Informatique — MPSI/PCSI, 2024–2025

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

Le code le plus simple est une structure à 3 branches.

```
def g(x):
    if x==0:
        y=0
    elif x>0:
        y=1
    else:
        y=-1
    return(y)
```

que l'on peut aussi limiter à l'évaluation d'une expression en faisant appel à la fonction `bool` :

```
def g(x):
    return((1-2*(x<0))*bool(x))
```

équivalente à :

```
def g(x):
    if bool(x):# x!=0
        y=1
        if x<0:
            y-=2# ou y=-1
    else:
        y=0
    return(y)
```

Exercice 1.7

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 1.8

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 1.9

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

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

Exercice 1.10

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 1.11

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 1.12

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 $\lim_{n\to+\infty} u_n = \pi \dots$ ce que l'on pourrait effectivement démontrer !

Exercice 1.13

1. C'est de nouveau une simple application de la syntaxe de la boucle `for`. Dans le programme suivant, la valeur initiale de P est l'élément neutre du produit 1, puis pour k allant de 0 à $n-1$ (ce qui s'écrit `for k in range(n)`), on multiplie P par $g(k)$ (ce qui s'écrit `P=P*g(k)`). En fin de boucle, on a alors $P = g(0) \times g(1) \times \cdots \times g(n-1)$.

```
def produit(g,n):
    P=0
    for k in range(n):
        P=P*g(k)
    return(P)
```

2. Il suffit de déclarer la fonction $g_3 : x \mapsto 1 + \dfrac{1}{1+x}$, puis d'utiliser le programme précédent.

```
def g3(x):
    return(1+1/(1+x))
```

```
>>> produit(g3,100)
101.00000000000003
>>> produit(g3,10**3)
1000.9999999999947
>>> produit(g3,10**4)
10000.999999999889
```

D'après les résultats, on peut conjecturer que le produit des n premiers termes est $n+1$. Pour le démontrer, il suffit d'écrire $1 + \dfrac{1}{k} = \dfrac{k+1}{k}$ puis d'observer que la plupart des termes du produit se simplifient. On notera tout de même que la machine ne renvoie pas la valeur $n+1$ à cause d'erreurs d'arrondis successifs.

Exercice 1.14

1. On commence par initialiser une variable `U` (nom différent de celui de la fonction) à la valeur 1, car cette valeur est celle de u_0. Pour calculer les autres valeurs de u_n, on utilise une boucle `for`, en n'oubliant pas d'exclure le cas $k = 0$, car la formule de récurrence $u_{k+1} = \dfrac{u_k + 2}{k}$ n'est pas valable si $k = 0$. Et comme on a $u_1 = 1$, il est cohérent de ne faire aucun calcul dans le cas $k = 0$: la valeur initiale est la bonne. On initialise donc le parcours de la boucle `for` à $k = 1$. Pour obtenir u_n avec la formule de récurrence $u_{k+1} = \dfrac{u_k + 2}{k}$, il faut que la valeur finale maximale du parcours soit $k = n-1$ et donc que $k \in [\![1, n-1]\!]$, ce que l'on écrit `for k in range(1,n)`). Le programme ci-dessous convient.

```
def u(n):
    U=1
    for k in range(1,n):
        U=(U+2)/k
    return(U)
```

2. En demandant quelques valeurs de u_n à la machine, on a rapidement l'impression que cette suite tend vers 0 (ce qui peut être démontré aisément).

```
>>> u(1000)
0.0020040100301044078
>>> u(10**6)
2.00000400001e-06
```

Notez que la quantité `2.00000400001e-06` *désigne le réel* $2,00000400001 \times 10^{-6}$. *Mais de même que dans l'exercice précédent, ces résultats sont potentiellement entachées d'erreurs d'arrondis.*

3. En se contentant de réutiliser la fonction `U` pour calculer les différents termes de la suite, puis sommer ceux-ci, il vient le code suivant :

```
def vnaif(p):
```

```
    S=0
    for k in range(p+1):
        S=S+u(k)
    return(S)
```

Mais l'évaluation de ce programme est relativement lente, et on peut aisément l'améliorer. Il faut déjà remarquer que pour calculer un terme u_n de la suite avec la fonction `u`, on doit passer par le calcul de tous les termes précédents : $u_0, u_1, \ldots, u_{n-1}$. Or à la k-ième étape de la boucle présente dans la fonction `vnaif`, on calcule le terme `u(k)`, ce qui impose à la machine de recalculer les termes $u_0, u_1, \ldots, u_{k-1}$ alors qu'elles les avait déjà calculé à l'étape précédente ! Il est donc plus rapide d'effectuer en parallèle le calcul des termes de la suite et la somme de ceux-ci.

```
def v(n):
    U=1
    S=1
    for k in range(n):
        if k!=0:
            U=(U+2)/k
        S=S+U
    return(S)
```

```
>>> u(0)+u(1)+u(2)
5.0
>>> v(2)
5.0
```

4. En calculant quelques exemples, il semble que la suite (v_p) soit croissante, mais de manière assez lente. On peut en fait démontrer que cette suite tend vers $+\infty$, à la même vitesse que la suite définie par $w_n = \ln n$.

```
>>> v(100)
16.688535360936662
>>> v(1000)
21.32102484678836
```

```
>>> v(10**5)
30.534338390795075
>>> v(10**8)
44.349878919063435
```

Exercice 1.15

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 1.16

Évidemment, on voudrait implémenter cette fonction avec une boucle `for` sous la forme :

```
def cubes(n):
    s=0
    for k in range(1,n+1):
        s+=k**3
    return(s)
```

ce que l'on peut traduire avec une boucle `while` sous la forme

```
def cubes(n):
    s=0
    k=1
    while k<=n:
        s+=k**3
        k+=1
    return(s)
```

où l'on voit qu'il est nécessaire d'initialiser la valeur de k avant la boucle, puis de mettre sa valeur finale en condition du `while`. On notera toutefois que l'on peut simplifier cette approche « en partant de la fin », l'argument `n` pouvant servir de variable k.

```
def cubes(n):
    s=0
    while n>0:
        s+=n**3
        n-=1
    return(s)
```

Exercice 1.17

1. On applique des divisions successives par b sachant que le k-ième reste correspond au k-ième chiffre en partant de la droite. On applique ce mécanisme tant qu'il reste des chiffres, c'est-à-dire tant que le quotient de la division euclidienne est non nul. Il vient le code suivant :

```
def chiffres(n,b):
    C=0
    while n!=0:
        C+=1
        n//=b
    return(C)
```

où nous avons noté `n//=b`, syntaxe équivalente à `n=n//b`.

2. On reprend le code précédent en comptant les chiffres à rebours en partant de `i` sachant que pour en avoir `i`, il faut faire $i-1$ divisions euclidiennes par `b`. Le code suivant convient.

```
def chiffre(n,b,i):
    while i>1 and n!=0:
        i-=1
        n//=b
    return(n%b)
```

Exercice 1.18

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 1.19

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)
```

Exercice 1.20

1. La procédure la plus simple pour jouer est :

```
import random as rdm
def Jeu():
    # on genere le nombre mystere
    nombreMystere = rdm.randrange(1,11)
    # initialisation
    nombrePropose=0
    # boucle tant que l'utilisateur n'aura pas trouve le nombre.
    while nombrePropose != nombreMystere:
        # l'utilisateur propose son nombre
        nombrePropose = int(input("Quel est le nombre mystere ? "))
        # si le nombre est trop petit...
        if nombrePropose < nombreMystere:
            print("Votre nombre est trop petit; essayez encore !\n")
        # sinon si le nombre est trop grand...
        elif nombrePropose > nombreMystere:
            print("Votre nombre est trop grand; essayez encore !\n")
        # sinon c'est gagne !
        else:
            print("Felicitations, vous avez trouve le nombre mystere !\n")
```

2. On complète le code précédent en demandant le nombre de coups maximum et on complète la condition de la boucle `while` en demandant la conjonction des propositions « le nombre donné est différent du nombre mystere » et « le nombre de coups joués est inférieur au nombre de coups maximum ».

```
def Jeu():
    nombreMystere = rdm.randrange(1,11)
    nombrePropose=0
    nombreEssais = int(input(" De combien d'essais avez-vous besoin ? "))
    nombreTentatives=0
    while nombrePropose != nombreMystere and nombreTentatives<nombreEssais:
        nombrePropose = int(input("Quel est le nombre mystere ? "))
        if nombrePropose < nombreMystere:
            print("Votre nombre est trop petit; essayez encore !\n")
        elif nombrePropose > nombreMystere:
            print("Votre nombre est trop grand; essayez encore !\n")
        else:
            print("Felicitations, vous avez trouve le nombre mystere !\n")
        nombreTentatives=nombreTentatives+1
    if nombrePropose != nombreMystere:
        print("Trop tard, vous avez depasse le nombre d'essais autorises!\n")
```

* *
*