Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
N. Carré, A. Troesch

2018/2019

TP n° 1 : Les bases de la programmation

Correction de l'exercice 1 – À la découverte des Variables

Une variable est la donnée d'une place en mémoire, destinée à stocker une valeur, et d'un nom permettant l'accès à cette place en mémoire. La valeur d'une variable peut changer au cours du temps. L'action de donner une valeur à une variable s'appelle « l'affectation », et s'effectue avec le signe =.

1. L'opération d'affectation n'est pas symétrique : la variable se trouve à gauche, la valeur à droite :

```
>>> x
3
>>> 3 = x
  File "<stdin>", line 1
SyntaxError: can't␣assign␣to␣literal
```

2. Une variable est un contenu. Définir une variable y par une expression en fonction d'une autre variable x se fait par le calcul de l'expression en x, avec le contenu de la variable x au moment de l'affectation. En aucun cas, une modification ultérieure de la valeur de x ne peut avoir d'effet sur y : le contenu de la variable y est une valeur et non une expression en x.

```
>>> y = 7 * x
>>> y
21
>>> x = 1
>>> y
21
```

3. • `help(id)` amène la page d'aide suivante :

```
Help on built-in function id in module builtins:

id(...)
    id(object) -> integer

    Return the identity of an object. This is guaranteed to be unique among
    simultaneously existing objects. (Hint: it's␣the␣object's memory address.)
(END)
```

   On y apprend que `id()` renvoie l'identifiant d'un objet (correspondant d'une façon ou d'une autreà l'adresse mémoire assignée à la variable)

   • De même help(type) nous apprend sans surprise que la fonction `type()` renvoie le type d'un objet. On y apprend aussi qu'on peut créer un nouveau type, mais cela ne nous concernera pas vraiment.

```
>>> id(x)
211273705440
>>> type(x)
<class 'int'>
```

4. Modifier la valeur de x modifie son identifiant, car une telle modification est en fait vue comme une nouvelle affectation (on redéfinit une nouvelle variable x). Toute opération sur une variable passant par une nouvelle affectation modifie son identifiant. On peut cependant modifier des variables sans affectation, en utilisant des méthodes associées à certains types. Ces méthodes ne modifient pas alors l'identifiant. Nous en verrons des exemples lorsque nous manipulerons des listes ou des chaînes de caractères.

```
>>> x += 1
>>> id(x)
211273705472
>>> type(x)
<class 'int'>
>>> x += 0.5
>>> id(x)
140638118502688
>>> type(x)
<class 'float'>
```

5. L'instruction `x += y` remplace le contenu de la variable x par la valeur de $x + y$. Remarquez que cela modifie comme précédemment l'identifiant de x (il s'agit d'une nouvelle affectation). Cette instruction est donc équivalente à $x = x + y$

```
>>> x = 3
>>> id(x)
211273705504
>>> x += 2
>>> id(x)
211273705568
>>> x
5
```

   De même pour les autres opérations :

```
>>> x = 3
>>> x -= 1
>>> x
2
>>> x *= 6
>>> x
12
>>> x /= 4
>>> x
3.0
```

6. On pourrait penser dans un premier temps faire :

```
>>> x = 2
>>> y = 5
>>> x = y
>>> y = x
>>> x;y
5
5
```

   C'est un échec : on n'a pas du tout fait l'échange des contenus de x et y ; les deux variables ont pris la valeur de y. En inversant les deux affectations, elles prennent toutes deux la valeur de x. En y réfléchisant un peu plus, on se rend facilement compte que c'est assez logique, les affectations se faisant successivement !

   Une méthode classique consiste à introduire une nouvelle variable, stockant provisoirement la valeur d'une des deux variables.

```
>>> x = 2
>>> y = 5
>>> z = x
>>> x = y
>>> y = z
>>> x ; y
```

```
5
2
```

On peut aussi chercher à effectuer les deux affectations simultanément. On pourrait penser à :

```
>>> x = y ; y = x
```

mais cela pose le même problème que précédemment, les deux instructions disposées sur la même ligne et séparées par un point-virgule étant effectuées successivement. On peut alors penser à :

```
x, y = y, x
```

et ça marche ! En fait, cette instruction est interprétée comme une unique affectation portant sur le couple (x, y).

Correction de l'exercice 2 – Premiers affichages

On répond en ligne de commande.

1. L'affectation d'une valeur à une variable se fait avec l'égalité simple =, en indiquant à gauche du signe le nom de la variable, et à droite la valeur (numérique, ou tout autre type disponible) :

   ```
   >>> Bonjour = 0
   ```

   Si on veut visualiser la valeur d'une variable, on tape son nom. L'ordinateur nous renvoie sa valeur :

   ```
   >>> Bonjour
   0
   ```

2. La fonction `print` convertit en chaînes de caractère les valeurs de ses différents paramètre afin faire un affichage à l'écran. L'instruction `print(Bonjour)` renvoie :

   ```
   >>> print(Bonjour)
   0
   ```

   En effet, ici, `Bonjour` désigne non pas la chaîne de caractère Bonjour (qui permettrait d'afficher le texte « Bonjour »), mais la variable définie précédemment. Une succession de caractère ne désigne une chaîne de caractères que si elle est entourée des délimiteurs convenables. Sinon, elle désigne le nom d'un objet ; ainsi, si la variable `Bonjour` n'avait pas été définie avant, nous aurions eu une erreur :

   ```
   >>> print(Bonjour)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   NameError: name 'Bonjour' is not defined
   ```

   Cette erreur possède un nom : `NameError` (le fait de connaître, et de pouvoir distinguer les noms d'erreur peut avoir un intérêt pour la gestion des exceptions, par la structure `try... except...`)

   Les délimiteurs de chaînes de caractère sont au nombre de 2 :
   - les apostrophes `'texte'`
   - les guillemets `"texte"`

   L'intérêt d'avoir plusieurs délimiteurs est de pouvoir plus facilement utiliser ces délimiteurs comme caractère du texte. Ainsi, `'Aujourd'hui'` n'est pas une syntaxe possible ; en revanche, on peut définir la chaîne `"Aujourd'hui"`. De même, on ne peut pas définir `"Il dit : "Bonjour"."`, mais `'Il dit : "Bonjour".'` convient.

   Un troisième délimiteur permet de gérer par exemple le cas du texte :

   `Il dit: "Il fait beau aujourd'hui".`

   Il s'agit du triple-guillemet `"""`. Ces triples-guillemets (définissant les raw-string) acceptent aussi les retours à la ligne par la touche « Entrée » au lieu du caractère spécial `\n`.

3. On obtient le type d'un objet par la fonction `type` :

   ```
   >>> type(Bonjour)
   <class 'int'>
   ```

Ainsi, la variable « Bonjour » est de type `'int'`, à savoir de type entier. La reconnaissance de type de la variable s'est fait automatiquement lors de l'affectation. Le type entier n'est en `Python3`, pas borné en taille, et comprend donc également le type entier long anciennement dans `Python2`. Pour la chaîne de caractère, on obtient

```
>>> type('Bonjour')
<class 'str'>
```

Ceci nous indique que le type d'une chaîne de caractères est `'str'`, pour `string`.

4. Nous avons déjà répondu plus haut :

```
>>> print("Aujourd'hui")
Aujourd'hui
```

```
>>> print('Il␣dit:␣"Bonjour"')
Il dit: "Bonjour"
>>> print("\"Aujourd'hui\"")
"Aujourd'hui"
```

Dans le dernier affichage, on distingue le caractère " du délimiteur de chaînes de caractères, en le faisant précéder du *backslash* \. On aurait pu s'en sortir avec une raw string, mais cela poserait en fait un problème d'espace, le premier et le dernier caractère d'une raw string ne pouvant pas être un guillemet normal.

5. Le caractère spécial `\n` est un retour à la ligne :

```
>>> print("Bonjour.\nAu␣revoir.")
Bonjour.
Au revoir.
```

6. Pour consulter l'aide associée à une fonction, on utilise la fonction `help`. L'instrution `help(print)`, renvoie par exemple :

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

On y apprend les différents paramètres utilisables avec `print`, ainsi que la syntaxe adaptée, et les valeurs par défaut (espace pour `sep`, retour à la ligne pour `end`).

```
>>> x = 1
>>> y = 2
>>> z = 3
>>> print(x,y,z,sep = '␣;␣')
1 ; 2 ; 3
>>> print(x,y,z,sep = '␣et␣')
1 et 2 et 3
>>> print(x,y,z, end = '.')
1 2 3.>>> print(x,y,z, end = '.\n')
1 2 3.
>>> print(x,y,z, sep = '␣;\n', end = '␣;\nCQFD\n')
1 ;
```

```
2 ;
3 ;
CQFD
```

Remarquez que sans retour à la ligne final, l'invite de commande se met à la suite du texte affiché, ce qui est un peu désagrable.

Correction de l'exercice 3 – On écrit un programme avec un éditeur de texte adapté (emacs, vi...), ou dans un environnement de programmation (Pyzo, Eclipse...) On peut dans le premier cas lancer l'exécution du programme en ligne de commande (par exemple `python3 essai.py` pour lancer l'exécution du programme `essai.py`), ou depuis l'environnement de programmation (depuis l'onglet `Run`). Il est recommandé, comme pour tout document informatique, de préciser quel est le langage utilisé en commentaire dans la première ligne. En Python, les commentaires se font en faisant précéder le texte du symbole ♯. Nous omettrons systématiquement cette étape. Depuis la version 3 de Python, il n'est plus nécessaire de spécifier l'encodage des caractères.

1. Nous avons là un programme très simple, d'une seule ligne :

   ```
   print('Bonjour')
   ```

   L'exécution de ce programme (que j'ai sauvegardé sous le nom `py011-1.py`) depuis un terminal donne :

   ```
   $ python3 py011-1.py
   Bonjour
   ```

2. Prenez l'habitude de commenter vos programmes. Cela peut faciliter une reprise ultérieure de votre code (par vous ou par quelqu'un d'autre). Un programme non commenté peut très vite devenir assez incompréhensible. Évidemment, ici, j'exagère un peu dans l'autre sens.

   ```
   # Définition de la fonction f
   def f(x):
       return 1 / (1 + x ** 2)

   # Demande d'une valeur x à l'utilisateur
   x = eval(input('Entrez une valeur de x : '))
   # Ne pas oublier de convertir cette entrée en valeur numérique grâce à eval

   #Affichage du résultat
   print('f({:g}) = {:g}'.format(x, f(x)))
   ```

   À nouveau, une exécution dans un terminal donne :

   ```
   $ python3 py011-2.py
   Entrez une valeur de x : 3
   f(3) = 0.1
   ```

Correction de l'exercice 4 – 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 5 –

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 6 – Formater l'affichage des nombres.

1. Pour commencer, vérifions que la fonction `str()` permet bien de faire une conversion d'un type numérique vers un type string :

   ```
   >>> x = 3
   >>> x
   3
   >>> str(x)
   '3'
   >>> 'Les␣' + x + '␣petits␣cochons'
   ```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't␣convert␣'int'␣object␣to␣str␣implicitly
>>>␣'Les '␣+␣str(x)␣+␣' petits cochons'
'Les 3 petits cochons'
>>>
```

Dans cet exemple, on essaie d'insérer la valeur de la variable x dans une chaîne de caractère, à l'aide de l'opération + (concaténation des chaînes de caractères). Si on essaye de le faire directement avec la valeur de x, on obtient une erreur de type, l'ordinateur ne pouvant effectuer la somme d'une chaîne de caractère et d'une valeur numérique. La conversion de x en chaîne de caractère résout le problème.

Le même exemple peut être traité à l'aide de la méthode `format`. Une méthode est une fonction définie pour une classe d'objets, dont la syntaxe est généralement une utilisation en suffixe : `objet.méthode(paramètres)`. La liste des méthodes disponibles pour une classe peut être obtenue à l'aide de la fonction `help()`. Ainsi, `help(str)` renvoie la liste des méthodes de la classe `str` des chaînes de caractères, liste dans laquelle nous retrouvons la méthode `format` définie dans l'énoncé. Nous obtenons la syntaxe suivante :

```
>>> 'Les␣{}␣petits␣cochons'.format(x)
'Les␣3␣petits␣cochons'
```

2. Nous testons ici les différents formats d'affichage décrits dans l'énoncé, avec la valeur $\frac{2}{7}$

```
>>> x = 2 / 7
>>> x
0.2857142857142857
>>> 'Que␣mon␣nombre␣{}␣est␣beau'.format(x)
'Que␣mon␣nombre␣0.2857142857142857␣est␣beau'
>>> 'Que␣mon␣nombre␣{:g}␣est␣beau'.format(x)
'Que␣mon␣nombre␣0.285714␣est␣beau'
>>> 'Que␣mon␣nombre␣{:3e}␣est␣beau'.format(x)
'Que␣mon␣nombre␣2.857143e-01␣est␣beau'
>>> 'Que␣mon␣nombre␣{:e}␣est␣beau'.format(x)
'Que␣mon␣nombre␣2.857143e-01␣est␣beau'
>>> 'Que␣mon␣nombre␣{:.3e}␣est␣beau'.format(x)
'Que␣mon␣nombre␣2.857e-01␣est␣beau'
>>> 'Que␣mon␣nombre␣{:.4f}␣est␣beau'.format(x)
'Que␣mon␣nombre␣0.2857␣est␣beau'
>>> 'Que␣mon␣nombre␣{:<15.4f}␣est␣beau'.format(x)
'Que␣mon␣nombre␣0.2857␣␣␣␣␣␣␣␣␣␣est␣beau'
>>> 'Que␣mon␣nombre␣{:>15.4f}␣est␣beau'.format(x)
'Que␣mon␣nombre␣␣␣␣␣␣␣␣␣␣0.2857␣est␣beau'
>>> 'Que␣mon␣nombre␣{:^15.4f}␣est␣beau'.format(x)
'Que␣mon␣nombre␣␣␣␣␣0.2857␣␣␣␣␣␣est␣beau'
```

La troisième tentative montre l'effet de l'oubli du point (simple passage en notation scientifique, mais on perd la spécification du nombre de décimales ; comme le montre la ligne suivante, cela revient au même que `:e`)

3. On calcule les valeurs de u_n les unes après les autres, en les stockant dans la même variable u qu'on actualise au fur et à mesure du calcul. Ne pas oublier d'importer la fonction sin du module `math` (ou de `numpy`).

```
from math import sin
u = 1
i = 0
while u > 2e-3:
    print('u_{}␣=␣{:.10f}'.format(i,u))
    u = sin(u)
    i += 1
```

La convergence est lente (en $\Theta(\sqrt{n})$), ce dont on peut se rendre compte en exécutant ce programme. si on remplace $2 \cdot 10^{-3}$ par 10^{-3}, il faut être beaucoup plus patient pour voir le programme terminer.

4. Soit on concatène les chaînes 3 par 3, soit on modifie le paramètre `end` de la fonction `print()`, afin de supprimer les retours à la ligne entre chaque affichage, en rajoutant un retour à la ligne (par concaténation) tous les 3 affichages. C'est cette dernière méthode que nous retenons :

```
n = 0
u = 1
while u < 1e20:
    affichage = '2^{:<3} = {:>20}'.format(n,u)
    if n % 3 == 0:
        affichage = '\n' + affichage
    print(affichage, end = ' ; ')
    u *= 2
    n += 1
```

L'alignement des valeurs par les unités se fait en justifiant à droite, la longueur totale réservée à l'affichage étant de 20 caractères, afin d'avoir la place d'afficher tous les nombres inférieurs à 10^{20}.

Correction de l'exercice 7 – 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 8 – 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 9 – La convergence de cette série n'est pas très dure à établir. Il s'agit en fait d'un cas particulier d'une situation plus générale (séries alternées). Un théorème donne alors la convergence du fait de la décroissance de $\dfrac{1}{k\ln(k)}$ vers 0. En effet, en notant $S_n = \displaystyle\sum_{k=2}^{n} \frac{(-1)^k}{k\ln(k)}$, les deux différences $S_{2n+2} - S_{2n}$ et $S_{2n+3} - S_{2n+1}$ sont toutes deux constituées de deux termes, et la décroissance de la valeur absolue du terme général permet de montrer que l'une est positive, et l'autre négative. La convergence du terme général vers 0 permet alors de montrer que (S_{2n}) et (S_{2n+1}) sont adjacentes, donc convergentes vers une même limite S. Il en résulte que (S_n) elle-même est convergente. Par ailleurs, des deux suites extraites (S_{2n}) et (S_{2n+1}), l'une converge en croissant vers S, l'autre en décroissant. On obtient alors facilement la majoration suivante :

$$\forall n \geqslant 2, \quad |S_n - S| \leqslant |S_n - S_{n+1}|.$$

Nous pouvons donc nous contenter de calculer la somme juqu'à ce que deux termes consécutifs aient une différence inférieure à l'erreur maximale souhaitée (ce qui revient à dire que la valeur absolue du terme général de rang $n+1$ est inférieur à cette erreur)

```
from math import log

def somme(err):
    # Initialisation:
    eps = 1             # signe
    S = 0               # Somme partielle initiale
    u = 1 / (2 * log(2)) # premier terme général
    i = 2               # premier indice
    # Itération:
    while abs(u) > err:
        S += u * eps
        i += 1
        u = 1 / (i * log(i))
        eps *= -1
    return S
```

Pour `err = 1e-8`, on obtient `S = 0.5264122415333191`.

Correction de l'exercice 10 – 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 11 – 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)
```