Informatique — MPSI/PCSI, 2023–2024

# Correction du TP n° 1

## 1 B.A.–BA

### 1.1 Affectation

**Exercice 1.1**

1. La méthode proposée dans l'énoncé ne fonctionne clairement pas !

```
>>> a=1
>>> b=2
>>> a=b
>>> b=a
>>> a
2
>>> b
2
```

Analysons le comportement de Python sur cet exemple. Lorsque l'on entre la ligne `a=b`, la variable `a` contient la même valeur que `b` (c'est-à-dire `2`). Mais cela implique que la valeur contenue initialement dans `a` (ici `1`) a été effacée. Donc lorsque l'on entre ensuite `b=a`, cette commande n'a aucun effet : la valeur que l'on veut mettre dans `b` est celle qui y est déjà présente !

2. On peut se tirer d'affaire en utilisant une troisième variable `c`, qui est utilisée pour *sauvegarder* la valeur initialement contenue dans `a`.

```
>>> a=1
>>> b=2
>>> c=a
>>> a=b
>>> b=c
>>> del(c)
>>> a
2
>>> b
1
```

On commence donc par sauvegarder dans `c` la valeur initialement contenue dans `a`. Lorsque l'on entre ensuite la ligne `a=b`, la variable `a` contient la valeur initiale de `b` (c'est-à-dire `2`). Puis lorsque l'on entre ensuite `b=c`, alors `b` contient la valeur initiale de `a` (c'est-à-dire `1`). On efface enfin du registre la valeur de `c` (avec `del`) pour libérer la mémoire.

### 1.2 Types de variables

**Exercice 1.2**

1. Sans surprise, on constate que les produits de variables conduisent aux types :

| * | bool | int | float |
|---|---|---|---|
| bool | int | int | float |
| int | int | int | float |
| float | float | float | float |

2. De l'expérience précédente, on en déduit que `bool` $\subseteq$ `int` $\subseteq$ `float`.

### 1.3 Fonction

**Exercice 1.3**

1. L'implémentation suivante convient.

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

2. L'implémentation suivante convient.

```
def f(x,y):
    return(x*y)
```

3. Le plus simple serait de définir la fonction

```
def h(x,y,z):
    return(1+x+y*z)
```

mais en utilisant les deux fonctions déja définies, on peut aussi écrire :

```
def h(x,y,z):
    return(f(x)+g(y,z))
```

### 1.4 Variable locale, variable globale

**Exercice 1.4**

Le programme permet d'afficher dans la console :

```
6 2
9 2
9 3
9 3
```

ce qui signifie que la redéfinition locale de la variable `a` dans celle de la fonction `g` n'a aucune incidence globale alors que celle dans la fonction `h`, évidemment précédée du mot clé `global`, si.

### 1.5 Expressions logiques et branchements conditionnels

**Exercice 1.5**

1. Le programme ci-dessous convient.

```
def f(x,y):
    if x==y:
        z=1
    else:
        z=0
    return(z)
```

```
>>> f(1,2)
0
>>> f(1,1)
1
```

2. On observe ici que notre fonction possède toujours le comportement attendu sur les booléens. Cela est dû au fait que `x==y` est une condition booléenne même si `x` et `y` sont des booléens.

```
>>> f(True, False)
0
>>> f(True, True)
1
```

**Exercice 1.6**

1. Ici tout est une question d'indentation. Dans la déclaration de `f1`, la ligne de code `y=0` est au même niveau que le `if` en termes d'indentation (les deux sont situés à quatre espaces du bord gauche de l'éditeur). Ce qui signifie que `y=0` n'est pas dans le bloc d'instruction associé à `if` ou `else`. Lors de l'évaluation de `f1(a,b)`, la machine effectuera donc les opérations suivantes :
   - si $a > 0$ alors $x = a$ et $y = b$ ;
   - sinon $x = a$ ;
   - puis, et dans tous les cas, $y = 0$ (car cette instruction n'est pas située dans le bloc du `if`).

En revanche, dans la déclaration de `f2`, la ligne de code `y=0` est située quatre espaces plus à droite que le `else`, et juste en dessous. Cela signifie que `y=0` est situé dans le bloc d'instruction du `else`. Lors de l'évaluation de `f1(a,b)`, la machine effectuera donc les opérations suivantes :

— si $a > 0$ alors $x = a$ et $y = b$ ;
— sinon $x = a$ et $y = 0$.

On voit ainsi que c'est la fonction `f2` qui est une implémentation de `f`.

2. En relisant les explications faites à la question 1, on voit que `f1` est une implémentation de la fonction $g : (a, b) \mapsto (a, 0)$.

**Exercice 1.7**

1. Le programme ci-dessous convient.

```
def maximum2(x,y):
    if x>y:
        z=x
    else:
        z=y
    return(z)
```

2. Il y a différentes manières de faire ici. Si le plus simple est de faire l'inventaire des trois cas

```
def maximum3(x,y,z):
    if x>=y and x>=z:
        m=x
    elif y>=x and y>=z:
        m=y
    else:
        m=z
    return(m)
```

Plus simple que de faire l'inventaire des cas, il est possible de diviser le problème et d'utiliser la fonction `maximum2` deux fois

```
def maximum3(x,y,z):
    return(maximum2(maximum2(x,y),z))
```

ce qui conduit à un programme plus court, et donc plus simple à déboguer. Dans tous les cas, il faut éviter d'utiliser autant de `if` qu'il y a de cas !

3. Une fois encore, le plus simple est de décomposer le problème pour éviter d'avoir à définir chacun des cas. En travaillant sur des couples avec la fonction `maximum2`, il vient en deux étapes :

```
def maximum4(a,b,c,d):
    return(maximum2(maximum2(a,b),maximum2(c,d)))
```

4. Commençons par remarquer que pour 2 entiers il faut réaliser 1 test, et que pour $4 = 2^2$ entiers, il faut réaliser 2 tests au rang 1 et 1 test au rang 2. Ainsi, supposant $n = 2^m$ avec $m \in \mathbb{N}$, il vient $n/2$ tests au rang 1, $n/4$ au rang 2, $n/8$ au rang 3,..., $n/2^k$ au rang $k$ ; d'où : $n \sum_{k=1}^{m} \frac{1}{2^k} = n - 1$ tests.

**Exercice 1.8**

Les programmes ci-dessous conviennent.

```
def NON(b):
    if b:
        z=False
    else:
        z=True
    return(z)
```

```
def ET(a,b):
    if a:
        if b:
            z=True
        else:
            z=False
    else:
        z=False
    return(z)
```

```
def OU(a,b):
    if a:
        z=True
    elif b:
        z=True
    else:
        z=False
    return(z)
```

En exploitant les similudes d'écriture des tables de vérité entre les opérateurs ET et OU, il est aussi possible d'ecrire la fonction `OU` avec les fonctions `NON` et `ET` selon :

```
def OU(a,b):
    return(NON(ET(NON(a),NON(b))))
```

**Exercice 1.9**

1. Il s'agit ici de calculer un discriminant, puis de distinguer les trois cas possibles. Le plus naturel est soit d'imbriquer deux if ou d'utiliser un `elif` (ce qui est plus simple ici). Il faut par contre éviter de mettre autant de `if` qu'il y a de cas car tous les tests seront réalisés à chaque fois, même si un des précédents était vrai. Les codes suivants conviennent.

```
def nb_racines(a,b,c):
    delta=b**2-4*a*c
    if delta>0:
        z=2
    elif delta==0:
        z=1
    else:
        z=0
    return(z)
```

```
def nb_racines(a,b,c):
    delta=b**2-4*a*c
    if delta>=0:
        if delta==0:
            z=1
        else:
            z=2
    else:
        z=0
    return(z)
```

2. On traite d'abord le cas $a = 0$ (qui se décompose en trois sous-cas), puis on est ramené au cas précédent : on appelle donc la fonction `nb_racines`.

```
def nb_racines_general(a,b,c):
    if a==0:
        if b==0:
            if c!=0:
                z=0
            else:
                z=-1
        else:
            z=1
    else:
        z=nb_racines(a,b,c)
    return(z)
```

```
>>> nb_racines_general(0,1,1)
1
>>> nb_racines_general(0,0,1)
0
>>> nb_racines_general(0,0,0)
-1
```

**Exercice 1.10**

1. L'instruction `a%b` où $a$, $b$ sont des entiers renvoie le reste de la division euclidienne de $a$ par $b$ et `a//b` renvoie le quotient de la division euclidienne de $a$ par $b$.
2. $b$ divise $a$ si, et seulement si, le reste de la division euclidienne de $a$ par $b$ est nul, ce qui signifie que $a$ est congru à 0 modulo $b$ ; ce qui se traduit par :

```
def divise(a,b):
    return(a%b==0)
```

3. Un entier est pair s'il est congru à 0 modulo 2, ce qui, en exploitant la fonction `divise`, s'écrit simplement :

```
def pair(n):
    return(divise(n,2))
```

4. Une année est bissextile si elle est divisible par 4 mais non par 100, sauf si elle est divisible par 400. En utilisant la fonction `divise`, la fonction `bissextile` peut s'écrire naïvement :

```
def bissextile(a):
    s=False
    if divise(a,4):
        if divise(a,100):
            if divise(a,400):
                s=True
        else:
            s=True
    return(s)
```

Notez qu'il est possible de simplifier cette fonction en notant qu'une année est bissextile si elle est :

— multiple de 4 (condition `divise(a,4)`) et
— non multiple de 100 ou multiple de 400 (condition `not(divise(a,100)) or divise(a,400)`, avec le « ou logique » incluant évidemment le « et »).

Ce qui conduit au code suivant.

```
def bissextile(a):
    return(divise(a,4) and (not(divise(a,100)) or divise(a,400)))
```

## 2 Structures itératives

**Exercice 1.11**

1. Simple application de la syntaxe de la boucle `for`. Dans le programme suivant, la valeur initiale de $S$ est l'élément neutre de la somme 0, puis pour $k$ allant de 0 à $n-1$ (ce qui s'écrit `for k in range(n)`), on ajoute $g(k)$ dans $S$ (ce qui s'écrit `S=S+g(k)`). En fin de boucle, on a alors $S = g(0) + g(1) + \cdots + g(n-1)$.

```
def sommation(g,n):
    S=0
    for k in range(n):
        S=S+g(k)
    return(S)
```

2. Il suffit de déclarer la fonction $g_1 : x \mapsto 2x+1$, puis d'utiliser le programme précédent.

```
def g1(x):
    return(2*x+1)
```

```
>>> sommation(g1,3)
9
>>> sommation(g1,100)
10000
```

D'après les résultats, on peut conjecturer que la somme des $n$ premiers impairs est $n^2$.

3. On raisonne comme pour la question précédente.

```
def g2(k):
    return((4*(-1)**k)/(2*k+1))
```

```
>>> sommation(g2,100)
3.1315929035585537
>>> sommation(g2,10**5)
3.1415826535897198
```

Il est alors tentant de conjecturer $\lim\limits_{n\to+\infty} u_n = \pi \ldots$ ce que l'on pourrait effectivement démontrer !

**Exercice 1.12**

1. On commence par remarquer qu'un entier $n$ est divisible par 7 si et seulement si $7 \times \lfloor \frac{n}{7} \rfloor = n$, c'est-à-dire que le reste de la division euclidienne de $n$ par 7 vaut 0, ce que l'on écrit `n%7==0`. On dira alors que $n$ est congru à 0 modulo 7. En rajoutant un `if` dans la boucle pour déterminer si l'on affiche le terme à l'écran, il vient le code :

```
def table131():
    for k in range(50):
        if (13*k)%7==0:
            print(13*k)
```

```
>>> table131()
0
91
182
273
364
455
546
637
```

Il faut noter ici que la fonction `table131` ne prend aucun argument en entrée (c'est pourquoi les parenthèses sont vides dans la ligne `def table131():`). On notera de plus que c'est une procédure car elle ne renvoie rien (il n'y a pas de `return` mais seulement des `print`).

2. Auparavant nous avions étudié les 50 premiers multiples de 13, ce qui au final ne donnait que 8 termes convenant. Pour en obtenir 50 il faudra donc étudier un plus grand nombre de multiples de 50. Pour faire cela il faudrait écrire un programme qui, tant qu'on n'a pas obtenu 50 nombres,

continue d'étudier les multiples de 13 suivants ; ce qui nécessite normalement une boucle `while`. On peut néanmoins s'en sortir en faisant varier `k` de 0 à un nombre $n$ suffisamment grand pour obtenir 50 nombres (en tâtonnant un peu, on voit que $n = 10^4$ convient), puis nous sortirons de la boucle avec un `break`, ce qui permet d'accélérer l'évaluation du programme.

```
def table132():
    s=0
    for k in range(10**4):# « bricolage !
        if s==50:
            break
        elif 7*int(13*k/7)==13*k:
            print(13*k)
            s=s+1
```

Notez que l'on a rajouté une variable `s` pour compter les nombres écrits à l'écran et déterminer à quel moment il faut s'arrêter.

Ce programme est l'exemple typique de *ce qu'il ne faut pas faire*, car c'est en bricolant qu'on finit par commettre des erreur importantes de programmation ! Ce programme **doit être réalisé avec une boucle `while`** sous la forme :

```
def table133():
    s=0
    k=0
    while s<50:
        if (13*k)%7==0:
            print(13*k)
            s=s+1
        k+=1
    return(L)
```

## 3 Boucle conditionnelle `while`

**Exercice 1.13**
Pour définir la fonction `EstPremier`, il est nécessaire de distinguer les cas $n \leqslant 2$ et les autres. Dans le cas $n \geqslant 3$, il faut d'abord tester la parité et si $n$ est impair tester tous les diviseurs possibles tant que $k^2 < n$. Le code suivant convient.

```
def EstPremier(n):
    s=True
    if n>2:
        if n%2==0:
            s=False
        else:
            k=3
            while k**2<=n and (n%k)!=0:
                k+=2
            s=(k**2>n)
    elif n==1:
        s=False
    return(s)
```

Comme la condition de la boucle *while* est la conjonction des deux propositions $(k^2 < n)$ et ($k$ ne divise pas $n$), il est possible de sortir de la boucle :

— si $k^2 > n$, auquel cas $n$ n'a pas de diviseur dans $[\![3, \lfloor\sqrt{n}\rfloor]\!]$, ce qui implique que $n$ est premier ;
— si $k$ divise $n$ auquel cas $n$ n'est pas premier.

Ainsi, la variable booléenne de sortie peut être simplement le résultat du test $k^2 > n$.

**Exercice 1.14**
1. Partant du fait qu'un entier $n$ est divisible par $k$ si, et seulement si, $k \times \left\lfloor \frac{n}{k} \right\rfloor = n$. Le principe de l'algorithme utilisant une boucle `for` est le suivant : nous allons déterminer dans l'ordre croissant

tous les entiers plus petits que $\min(a,b)$ (car tout diviseur de $a$ et de $b$ est nécessairement inférieur à $a$ et $b$) qui divisent à la fois $a$ et $b$, et à chaque fois que l'on en rencontre un on le stocke dans une variable `d`. Ainsi cette variable prendra successivement la valeur de chacun des diviseurs de $a$ et de $b$ (chaque nouvelle valeur mise dans `d` écrasera la précédente), et la dernière valeur stockée sera bien le plus grand commun diviseur de $a$ et de $b$.

```
def PGCD(a,b):
    d=1
    for k in range(1,min(a,b)+1):
        if a%k==0 and b%k==0:
            d=k
    return(d)
```

2. Pour déterminer le PGCD avec une boucle `while`, il suffit d'utiliser l'algorithme d'Euclide tant que les restes sont non nuls et de renvoyer le dernier reste non nul. Initialisant la variable `x` à $r_0 = a$ et `y` à $r_1 = b$, il vient le code suivant :

```
def PGCD(a,b):
    x,y=a,b
    while y != 0:
        x,y = y,x%y
    return(x)
```

* *
*