`	Compte le nombre d'occurrences du motif `m`, sans chevauchement.
`find(m)`	Renvoie la première occurrence du motif `m`.
`islower() (et autres)`	Renvoie un booléen indiquant si la chaîne est en minuscules.
`join(bout)`	Concatène les éléments de la liste `bout`, séparés par la chaîne.
`lower()`	Met en minuscule la chaîne de caractères.
`split(sep)`	Sépare la chaîne selon le séparateur `sep`
`strip(char)`	Enlève les lettres en début/fin si elles sont dans la chaîne `char`.
`upper()`	Mets en majuscules la chaîne.

```
s = "    que ; d'espaces ; mes ; amis ! ;   "
s.count('es')
s.find('es')
s.islower()
'+'.join([str(i) for i in range(14)])
'HAHAHAHAHA !'.lower()
s.split(';')
s.split()
s.strip()
s.strip('q; ')
s.upper()
```