Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

2015/2016

TP n° 2 : Boucles

Correction de l'exercice 1 – Première version, avec tests emboîtés :

```
def bissextile1(n):
    bis = (n % 4 == 0) # bis prend la valeur True ssi n divisible par 4
    if n > 1582:      # sinon, c'est terminé
        # première exception:
        if n % 100 == 0:
            bis = False
        # exception de l'exception:
        if n % 400 == 0:
            bis = True
    return bis
```

Voici une deuxième version, dans laquelle nous avons regroupé les tests concernant le calendrier grégorien :

```
def bissextile2(n):
    bis = (n % 4 == 0)
    if (n > 1582) and (n % 100 == 0) and not (n % 400 == 0):
        bis = False
    return bis
```

La troisième version est encore plus synthétique, et n'utilise que les opérations sur les booléens :

```
def bissextile3(n):
    return n % 4 == 0 and ((n <= 1582) or (not (n % 100 == 0) or (n % 400 == 0)))
```

Correction de l'exercice 2 –

1. Après initialisation, on itère la construction de la série harmonique tant qu'on n'a pas dépassé le seuil A. La terminaison de cet algorithme est assurée par la divergence de la série harmonique. Mais cette divergence est lente (de l'ordre de $\ln(n)$). Cela explique le temps d'attente assez long pour des valeurs trop grande de A. Lors d'une itération aussi longue, il faut aussi prendre garde aux problèmes d'inexactitude de la représentation des réels, qui peuvent légèrement perturber les calculs.

   ```
   A = eval(input("Entrez␣une␣valeur-seuil␣A␣:␣"))
   S = 0
   n = 0
   while S < A:
       n += 1
       S += 1 / n
   print("La␣série␣harmonique␣dépasse␣le␣seuil␣{}␣pour␣la␣première␣fois␣à ␣l'indice
   {}".format(A,n)) #(en une seule ligne)
   ```

2. On peut remarquer que contrairement à ce qu'on pourrait penser, il n'est pas nécessaire de faire explicitement l'échange de a et b lorsque $a > b$ initialement. Cet échange se fait automatiquement lors de la première étape de l'algorithme (pourquoi ?) On obtient alors :

   ```
   def pgcd (a,b):
       while a!= 0:
           a, b = b % a, a
       return(b)
   ```

   On peut aussi en donner une version récursive (fonction faisant appel à elle-même pour des valeurs différentes des paramètres). Attention dans cette situation à donner la condition initiale, sous forme d'une struture conditionnelle.

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

On n'a pas besoin de mettre de `else` ici, car l'instruction `return` provoque la sortie de la fonction. Ainsi, si $a = 0$, la dernière ligne n'est pas lue.

Correction de l'exercice 3 – La suite de Syracuse, aussi appelée suite de Collatz, fournit une des plus célèbres conjectures non élucidées à ce jour, à l'énoncé particulièrement simple : pour toute valeur initiale n, la suite définit finit-elle toujours par tomber sur la valeur 1 (puis boucler sur le cycle 4,2,1) ?

```
def syracuse(n):
    """ Calcul de la suite de Syracuse (Collatz) de valeur initiale n """
    u = n         # initialisation de u
    M = n         # initialisation du maximum rencontré
    i = 0         # indice (temps de vol)
    while u != 1:
        if u % 2 == 0:
            u = u // 2
        else:
            u = 3 * u + 1
        if M < u:
            M = u
        i += 1
    return M, i

n = eval(input('Valeur␣initiale␣:␣'))
h, t = syracuse(n)
print("Hauteur␣de␣vol␣:␣{}".format(h))
print("Temps␣de␣vol␣:␣{}".format(t))
```

Correction de l'exercice 4 – Calculs de suites récurrentes, et de sommes
Les questions sont indépendantes.

1. Soit la suite définie par $u_0 = 0$, et pour tout $n \in \mathbb{N}$, $u_{n+1} = \sqrt{3u_n + 4}$.
   (a) Il s'agit simplement d'une itération avec la boucle `for`, puisqu'on sait quand on s'arrête.

```
import math

n = eval(input('Donnez␣n␣:'))

u = 0
print('u_{}␣=␣{}'.format(0,u))
for i in range(n):
    u = math.sqrt(3 * u + 4)
    print('u_{}␣=␣{}'.format(i+1,u))
```

   L'utilisation de ce programme semble indiquer que (u_n) converge vers 4, ce qu'il n'est pas très dur de prouver mathématiquement.

   (b) Cette fois, on utilise une boucle `while`. Contrairement à la boucle `for`, il nous faut ici définir manuellement notre compteur d'indice i.

```
import math

def rang_minimal(eps):
    u = 0
    i = 0
    while u <= 4 - eps:
        u = math.sqrt(3 * u + 4)
        i += 1
    return i
```

Utilisé avec $\varepsilon = 10^{-8}$, on trouve 21, ce qui est compatible avec les valeurs de u_n qu'on a calculées dans la question précédente.

2. Attention à ne pas mélanger terme général et somme partielle. Nous avons donc besoin de deux variables, en plus de l'indice, afin de faire le calcul de la somme au fur et à mesure du calcul des termes u_n. On pourrait aussi envisager de calculer d'abord tous les termes u_n, les stocker dans un tableau et calculer la somme ensuite, mais cette méthode est gourmande en mémoire.

```
def calcule_S(n):
    u = 1
    S = 1
    for i in range(n):
        u = 1 / (u + 1)
        S += u
    return S
```

On trouve $S_{1000} = 618.9516037633203$. Si on calcule le terme général, on trouve $u_{1000} = 0.6180339887498948$, c'est-à-dire presque la même chose à un facteur 1000 près. C'est normal, ce n'est rien de plus que le théorème de la limite de Cesàro.

3. Il s'agit ici d'une récurrence d'ordre 2, ce qui revient à une récurrence d'ordre 1 sur le couple (u_n, u_{n+1}). Le principe est donc le même que plus haut, en faisant attention à bien faire des affectations de couples.

```
import math

def suiteu(n):
    print("u_0 = 0")
    if n > 0:
        u = 0
        v = 2     #initialisation: les 2 premières valeurs
        print("u_1 = 2")
        for i in range(n-1):
            u, v = v, math.sin(u) + 2 * math.cos(v)
            print("u_{} = {}".format(i+2,v))

n = eval(input("Entrez une valeur de n : "))
suiteu(n)
```

L'observation des 1000 premiers termes semble indiquer que la suite ne converge pas.

4. Nous avons maintenant une relation qui est, globalement, d'ordre 3. Nous adoptons la même démarche que ci-dessus, en itérant la suite (u_n, u_{n+1}, u_{n+2}), et en faisant une distinction de cas suivant la parité de n.

```
import math

def f(x,y):
    return x * math.cos(y) + y * math.cos(x)

def un(n):
    print("u_0 = 0")
    if n > 0:
        print("u_1 = 1")
        v, w = 0, 1 #initialisation sur 2 termes, le 3e pouvant se déduire de la
                    #relation de récurrence, d'ordre 2 pour cette parité
        for i in range(n-1):
            if i % 2 == 0:
                u, v, w = v, w, f(v,w)
            else:
                u, v, w = v, w, f(u,w)
            print("u_{} = {}".format(i+2,w))

n = eval(input("n = "))
un(n)
```

Cette fois-ci, il semble y avoir une convergence, vers une valeur à peu près égale à 1.0471975512. Comparez à $\frac{\pi}{3}$...

Correction de l'exercice 5 – On calcule les lignes les unes après les autres, dans une liste. La formule de Pascal nous permet de passer d'une ligne à l'autre : pour trouver la nouvelle valeur au rang k, il suffit d'ajouter à l'ancienne valeur à rang k l'ancienne valeur au rang $k-1$. À condition de partir de la fin de la liste, cela peut se faire en place (la nouvelle ligne est calculée dans la même liste que l'ancienne). On réserve 6 caratères pour les valeurs (afin d'obtenir l'alignement). Cela nous permet d'aller jusqu'à la ligne d'indice 22. De toute manière, au-delà, l'affichage d'une ligne ne tient plus sur un écran.

```
def affiche(liste):
    for i in liste:
        print('{:>6}'.format(i), end='␣')
    print()                          #pour faire le retour à la ligne à la fin de la ligne

def triangle_pascal(n):
    li = [1]                         #initialisation, ligne 0
    for i in range(n):
        li.append(1)
        for j in range(len(li)-2,0,-1): #j va de len(li)-2 à 1 = 0+1, par pas de -1
            li[j] += li[j-1]         #Formule de Pascal
        affiche(li)

n = eval(input('n␣?'))
triangle_pascal(n)
```

Correction de l'exercice 6 –

1. La façon la plus naturelle de faire est de parcourir tous les entiers strictement positifs et strictement inférieurs à n, et de tester s'ils divisent n. Si oui, on les ajoute à la liste des diviseurs propres :

   ```
   def divpropre(n):
       """ Renvoie la liste des diviseurs propres de n """
       diviseurs = []
       for i in range(1,n):
           if n % i == 0:
               diviseurs.append(i)
       return diviseurs
   ```

   Voici une version plus compact du même algorithme, utilisant la possibilité de définir une liste par compréhension :

   ```
   def divpropre2(n):
       """ Renvoie la liste des diviseurs propres de n """
       return [i for i in range(1,n) if n % i == 0]
   ```

   Remarquez l'utilisation des raw string pour décrire la fonction. Placé à cet endroit, le commentaire sera affiché lors de l'appel à l'aide (fonction `help`) pour la fonction définie. Ainsi, `help(divpropre)` affiche la page suivante :

   ```
   Help on function divpropre in module __main__:

   divpropre(n)
       Renvoie la liste des diviseurs propres de n
   (END)
   ```

2. On recherche les nombres parfaits en calculant pour tout i la somme des diviseurs propres de i, et en la comparant à i. Si le nombre est parfait, on calcule ensuite la somme des inverses des diviseurs de i, en n'oubliant pas i parmi ces diviseurs.

   ```
   def parfaits(N):
       """ Renvoie la liste des nombres parfaits inférieurs à N,
           et de la somme des inverses de leur diviseurs """
       parfaits = []
       for i in range(1,N+1):
           S = 0
           divi = divpropre(i)
           for d in divi:       # calcul de la somme des diviseurs propres
   ```

```
            S += d
        if S == i:          # cas où i est parfait
            T = 1 / i       # calcul de la somme des 1/d (y compris d=i)
            for d in divi:
                T += 1 / d
            parfaits.append((i,T))
    return parfaits
```

Remarquez que j'ai pris soin à ne pas calculer 2 fois la liste des diviseurs. Cela n'a en fait pas tellement d'importance ici, vu le faible nombre de nombres premiers, mais dans certaines situations, calculer plusieurs fois la même chose peut beaucoup ralentir le programme.
Pour $N = 10000$, on obtient :

```
[(6, 2.0), (28, 2.0), (496, 2.0), (8128, 2.0)]
```

3. Pour trouver les couples amicaux, pour toute première coordonnée m, on calcule la somme n de ses diviseurs propes. Le couple (m, n) est amical si la somme des diviseurs propres de n est alors égal à m.

```
def amicaux(N):
    """ Renvoie les couples amicaux (m,n) pour m <= N """
    ami = []
    for m in range(1,N+1):
        n = somme(divpropre(m))
        if somme(divpropre(n)) == m:
            ami.append((m,n))
    return ami
```

Pour $N = 10000$, on trouve :

```
[(6, 6), (28, 28), (220, 284), (284, 220), (496, 496), (1184, 1210), (1210, 1184),
(2620, 2924), (2924, 2620), (5020, 5564), (5564, 5020), (6232, 6368), (6368, 6232),
(8128, 8128)]
```

Il est normal de retrouver les nombres parfaits. Il est aussi normal d'observer une symétrie.

Correction de l'exercice 7 – Supposons dans un premier temps la date valide, et postérieure au 1 janvier 1583. On commence par déterminer le rang à partir du 15 octobre 1582 du premier janvier de l'année qui nous intéresse, la date du 15 octobre 1582 étant considérée comme le jour 0. Pour cela on ajoute au nombre de jours restant en 1582, le nombre de jour de chacune des années entières s'étant écoulées. Cela nécessite de savoir si ces années sont bissextiles ou non. On calcule ensuite le rang dans l'année du jour considéré, en ajoutant au rang du jour dans le mois le nombrede jour des mois écoulés. Pour éviter les trop grandes discussions, le plus simple est de stocker dans une liste la longueur des mois, en faisant bien attention au cas des années bissextiles. La somme du rang du premier janvier et du rang dans l'année fournit alors le rang du jour voulu depuis le 15 octobre 1582. On réduit modulo 7 pour trouver le jour de la semaine. La liste des jours de la semaine commence par vendredi, car le jour 0 est un vendredi.
La fonction `convertit_date` a pour rôle de convertir la date donnée sous format `string` en une date numérique, en effectuant au passage les tests de compatibilité. Attention au fait que `eval` ne permet pas de convertir des chaînes du type `'01'`. Il faut d'abord supprimer les 0 inutiles. On obtient le programme finalisé suivant :

```
def bissextile(n):
    """ True ssi n est bissextile, pour n > 1582 """
    return ((n % 4 == 0) and (n % 100 != 0)) or (n % 400 == 0)

def calcul_premier_janvier(n):
    """ calcul en prenant pour 0 le 15 octobre 1582, le numéro du jour
    correspondant au 1er janvier de l'année n > 1582"""
    S = 16 + 30 + 31 + 1  # 1er janvier 1583
    for annee in range (1583,n):
        if bissextile(annee):
            S += 366
        else:
            S += 365
    return(S)
```

```
def calcul_rang_dans_annee(jj,mm,aaaa):
    """ calcul du rang dans l'année aaaa du jour jj/mm
    le 1er janvier étant le jour 0"""
    duree_mois = [31,28,31,30,31,30,31,31,30,31,30,31]
    if bissextile(aaaa):     # rajout d'un jour à février si nécessaire
        duree_mois[1] += 1
    rang = 0
    for i in range(mm - 1):
        rang += duree_mois[i+1] # somme des jours des mois pleins qui précèdent
    rang += jj-1             # somme des jours du mois en cours
                             # sachant qu'on commence la numérotation à 0
    return rang

def calcul_rang_depuis_15101582(jj,mm,aaaa):
    """ rang à partir du 15/10/1582 """
    if aaaa == 1582:
        rang = 0
        if mm == 10:
            rang = jj - 15
        elif mm == 11:
            rang = jj + 16
        else:
            rang = jj + 16 + 30
    else:
        rang = calcul_premier_janvier(aaaa) + calcul_rang_dans_annee(jj,mm,aaaa)
    return rang

def supprime_zeros(ch):
    """ Supprime les zéros initiaux d'une chaîne de caractère,
    afin de pouvoir faire ensuite la conversion en format numérique,
    impossible su la chaîne commence par des 0 """
    while ch[0] == '0':
        ch = ch[1:]
    return ch

def convertit_date(date):
    """ Convertit une date donnée sous le format 'aaaammjj' en
    trois entiers aaaa, mm, jj; avec tests de cohérence. On vérifie
    au passage que la date est postérieure au 15/10/1582"""
    if len(date) != 8:
        raise ValueError("Format␣de␣date␣invalide:␣format␣requis␣'aaaammjj'")
    elif date < '15821015':
        raise ValueError("Date␣invalide,␣antérieure␣au␣15/10/1582")
    else:
        aaaa = eval(supprime_zeros(date[:4]))
        mm = eval(supprime_zeros(date[4:6]))
        jj = eval(supprime_zeros(date[6:]))

        duree_mois = [31,28,31,30,31,30,31,31,30,31,30,31]
        if bissextile(aaaa):
            duree_mois[1] += 1
        if (mm > 12) or (jj > duree_mois[mm-1]):
            raise ValueError("Date␣invalide,␣inexistante")
        else:
            return (aaaa,mm,jj)

def cal_perpetuel(date):
    """ Calcule le jour de la semaine correspondant à la date donnée sous
    le format 'aaaammjj' """
    jours_semaines = ['vendredi','samedi','dimanche','lundi', 'mardi', 'mercredi', 'jeudi']
    aaaa,mm,jj = convertit_date(date)
```

```
    rang = calcul_rang_depuis_15101582(jj,mm,aaaa)
    return jours_semaines[rang % 7]

print(cal_perpetuel('20141013'))
```

Le test pour le 13 octobre 2014, jour de rédaction de cette correction, donne la réponse `'lundi'`, qui est correcte.

Correction de l'exercice 8 – On commence par convertir en base 10. Étant donné un nombre x en base b dont les chiffres correspondent (de la droite vers la gauche) aux entiers $x_0, \ldots, x_k$, le nombre correspondant est :

$$x = \sum_{i=0}^{k} x_k b^k.$$

On est donc ramené à l'évaluation d'un polynôme, ce qu'on peut faire par la méthode de Hörner, consistant à utiliser la factorisation suivante :

$$x = x_0 + b(x_1 + b(x_2 + \cdots + b(x_{k-1} + bx_k))).$$

```
correspondance = '0123456789abcdef'

def conv_vers_base_10(n,b): # n entré sous forme de chaîne de caractères
    """convertisseur de la base b à la base 10, appliqué à l'entier n"""
    m = 0
    for c in n:           # on énumère les caractères de n
        i = correspondance.find(c)
        if (i < 0) or (i >= b):
            raise ValueError("Bases incohérentes")
        m = b * m + i
    return m
```

La variable `correspondance` permet de faire la corespondance entre l'écriture des chiffres (lettres) et la valeur de ces chiffres (position dans la chaîne de caractères).
Il nous reste à faire l'opération inverse de conversion de la base 10 vers une base quelconque, ce qu'on fait par divisions euclidiennes successives : le premier reste est le chiffre des unités, le deuxième reste le chiffre des b-aines etc.

```
def conv_depuis_base_10(n,b): # n entré sous forme de chaîne de nombre
    """convertisseur de la base 10 à la base b, appliqué à l'entier n"""
    m = ''
    if n == 0:
        return '0'
    while n > 0:
        n,i = divmod(n,b)
        m = correspondance[i] + m
    return m
```

La fonction `divmod` permet de ne calculer qu'une fois la division euclidienne (elle renvoie le couple formée du quotient et du reste). Sans cela, nous serions obligé de calculer 2 fois cette division euclidienne, une première pour la recherche du reste, une deuxième pour le calcul du quotient. Évidemment, cela serait moins efficace.
Il nous reste enfin à mettre bout-à-bout les 2 convertisseurs pour obtenir plus généralement un convertisseur entre deux bases quelconques.

```
def conv_b_vers_c(n,b,c):
    """ Convertit en base c la chaîne n, écrite en base b"""
    return conv_depuis_base_10(conv_vers_base_10(n,b),c)
```

Correction de l'exercice 9 –

1. On effectue la somme chiffre par chiffre en commençant par les chiffres de poids faible (les derniers dans l'écriture), et sans oublier de reporter les retenues. On a créé une chaîne de correspondance entre les chiffres utilisés et leurs valeurs. Étant donnée une valeur k d'un chiffre, on retrouve la lettre correspondant en considérant le k-ième caractère de la chaîne `corresp`. Réciproquement, étant donnée une lettre, on retrouve sa valeur en cherchant le premier (et seul) rang auquel il apparaît dans la liste.
   Remarquez l'usage d'une indexation négative de mes chaînes de caractères. Cela me permet de parcourir la chaîne à partir de la fin. En effet, `ch[-1]` donne le dernier caractère de la chaîne, `ch[-2]` l'avant-dernier etc. Cette indexation négative est aussi possible avec les listes.

```
corresp = '0123456789abcdefghijklmnopqrstuvwxyz'

def somme(x,y,b):
    """ Calcule la somme des entiers représentés en base b par les chaînes x et y"""
    if len(x)< len(y):          # ajout de 0 pour avoir des entiers de même taille
        for i in range(len(y)-len(x)):
            x = '0' + x
    elif len(x) > len(y):
        for i in range(len(x)-len(y)):
            y = '0' + y
    s = ''
    ret = 0   # retenue initiale
    for i in range(len(x)):
        (ret,ch) = divmod(corresp.find(x[-i-1]) + corresp.find(y[-i-1]) + ret,b)
              # calcul de la retenue et du chiffre
        s = corresp[ch] + s
    if ret != 0:
        s = corresp[ret] + s
    return(s)

print(somme('86','37',10))
print(somme('11001','1001',2))
print(somme('ab2','e',16))
print(somme('kjhzedv56','85ggkvz57726hja',36))
```

Les tests renvoient les valeurs :

```
123
100010
ac0
85ggkwjop6gkcog
```

qu'on vérifie être correctes.

2. On commence par la multiplication par un nombre à 1 chiffre, sur le même principe que pour la somme.

```
def produit_chiffre(x,ch,b):
    """calcule le produit x * ch ou ch est de longueur 1"""
    ret = 0
    p = ''
    val_ch = corresp.find(ch)
    for i in range(len(x)):
        (ret,chiffre) = divmod(corresp.find(x[-i-1]) * val_ch + ret,b)
        p = corresp[chiffre] + p
    if ret != 0:
        p = corresp[ret] + p
    return p

print(produit_chiffre('63','6',10))
print(produit_chiffre('12f','3',16))
```

Les deux tests effectués renvoient `378` et `38d`, ce qui est concluant. On passe maintenant au cas général, obtenu en multipliant successivement le premier nombre par chacun des chiffres du second, en décalant les résultats (ce qui revient à ajouter des 0) et en sommant :

```
def produit(x,y,b):
    """calcule le produit x * y en base b"""
    p = ''
    for i in range(len(y)):
        pr = produit_chiffre(x, y[-1-i],b)
        for j in range(i):
            pr = pr + '0'
```

```
        p = somme(p,pr,b)
    return p

print(produit('63','11',10))
print(produit('28817','1679',10))
print(produit('12f','3e',16))
```

Les trois tests renvoient 693, 48383743 et 4962, ce qui, après vérifications, est concluant.

Correction de l'exercice 10 –

1. On utilise les fonctions de conversion `conv_vers_base_10` et `conv_depuis_base_10` définies dans un exercice antérieur. On obtient alors, pas définition du complément à 2 :

```
def complete(code):
    l = len(code)
    for i in range(l,16):
        code = '0' + code
    return code

def compl2(n):
    if (n > 32767) or (n < -32768):
        raise ValueError("mauvais␣intervalle")
    if n >=0:
        return complete(conv_depuis_base_10(n,2))
    else:
        return complete(conv_depuis_base_10(n+2 ** 16,2))

print(compl2(100))
print(compl2(-100))
```

Les deux tests amènent : 0000000001100100 et 1111111110011100, ce qui est concluant.

2. Pour faire la conversion dans l'autre sens, on repère le signe grâce au premier bit :

```
def decode_compl2(code):
    n = conv_vers_base_10(code,2)
    if code[0] == '1':
        n -= 2**16
    return n

print(decode_compl2('0000000001100100'))
print(decode_compl2('1111111110011100'))
```

Les deux tests renvoient 100 et -100, ce qui est concluant.