Précisions sur les variables

C. Charignon

Dans ce chapitre nous revenons sur quelques subtilités dans la manipulation des variables, et en particulier sur une grosse différence entre les tableaux et les autres types que nous avons rencontrés (entiers, flottants, booléens, chaînes).

Table des matières

Première partie
Cours

1 Variables globales et variables locales

1.1 Principe de base

Par défaut, toutes les variables utilisées dans un programme sont « locales », ce qui signifie qu'elles n'existent que dans ce programme.

```
def somme(t):
    res=0
    for x in t:
       res+=x
    return res


somme([1,2,3])
res
```

En pratique, seules les valeurs qui ont été renvoyées (via un `return`) seront accessibles depuis l'extérieur. Mais seule la valeur sera accessible, pas son nom.

De même un programme ne peut accéder qu'aux valeurs qui lui ont été envoyées en argument :

```
def périmètre(r):
    pi=3.14
    return 2*pi*r

périmètre(5)

def aire(r):
    return pi*r*r

aire(5)
# Ne fonctionne pas : aire ne connait pas pi
```

Ce comportement est sécurisant : chaque programme est parfaitement indépendant des autres. En cas de bug, il suffira de tester un à un chaque programme pour identifier celui qui est faux.

En outre, cela permet d'utiliser un même nom de variable (typiquement « `i` ») dans différents programmes sans créer d'interférence qui donnerait des erreurs incompréhensibles.

Cependant, on peut à titre exceptionnel créer une variable qui soit accessible par tous les programmes. Une telle variable sera dite « globale ». On réserve les variables globales aux données importantes et constantes du problème. Par exemple π [1], les constantes physiques comme $\mathcal{G}$, ...

C'est très simple : il suffit de définir la variable hors de tout programme :

```
pi=3.14 # variable globale

def périmètre(r):
    return 2*pi*r

périmètre(5)

def aire(r):
    return pi*r*r

aire(5)
```

Remarques :

1. La bibliothèque `numpy` charge en effet `np.pi` en tant que variable globale.

- Lorsqu'on charge une bibliothèque, par exemple `numpy` via un `import numpy as np`, ceci charge au passage plusieurs variables globales. Dans le cas de numpy, nous obtenons la variable π, accessible via `np.pi`.
- La coutume en Python est de mettre les variables globales entièrement en majuscule. Donc pour π on utiliserait plutôt `PI=3.14`.

1.2 Quelques subtilités

1.2.1 Modifier une variable mutable

Attention : à moins qu'elle ne soit mutable (voir partie suivante), une variable globale ne peut a priori pas être modifiée par un programme.

```
pi=3.14

def geométrieShadock():
    pi=12 # Ceci crée une variable *locale* pi, qui écrase la variable globale du même nom,
        ↪ mais seulement au sein de ce programme.

pi
```

Si vraiment vous y tenez, voici la syntaxe (hautement déconseillé) :

```
pi=3.14

def geométrieShadock():
    global pi # on dit à cette fonction d'utiliser la variable globale pi
    pi=12 # Ceci modifie la variable globale pi

pi
```

C'est pourquoi on conseille de réserver les variables globales aux constantes (constantes physiques du problème qu'on résout par exemple).

1.2.2 Les programmes sont dans des variables comme tout le monde

Les fonctions qu'on crée sont généralement stockées dans des variables globales : ils sont accessibles depuis n'importe quel autre programme. Ceci dit vous pouvez décider de créer un programme localement dans un autre programme : il ne sera alors accessible que depuis son programme père.

```
def sérieHarmonique(n):
    """ Renvoie sum_{i=1}^{n} 1/i """

    def inverse(x):
        return 1/x

    res=0
    for i in range(1,n+1):
        res+= inverse(i)
    return res

inverse(3)
```

1.2.3 Il y a différents niveaux de « globalité »

Considérons l'exemple suivant :

```
def somme_puissance(N, alpha):
    """ Renvoie sum_{k=1}^{N-1} 1/k^alpha """

    def puissance(k):
        return 1/k**alpha
```

```
    res=0
    for i in range(1,N):
        res+= puissance(k)

  return res
```

L'argument `alpha` de la fonction `somme_puissance` est accessible à la fonction `puissance`. On peut considérer qu'il s'agit d'une variable globale pour `puissance`.

Un exemple où on calcule $\sum_{k=0}^{N-1} e^k$, en commençant par calculer une valeur approchée de e.

```
def somme(N):

    # 1) Calcul de e
    e=0
    i_fact=1
    for i in range(N):
        e+= 1/i_fact
        i_fact*=(i+1)

    # 2) La suite qu'on va sommer
    def p(k):
        return e**k

    # 3) Calcul de la somme
    res=0
    for i in range(N):
        res+=p(k)

    return res
```

Ici, la variable `e` est locale à `somme`, mais elle est accessible à `p`, on pourrait dire qu'elle est globale pour `p`.
Remarque : Méthode complètement artificielle puisque le calcul de puissance (les `**`) de Python utilise de toute façon une exponentielle.

1.2.4 Parallèle avec la rédaction mathématique

En maths, lorsqu'on définit un nouvel objet, il y a deux manières :

- Avec un « soit ». Dans ce cas, la variable est globale ; elle est définie pour tout le paragraphe.
- Avec un quantificateur, ou autre notation spécifique (limite, somme, intégrale, ensemble). Dans ce cas elle est locale : elle n'est définie que dans la formule qui l'a créée.

2 Objets mutables

2.1 Principe

Le type `list` de Python est modifiable, ce qui signifie qu'on peut modifier le contenu d'une variable de type `list`.
Au contraire, les booléens, flottants, entiers, et chaînes de caractères sont des types persistants,ce qui signifie qu'on ne peut pas les modifier.

Lorsque j'effectue :

```
x=1
x=x+2
```

la seconde ligne n'a pas modifié le contenu de `x`, mais créé une nouvelle variable, encore appelée x et qui a écrasé l'ancienne.

Au contraire, si j'effectue :

```
t=[1,2,3]
t[1]=4
```

j'ai modifié le contenu de `t`.

(Faire des dessins pour expliquer la différence en mémoire.)

2.2 Conséquences pratiques

Il ne s'agit pas uniquement d'une question de point de vue : la différence entre une variable modifiable ou persistante a des conséquences très concrètes en Python.

2.2.1 Un programme peut modifier un objet mutable

Lorsqu'on passe en argument une variable mutable à un programme, celui-ci peut la modifier, et ces modifications sont répercutées hors de la fonction.

Exemple :

```
def vide(t):
    while t != []:
        t.pop()

t=[1,2,3,4]
vide(t)
t
```

À comparer avec :

```
def ajouteUn(x):
  x+=1

x=3
ajouteUn(x)
x
```

Le programme ajouteUn n'a pas l'effet escompté. Ici, x est de type int, qui est un type non modifiable, et l'expression `x+=1` ne peut modifier x, mais se contente de créer un nouvel entier, encore appelé x, qui cache l'ancien.

2.2.2 Un objet mutable n'est par défaut pas copié

Exemple :

```
t1=[1,2,3,4]
t2=t1
vide(t1)
print(t2)
```

Ici, `t2` et `t1` désignent le *même* tableau. Ainsi, quand je modifie `t1`, je modifie aussi `t2`.

Si on veut effectuer une vraie copie, on peut utiliser la fonction `deepcopy` de la bibliothèque `copy`.

```
import copy
t1=[1,2,3,4]
t2=copy.deepcopy(t1)
vide(t1)
print(t2)
```

Ici, `t2` et `t1` sont deux tableaux distincts. Initialement, `t2` contient les mêmes éléments que `t1`, mais ensuite on modifie `t1`.

2.2.3 Vocabulaire

Bien que ce soit une habitude loin d'être universelles, je prendrai désormais l'habitude de réserver le mot « variable » aux objets modifiables. Pour désigner le nom d'un objet non modifiable, j'utiliserai juste « nom » ou « identifiant ».

3 Deux types de programmes, deux types de commandes

Vocabulaire :

- Une « expression » est une ligne de code qui renvoie un résultat. Par exemple `2+2`.
- Une « instruction » est une ligne de code qui ne renvoie rien. Par exemple `x=2`, ou `t[2]=3` ou encore toutes les commandes d'affichage : `print, plot, show,...`.
  Même si rien n'interdit d'écrire une ligne parfaitement inutile, en général une instruction aura un effet sur la mémoire : elle aura créé ou modifié une variable, ou alors un effet d'affichage.

De manière similaire :

- un programme qui renvoie un résultat s'appelle une « fonction » ;
- et un programme qui ne renvoie rien s'appelle une « procédure ».

N.B. Concrètement, une procédure est un programme qui n'a pas de `return`.

Comme pour une instruction, une procédure aura en général un effet sur la mémoire : elle peut modifier une ou plusieurs variables mutables passées en argument (ou éventuellement des variables globales). Ou alors, elle va interagir avec le monde extérieur : afficher quelque chose à l'écran, envoyer un fichier à l'imprimante, jouer un son...

N.B. Il est tout à fait possible qu'un programme modifie une variable mutable *et* renvoie un résultat. Par exemple le `.pop` des tableaux : il *renvoie* la valeur du dernier élément du tableau, et il *modifie* le tableau en supprimant ce dernier élément. Il n'y a pas de vocabulaire spécifique pour cette situation. Pour ma part je dirai qu'il s'agit à la fois d'une fonction et d'une procédure.
Remarque : Le vocabulaire ci-dessus n'est pas universel : les sujets de concours peuvent préférer utiliser toujours le mot « fonction ». D'ailleurs, en Python, une procédure renvoie une valeur : `None`. Donc on peut dire qu'une procédure est une fonction qui renvoie `None`. (Subtile nuance entre une fonction qui renvoie « rien » et une fonction qui ne renvoie rien...)

Exemples :

```
def f(t):
    """ Ajoute 1 à la fin du tableau."""
    t.append(1)


t=[1,2,3,4]
t=f(t)
# Que contient t à présent ?
```

3.1 Un peu de vocabulaire et de convention

- Une bonne pratique concernant les noms de programme est d'utiliser un verbe au présent pour une procédure, et un nom ou un participe passé pour les fonctions. En effet, un utilisateur de base sera souvent porté à penser qu'un verbe conjugué indique une procédure.
  Par exemple Python propose de trier un tableau grâce à la fonction `sorted` ou la procédure `sort` (en l'occurrence, la procédure `sort` est une « méthode » sur les tableaux, ce qui signifie qu'elle s'utilise par `t.sort()`).
- Le mot « variable » n'est pas très clair, puisqu'une variable persistante ne peut pas varier : seules les variables modifiables peuvent vraiment varier... Un vocabulaire plus correct serait « identificateur », ou « identifiant », ou juste « nom » pour le nom utilisé, et « valeur » pour l'objet désigné par l'identificateur.
  Lorsque je tape `x=1`, j'ai associé la valeur 1 (c'est un entier) à l'identifiant `x`.

4 Création d'un tableau ou d'une matrice

Nous n'avons pour l'instant pas vu comment créer un tableau contenant un nombre de cases fixé à l'avance : jusqu'ici nous créeions uniquement des tableaux vides que nous remplissions ensuite par des `append`. Et nous n'avons pas du tout vu comment créer une matrice.

4.1 Tableau en compréhension

Le langage Python propose une syntaxe pratique proche de celle utilisée en maths pour définir des ensembles.

- Version simple : si f est une fonction,

```
[ f(i) for i in range(n) ]
```

crée le tableau `[f(0), f(1), ..., f(n-1)]`.

- Avec une condition : on peut ne garder que les éléments vérifiant une certain condition :

```
[ f(i) for i in range(n) if (condition) ]
```

- Enfin comme pour une boucle for, on peut itérer directement sur les éléments d'un tableau. Imaginons qu'on dispose d'un tableau `t` et qu'on veuille appliquer la fonction `f` à chacun de ses éléments, nous tapons alors :

```
[ f(x) for x in t]
```

C'est exactement l'analogue de la notation mathématique :

$$\{f(x)\ ;\ x \in t\}$$

4.2 Création d'un tableau

Par exemple, voici une fonction prenant en entrée un entier n et un élément x et renvoyant un tableau contenant n fois x :

```
def nouveauTableau(n,x):
    return [ x for i in range(n) ]
```

4.3 Création d'une matrice

Essayons ceci :

```
def nouvelleMatrice(n,p,x):
    ligne = [ x for j in range(p) ]
    return [ ligne for i in range(p)]
```

Ou encore, avec la fonction `nouveauTableau` :

```
def nouvelleMatriceBis(n,p,x):
    ligne = nouveauTableau(p,x)
    return nouveauTableau(n, ligne)
```

Sur ce, écrivons une fonction qui renvoie la matrice identité :

```
def identité(n):
    m= nouvelleMatrice(n,n,0)
    for i in range(n):
        m[i][i]=1
    return m
```

Ça ne marche pas !! Que s'est-il passé ?

La matrice renvoyée par `nouvelleMatrice` contient n fois la *même* ligne. De sorte que lorsqu'on modifie une ligne, on modifie en même temps toutes les lignes.

Remarque : `nouvelleMatrice(n,p,x)` a une complexité en $O(n+p)$, ce qui n'est guère plausible pour une fonction qui doit créer $n \times p$ cases.

Python ne fournit pas de fonction pour renvoyer une nouvelle matrice (par contre nous verrons qu'il y en a dans numpy). Écrivons-en une. Il s'agit de créer une *nouvelle* ligne pour chaque ligne, et non de reprendre tout le temps la même.

```python
def nouvelleMatriceCorrect(n,p,x):
    m=[]
    for i in range(n):
        m.append(nouveauTableau(p,x)) # à chaque ligne, on ré-appelle nouveauTableau, ce qui nous donne bien un *nouveau* tableau.
    return m
```

Ou, plus concis :

```python
def nouvelleMatriceCorrect(n,p,x):
    return [ nouveauTableau(n,x) for i in range(n)]
```

Ou même sans utilise la fonction `nouveauTableau` :

```python
def nouvelleMatriceCorrect(n,p,x):
    return [ [x for j in range(p)] for i in range(n) ]
```

Deuxième partie

Exercices

Exercices : Tableaux 2

Exercice 1. *! Type mutable ou pas

1. À quelles valeurs renvoient les identificateurs utilisés après exécution des codes suivants ? Expliquer.

```
x = 1
y = x
y = y+1
```

```
t = [1]
s = t
s.append(2)
```

2. Même question avec :

```
def f(x):
    x = x + 1
    return x
x=1
f(x)
```

```
def f(t):
    t.append(1)

t=[1]
f(t)
```

Préciser le type de f (ensembles de départ et d'arrivée de la fonction) dans chaque cas.

Exercice 2. **! Crible d'Ératosthène

Voici un algorithme très connu permettant étant donné un entier n de calculer tous les nombres premiers entre 2 et $n-1$.

Voici le principe : on commence par écrire tous les entiers jusqu'à $n-1$. Puis on « barre » les multiples de 2 sauf 2, puis les multiples de 3 sauf 3, etc. Au final, il ne reste que les nombres premiers.

Dans l'énoncé des questions intermédiaires ci-dessous, j'utilise le mot « programme ». Précisez à chaque fois s'il s'agit d'une fonction ou d'une procédure.

1. Écrire un programme `barre` prenant en entrée un tableau t d'entiers et un entier a et qui « barre » (en pratique qui mets 0) dans t toutes les cases d'indice multiple de a mais différent de a.
2. Écrire un programme `initialisation% prenant en entrée un entier` n `et renvoyant un tableau contenant tous les entiers de 0 à` $n-1$`. Pour simplifier l'indiçage, faire en sorte que pour tout` $i \in [\![0, n[\![$ `, \verb%t[i]=i`.
3. Écrire un programme `sansZero` prenant en entrée un tableau d'entier t et renvoyant un nouveau tableau contenant tous les nombres non nuls de t.
4. (bonus) Écrire un programme `enlèveZeros` prenant un tableau t et supprimant tous les zéros de t.
5. Écrire le programme final `Eratosthene` prenant en entrée un entier n et renvoyant la liste des nombres premiers entre 2 et $n-1$.

Exercice 3. **! Permutations de cases

1. Écrire une procédure `transpose` prenant en entrée un tableau `t` et deux indices `i j` qui échange le contenu des cases `i` et `j` de `t`.
2. Écrire une procédure qui inverse l'ordre des éléments dans un tableau.
3. À titre de comparaison, écrire une *fonction* qui prend un tableau et renvoie un nouveau tableau contenant les mêmes éléments mais dans l'ordre inverse.
4. Écrire une procédure prenant en entrée un tableau `t` et un indice i qui effectue une permutation circulaire sur `t[i:i+3]` : `t[i]` ira en case $i+1$, `t[i+1]` en case $i+2$ et `t[i+2]` en case `t[i]`.
5. Finalement écrire une procédure prenant en entrée un tableau `t` et deux indices i et j et qui effectue comme à la question précédente une permutation circulaire sur `t[i:j]`.

Exercice 4. ** Manipulation de matrices

1. Écrire une procédure pour échanger deux lignes dans une matrice.
2. Écrire une procédure pour échanger deux colonnes dans une matrice.
3. Écrire une fonction `transpose` prenant en entrée une matrice A de format $n \times p$ et renvoyant la matrice A^T, de format $p \times n$ telle que $\forall (i,j) \in [\![0,p[\![\times [\![0,n[\![,\ \left(A^T\right)_{i,j} = A_{j,i}$.

   Pourquoi est-ce compliqué de créer une *procédure* qui transforme A en A^T ?

Exercice 5. ** Tri par extraction

On propose de trier un tableau selon la méthode suivante : on trouve le plus petit élément et on le met en case 0 à l'aide de la procédure `transpose` de l'exercice 3. Ensuite on cherche le minimum de `t[1:]` qu'on mettra en case 1, etc.

1. Programmer ce tri.
2. Calculer sa complexité.

Exercice 6. ** Création de tableaux en compréhension

Le but de cet exercice est de manipulation la syntaxe propre à Python permettant de créer un tableau en compréhension.

1. Créer le tableau contenant les carrés des 10 premiers nombres entiers.
2. Créer le tableau contenant les carrés des entiers de $[\![0, 10[\![$ qui ne sont pas multiples de 3.
3. Écrire une fonction prenant en entrée un tableau `t` et renvoyant le tableau contenant les carrés des éléments de `t`.
4. Créer une fonction prenant en entrée deux tableaux `t1` et `t2` et renvoyant le tableau obtenu en sommant chaque élément de `t1` avec l'élément de `t2` correspondant.

Exercice 7. *! Supprimer un élément**

Le but de cet exercice est de se rendre compte des difficultés à vouloir supprimer un élément dans un tableau.

1. Écrire une procédure qui prend en entrée un tableau `t` et un entier `i` et qui supprime l'élément d'indice `i` de `t`. Les seules opérations autorisée sont les échanges de cases de tableau et le `t.pop()`.
2. Même question, mais sans perturber l'ordre des éléments de `t`.
   Remarque : Cette procédure est déjà programmée dans Python, on l'exécute en tapant `t.pop(i)`.
3. Calculer la complexité de la procédure précédente.
4. (a) Écrire une procédure `supprime` prenant en entrée un tableau `t` et un élément x et supprimant de `t` tous les x qu'il contient. L'ordre des éléments doit être conservé.
   (b) Démontrer la terminaison de votre procédure.
   (c) Quelle est sa complexité ?
5. (a) À présent, écrire une fonction `sansLesx` prenant les mêmes arguments mais qui *renvoie* un nouveau tableau contenant les éléments de `t` sauf les `x`.
   (b) Quelle est la complexité de cette fonction ?

Quelques indications

2 Entre deux versions plus ou moins optimisées de `barre` et `Erathostène`, il peut y avoir de très grosses différences de performance !

3 4) Attention à l'ordre des opérations.

4 1) se fait en une ligne alors que 2) nécessite une boucle.

5 Pour tout $i \in [\![0, n[\![$, on va chercher le minimum de `t[i:]` et le mettre en case i. Il sera plus lisible d'écrire une fonction `indiceDuMinAPartirDe` prenant en entrée `t` et `i` et renvoyant l'indice du minimum de `t[i:]`.

7
1. Il y a des pièges ! Par exemple testez votre programme pour retirer les 2 de `[2,2,3,24]`.
2. Lorsqu'on effectue `t.pop(i)`, il faut décaler d'une case vers la gauche tous les éléments suivant l'élément supprimé.