# Chapitre 3. Compléments (typage, complexité) et exercices de révisions

## 3.1 Indications de typage

On a vu au chapitre précédent que l'on pouvait passer différents types de paramètres à une fonction, par exemple il y a une grande différence entre ces deux fonctions :

```
def mystere1(L):
    s = 0
    for elt in L:
        s = s + elt
    return s
```

```
def mystere2(n):
    s = 0
    for i in range(n):
        s = s + i
    return s
```

La fonction `mystere1` prend en paramètre une liste de nombres `L` et calcule la somme de ses éléments, alors que la fonction `mystere2` prend juste en paramètre un nombre, et calcule la somme des nombres de 0 à $n-1$. A priori, lorsque l'on commence à lire le code de la fonction, on ne sait pas vraiment quel type de paramètre la fonction s'attend à recevoir. En regardant le nom du paramètre, on peut se douter que `L` est probablement censé être une liste, mais on pourrait avoir n'importe quel nom de paramètre... Lorsque l'on lit la ligne `for elt in L`, on devient sûr que `L` est censé être une liste : `for` veut absolument une liste à droite du `in`[1].

Une bonne pratique est de *documenter* le type de paramètre qui est attendu par la fonction, en ajoutant l'indication du type dans la liste des paramètres, par exemple `int` pour un entier, `list` pour une liste. On peut également documenter le type de ce qui est retourné par la fonction avec `->`. On obtient ainsi :

```
def mystere1(L: list) -> int:
    s = 0
    for elt in L:
        s = s + elt
    return s
```

```
def mystere2(n: int) -> int:
    s = 0
    for i in range(n):
        s = s + i
    return s
```

Ainsi, les fonctions perdent un peu de leur mystère, elles sont plus faciles à comprendre !

Attention, ces indications de typage ne sont à écrire qu'au moment de la *définition* de la fonction, il ne faut pas les écrire au moment des *appels* à la fonction.

## 3.2 Complexité

### 3.2.1 Introduction

Supposons que l'on dispose de deux listes d'entiers, on souhaite déterminer combien il y a d'entiers qui apparaissent à la fois dans les deux listes (On suppose que les listes sont elles-même sans doublons).

1. ou du moins quelque chose d'*itérable*.

```
def nbDoublons(L1: list, L2: list) -> int:
  n = 0
  for elt1 in L1:
    for elt2 in L2:
      if elt2 == elt1:
        n = n + 1
  return n
```

On se pose la question de la complexité de cette fonction : combien d'opérations effectue-t-elle ? Par opération, on entend chaque calcul, chaque comparaison, chaque affectation, etc.

Ici, pour chaque élément de la liste `L1`, on parcourt toute la liste `L2` pour déterminer s'il apparaît dedans. Ainsi, si l'on note $n_1$ la longueur de la liste `L1` et $n_2$ la longueur de la liste `L2`, la comparaison `elt2 == elt1` est faite $n_1 \times n_2$ fois, et le calcul `n + 1` et l'affectation `n = n + 1` sont faites au plus $n_1 \times n_2$ fois.

On dira alors ici que la complexité de `doublon` est $\mathcal{O}(n_1 \times n_2)$.

Il est important de remarquer que c'est juste l'ordre de grandeur qui nous intéresse : le fait qu'il y ait *trois* opérations (ou une seule, selon le résultat de la comparaison) dans chaque tour de la boucle `for j` n'est pas vraiment intéressant du point de vue théorique : si l'on avait un ordinateur deux-trois fois plus rapide ce facteur trois disparaîtrait. C'est pour cela que l'on ne fait pas apparaître ce facteur 3 dans la complexité, et l'on ne fait apparaître que les tailles des listes : un programme qui est trois fois plus rapide ou trois fois plus lent, ce n'est pas très important d'un point de vue théorique. Par contre, selon que les listes manipulées ont pour taille 1000, 10 000, 1 000 000, cela change énormément le temps de calcul !

### 3.2.2 Attention aux complexités cachées

Soient les fonctions :

```
def nbOccurrences(x: int, L: list) -> int:
  n = 0
  for elt in L:
    if elt == x:
      n = n + 1
  return n

def nbDoublons(L1: list, L2: list) -> int:
  n = 0
  for elt in L1:
    n = n + nbOccurrences(elt, L2)
  return n
```

Quelle est la complexité de la fonction `nbDoublons` ?

Attention à la complexité cachée dans l'appel à la fonction `nbOccurrences` ! Même si l'affectation a l'air simple, il faut compter tout le temps de calcul de l'expression utilisée. La fonction `nbOccurences` a en effet une complexité qui n'est pas triviale : si on note $n$ la taille de la liste `L`, elle a pour complexité $\mathcal{O}(n)$.

Ainsi, dans cette version aussi `nbDoublons` a pour complexité $\mathcal{O}(n_1 \times n_2)$

**Exercice 3.2.1** Quelles sont les complexités des fonctions `moyenneListe`, `nbPairsListe`, `maximumListe` ?

**Exercice 3.2.2** Quelle est la complexité de la fonction suivante :

```
def nbPairs(L1: list, L2: list) -> int:
  n = 0
  for elt in L1:
    if elt % 2 == 0:
      n = n + 1
  for elt in L2:
    if elt % 2 == 0:
      n = n + 1
  return n
```

## 3.3 Expressions booléennes

**Exercice 3.3.1** Parmi les expressions suivantes, quelles sont celles qui valent `True` si et seulement si les trois variables `a, b, c` ont toutes la même valeur ? Rayer les expressions incorrectes.

- `a == b and b == c`
- `a == b and b == c and a == c`
- `not (a != b or b != c)`
- `not (a != b or b != c or a != c)`

## 3.4 Exécution conditionnelle

**Exercice 3.4.1** Écrire une fonction `max3(x: int, y: int, z:int) -> int` qui calcule et retourne le maximum de trois nombres $x$, $y$, $z$. Donner plusieurs versions de cette fonction dont une utilise la fonction `max2` de l'exercice 1.4.1

**Exercice 3.4.2** Écrire une fonction `uneMinuteEnPlus(h: int, m: int)` qui calcule et retourne l'heure une minute après celle passée en paramètre, sous la forme d'un `tuple` de deux entiers correspondant à une heure et une minute valides. Exemples :

- `uneMinuteEnPlus(14,32)` retourne `(14, 33)`.
- `uneMinuteEnPlus(14,59)` retourne `(15, 0)`.

Ne pas oublier le cas de minuit.

**Exercice 3.4.3** Le service de reprographie propose les photocopies avec le tarif suivant : les 10 premières coûtent 20 centimes l'unité, les 20 suivantes coûtent 15 centimes l'unité et au-delà de 30 le coût est de 10 centimes. Écrire une fonction `coutPhotocopies(n: int) -> int` qui calcule et retourne le prix à payer pour `n` photocopies, en centimes.

## 3.5 Utilisation de `range` dans des boucles for

**Exercice 3.5.1** La fonction `factorielle(n)` (notée mathématiquement « $n!$ ») peut être définie de la manière suivante (on suppose que $n \geq 1$) :

$$n! = 1 \times 2 \times \cdots \times n$$

Écrire une fonction `factorielle(n: int) -> int` qui utilise cette définition pour calculer et retourner $n!$. Tester votre fonction en affichant les factorielles des nombres de 0 à 100.

**Exercice 3.5.2 (extrait DS 2015-16)** Rappel : On dit que $i$ est un diviseur de $n$ si le reste de la division de $n$ par $i$ est égal à 0.

1. Écrire une fonction `estDiviseur(i: int, n: int) -> bool` qui retourne `True` si `i` est un diviseur de `n` et `False` sinon.

2. Un nombre est dit *premier* s'il n'a que 2 diviseurs : 1 et lui-même. Calculez à la main sur papier la liste des nombres premiers inférieurs à 15.

3. Écrire une fonction `estPremier(n: int)` qui retourne `True` si `n` est premier, `False` sinon (on profitera du fait que seuls les nombres strictement inférieurs à `n` peuvent être diviseurs de `n`). Quelle est la complexité de cette fonction, en fonction de `n` ?

4. En s'aidant de la fonction `estPremier`, écrire une fonction `nbPremiers(n: int)` qui retourne le nombre de nombres premiers strictement plus petits que n. Note : l'idée est que `nbPremiers` *appelle* `estPremier`, ce qui simplifie beaucoup son écriture, dans `nbPremiers` il y a seulement besoin d'une boucle for. Quelle est la complexité de `nbPremiers`, en fonction de `n` ?

## 3.6 Pour aller plus loin : Analyse de données

Nous vous fournissons un module `bibcsv.py` qui permet d'ouvrir des fichiers CSV contenant juste une liste de nombres. Ce module est disponible en téléchargement sur le site https://moodle.u-bordeaux.fr/course/view.php?id=14677. Utiliser un clic droit et "enregistrer sous" pour l'enregistrer à côté de vos autres fichiers python. Pour utiliser un module, il faut commencer par l'*importer*, et toute session de travail utilisant ce module doit commencer par la phrase magique :

```
from bibcsv import *
```

Vous disposez alors de la fonction

| | |
|---|---|
| `ouvrirCSV(nom:str) -> list` | Ouvre le fichier *nom* et retourne la liste de nombres qu'il contient, par exemple : `l = ouvrirCSV("notes.csv")` |

**Exercice 3.6.1** Récupérer le fichier `notes.csv` depuis le site du cours, l'enregistrer de la même façon, et utiliser `ouvrirCSV` pour récupérer la liste des nombres stockée dans le fichier CSV :

```
maliste = ouvrirCSV("notes.csv")
```

et observer le contenu de la variable `maliste`.

1. Utilisez les fonctions `moyenneListe`, `ecartTypeListe`, `maximumListe`, pour analyser la liste de notes contenue dans `maliste`.

2. Vous pouvez créer votre propre fichier `.csv` avec LibreOffice. Dans une feuille de calcul, mettez les nombres à la suite dans la première colonne uniquement (ou bien en les copiant/collant depuis un document existant). Utilisez "Fichier", "Enregistrer sous", saisissez un nom de fichier en utilisant l'extension `.csv` et validez, confirmez que c'est bien le format CSV que vous désirez utiliser, et utilisez les options par défaut. Vous pouvez alors charger le fichier dans python à l'aide d'`ouvrirCSV` et effectuer les mêmes analyses.

3. Récupérez sur le site et ouvrez de la même façon le fichier `temperatures.csv`, et effectuez les mêmes analyses.

4. Observez la fonction suivante :

```
def mystere(L: list, x: int) -> bool:
    cpt = 0
    for elt in L :
        if elt > x:
            cpt = cpt + 1
            if cpt == 3:
                return True
        else:
            cpt = 0
    return False
```

Que retourne-t-elle lorsqu'on lui passe en paramètres la liste des températures et 30 ? Et lorsque l'on passe 35 au lieu de 30 ? Faites-la tourner dans *Python Tutor* pour bien comprendre ce qui se passe. Quelle est sa complexité ?

## 3.7 Pour aller plus loin : notion de complexité en moyenne

Dans l'exercice 2.1.3 nous avons effectué des mesures sur des listes, mesures dont l'évaluation nécessite la prise en compte de *tous les éléments de la liste* et, ce, *quel que soit la liste* considérée. Il existe cependant des propriétés qui peuvent être calculées sans nécessairement prendre en compte tous les éléments de la liste considérée. Par exemple, si l'on cherche à évaluer une propriété *"au moins un des éléments de la liste est pair"*, alors on peut arrêter l'évaluation dès qu'un élément pair est rencontré : le résultat est alors déterminé et ne changera pas quelques soient la valeur des autres éléments. Dans le même registre, le code de la fonction suivante est aussi élégant qu'inefficace :

```
def existePairListeInefficace(L: list):        # gaspille de l'energie
    return nbPairsListe(L) != 0
```

**Exercice 3.7.1** On considère les fonctions `existePairListe(L)`, qui renvoie `True` si au moins un des nombres de la liste est pair et `False` sinon, et `tousPairsListe(L)` qui renvoie `True` si tous les nombres de la liste sont pairs, et `False` sinon.

Écrire ces deux fonctions en faisant en sorte qu'elles retournent leur résultat dès que celui-ci est déterminé. Testez ces fonctions.

**Exercice 3.7.2** On se propose d'approfondir l'exercice 3.7.1

Comparer le nombre de tests réalisés par les fonctions `existePairListe(L)` et `existePairListeInefficace(L)` pour les cas où une liste de 100 éléments est passée en paramètre et que cette liste :

- ne contient aucun nombre pair (appelé complexité dans le pire cas) ;
- ne contient aucun nombre impair (appelé complexité du meilleur cas).

Pour déterminer si une liste contient ou pas un nombre pair, nous avons implémenté deux algorithmes, l'un naïf (exercice 3.7.2) l'autre optimisé (exercice 3.7.1) puis nous avons comparé leurs performances dans les meilleurs et pires cas. Mais en pratique, est-on généralement plus proche du meilleur cas ? du pire cas ? ou bien entre les deux ? En fait, pour comparer les performances de ces deux algorithmes, il est intéressant de comparer le nombre *moyen* de tests utilisés par chaque algorithme pour traiter un ensemble *pertinent* de listes.

Pour définir mathématiquement cette notion d'ensemble pertinent, il est d'usage en informatique de considérer des ensembles regroupant toutes les entrées ayant la même taille. Ici, la

taille de l'entrée de nos deux algorithmes correspond à la longueur de la liste passée en paramètre. Ainsi la question que nous allons résoudre pour chaque algorithme est : « combien de tests nécessite en moyenne cet algorithme pour traiter une liste de $n$ entiers ? ».

Pour l'algorithme naïf (noté $\mathcal{A}_{\text{naïf}}$), qui parcourt systématiquement toute la liste $\mathsf{liste}_n$ de $n$ éléments, le coût de traitement de toute liste de $n$ entiers est de $n$ tests. On a donc :

$$\text{Complexité\_Moyenne}\left(\mathcal{A}_{\text{naïf}}(\mathsf{liste}_n)\right) = n$$

Pour analyser l'algorithme optimisé (noté $\mathcal{A}_{\text{optimisé}}$) appliqué à une liste $\mathsf{liste}_n$ de $n$ éléments, nous allons simplifier le problème en considérant le cas où les nombres pairs et impairs apparaissent de façon équiprobable dans les listes considérées. Comme on s'intéresse à la parité, on peut de plus restreindre l'analyse aux listes dont les éléments appartiennent à $\{0,1\}$ [2]. On note $L_n$ l'ensemble des listes de $n$ éléments dans $\{0,1\}$. Calculons maintenant le coût de traitement de toutes les listes de $L_n$ en remarquant les faits suivants :

- il y a $2^n$ listes dans $L_n$ ;
- une liste sur deux de $L_n$ commence par un nombre pair et donc $2^{n-1}$ listes nécessitent un seul test pour être traitées par l'algorithme optimisé ;
- une liste sur quatre de $L_n$ a son premier nombre pair en deuxième position et donc $2^{n-2}$ listes nécessitent deux tests ;
- de façon plus générale, il y a $2^{n-i}$ listes de $L_n$ ayant son premier nombre pair en $i$-ième position et demandant $i$ tests ;
- le traitement de la seule liste de $L_n$ ne contenant pas de nombre pair nécessite $n$ tests.

Au total il faut $1.2^{n-1} + 2.2^{n-2} + 3.2^{n-3} + \cdots + n.2^{n-n} + n$ tests, le nombre moyen de tests est donc de

$$\frac{\sum_{i=1}^{i=n} i.2^{n-i} + n}{2^n} = \sum_{i=1}^{i=n} \frac{i}{2^i} + \frac{n}{2^n} = 2 - \frac{1}{2^{n-1}}$$

Sous l'hypothèse d'équiprobabilité des nombres pairs et impairs, on a donc :

$$\text{Complexité\_Moyenne}\left(\mathcal{A}_{\text{optimisé}}(\mathsf{liste}_n)\right) = 2 - \frac{1}{2^{n-1}} < 2$$

**Exercice 3.7.3** Calculer le nombre *moyen* de tests réalisés par l'algorithme optimisé dans le cas de listes ne contenant qu'un seul nombre pair placé aléatoirement dans la liste.

## 3.8 L'essentiel du chapitre

### 3.8.1 Typage

Une bonne pratique est de mettre les types dans les définitions de fonctions :

```
def f(n: int, m: int) -> int:
    if n < m:
        return n
    return m
```

On peut ainsi faire un résumé des fonctions que l'on connait avec leur typage :

2. Cette restriction ne modifie en rien le calcul du nombre de tests à réaliser puisqu'il suffit de transformer les éléments pairs en 0 et les impairs en 1 pour passer d'une liste d'entiers à une liste de 0 et de 1 .

```
len(L: list) -> int
range(fin: int) -> list
range(debut: int, fin: int) -> list
range(debut: int, fin: int, pas: int) -> list
```

Attention, ces indications de typage ne sont à écrire qu'au moment de la *définition* de la fonction, ou quand on documente leur existence comme ci-dessus, il ne faut pas les écrire au moment des *appels* à la fonction.

### 3.8.2 Complexité

La *complexité* exprime l'ordre de grandeur du nombre d'opérations effectuée par une fonction, en fonction des tailles des paramètres. Typiquement quand une fonction prend en paramètre une liste de taille `n`, la complexité peut être $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, ... Quand elle prend en paramètre deux listes, l'une de taille $n_1$ et l'autre de taille $n_2$, la complexité peut être $\mathcal{O}(n_1+n_2)$, $\mathcal{O}(n_1 \times n_2)$, ...