Corrigé du TP 03 Boucle

Chenevois-Jouhet-Junier-Rebout

1 Quelques boucles

1.1 Import de bibliothèques/modules

```
import math
```

1.2 Exercice 2

```
s = 0
for k in range(831, 945):
    s += k**10 #ou s = s + k**10

"""
>>> s
3672419115036510057216102022082 5
"""
```

1.3 Exercice 3

```
res = 1
for i in range(1, 844):
    res = res * 3

"""
>>> res-3**843
0
"""
```

1.4 Exercice 4

```
facto100 = 1
for i in range(2, 101):
    facto100 = facto100 * i
"""
>>> facto100
933262154439....16864000000000000000000000000
>>> import math
>>> facto100 - math.factorial(100)
```

```
0
"""
```

1.4.1 Exercice 5

Calcul du terme de rang n de $u(0) = 3$ et $u(n+1) = 3 \times u(n) + n$. On trouve $u(0) = 3$, $u(1) = 9$, $u(2) = 28$ ...

```
n = int(input('Entrez un entier n : '))
u = 3
for k in range(1, n+1):
    u = 3*u + k - 1
print('u({:d})={:d}'.format(n, u))
```

Soit la suite (v_n) définie par $v(0) = 1$ et pour tout entier $n \geqslant 0$, $v(n+1) = 1 + 2/v(n)$. Algorithme de seuil pour déterminer le plus petit indice p tel que $|v(p) - 2| < 10^{-6}$.

On trouve $v(21) = 2,00000072$

```
n = 0
v = 1
while abs(v - 2) >= 1e-6:
    n += 1
    v = 1 + 2.0/v
print('v({:d})={:1.8f}'.format(n,v))
```

1.5 Exercice 6 Projet Euler Problème 20

```
somme_dec = 0
n = math.factorial(100)
while n > 0:
    somme_dec += n%10
    n = n//10
print(somme_dec)

"""
>>> somme_dec
648
"""
```

1.6 Exercice 7

- Script 1 : la boucle externe est exécutée 100 fois, à chaque itération elle affiche une * et la boucle interne est exécutée une seule fois où elle affiche 100 '*' soit un total de 200 *
- Script 2 : la boucle externe est exécutée 100 fois, à chaque itération elle affiche une * et la boucle interne est exécutée 100 fois et à chaque itération de la boucle externe elle affiche 100 * soit un total de $100 + 100 \times 100 = 101 \times 100$ *
- Script 3: la boucle externe est exécutée 100 fois, à chaque itération elle n'affiche aucune * et la boucle interne est exécutée 100 fois à chaque itération de la boucle externe elle affiche 100 * soit un total de 100×100 *
- Script 4: la boucle externe est exécutée 50 fois, à chaque itération elle affiche une * et la boucle interne est exécutée 50 fois à chaque itération de la boucle externe elle affiche 50 * soit un total de $50 \times (1 + 50)$ *

1.7 Exercice 8

```
n = int(input('Entre le rang n : '))
f0, f1 = 0, 1
for k in range(1, n+1):
    f0, f1 = f1, f0+f1
print('fibo({:d})={:d}'.format(k,f0))

"""
>>> (executing cell "EXO 8 #########" (line 92 of "tp-python-sup-2015-03.py"))
Entre le rang n : 0
fibo(0)=0

>>> (executing cell "EXO 8 #########" (line 92 of "tp-python-sup-2015-03.py"))
Entre le rang n : 100
fibo(100)=354224848179261915075

>>> (executing cell "EXO 8 #########" (line 92 of "tp-python-sup-2015-03.py"))
Entre le rang n : 1000
fibo(1000)=43466557686937456435...5166849228875
"""
```

1.8 Exercice 9 : algorithme d'Euclide

- Algorithme des différences

```
def euclide_difference(a, b):
    while a != b:
        if b > a: #on doit avoir a >= b pour que a - b >= 0
            a, b = b, a
        a, b = b, a - b
    return a

"""
In [10]: [euclide_difference(30, k) for k in range(1, 20)]
Out[10]: [1, 2, 3, 2, 5, 6, 1, 2, 3, 10, 1, 6, 1, 2, 15, 2, 1, 6, 1]
"""
```

- Algorithme classique

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

"""
In [13]: [euclide_classique(30, k) for k in range(1, 20)]
Out[13]: [1, 2, 3, 2, 5, 6, 1, 2, 3, 10, 1, 6, 1, 2, 15, 2, 1, 6, 1]
"""
```

1.9 Exercice 10 : Projet Euler problème 9

```
a, trouve = 1, False

#a<b<c et a+b+c=1000 donc a<1000//3 et 1000 - (a+b) > b

while not trouve and a < 1000//3:
    b = a+1
    while not trouve and b < (1000 - a)//2:
        c = 1000 - a - b
        if c**2 == a**2 + b**2:
            trouve = True
            produit = a*b*c
        b += 1
    a += 1
print('{0:} + {1:}+ {2:} = 1000 et {0:}**2 + {1:}**2 = {2:}**2 \
et produit = {3:}'.format(a-1 , b-1 , c, produit))

#200 + 375+ 425 = 1000 et 200**2 + 375**2 = 425**2 et produit = 31875000
```

2 Autour des nombres premiers

2.1 Exercice 11

Dans une fonction si chaque clause se termine par un `return` on n'a pas besoin d'une structure `if elif else`. Si une clause est réalisée, le `return` fait sortir de la fonction et ce qui suit n'est pas exécuté.

```
from math import sqrt

def est_premier(n):
    '''test de primalité'''
    if n <= 1:
        return False
    for d in range(2, int(sqrt(n)) + 1):
        if n%d == 0:
            return False
    return True

for n in range(20):
    if est_premier(n):
        print(n, end=',')
```

2.2 Exercice 12

- Si n est pair, un ou deux tests sont effectués et une division euclidienne.
- Si n est premier, environ $\lfloor\sqrt{n}\rfloor$ tests et divisions euclidiennes sont effectués.

2.3 Exercice 13

- Pour les entiers premiers, le coût de traitement sera de l'ordre de : $N/(\ln(N)) \times \sqrt{N}$, soit $10^9/(6\ln(10))$ pour $N = 10^6$: c'est acceptable. Un microprocesseur à 2,5 GHz peut réaliser un milliard de cycles par seconde donc

si un cycle peut réaliser 4 unités de calculs en virgule flottantes, il peut réaliser 10 milliards de ces opérations par secondes soit 10 **gigaflops**. La puissance de calcul peut être estimée grossièrement par la formule *puissance = fréquence × nombre d'opérations simultanées × nombre de coeurs* Voir https://interstices.info/idee-recue-comparer-la-puissance-de-deux-ordinateurs-cest-facile/ pour plus de détails.

- C'est plus difficile à évaluer pour les composés : certains auront un coût proche de $\sqrt{N}$, mais ils seront peu nombreux. La plupart auront un premier diviseur faible, donc on peut espérer un coût qui soit plus proche de N ou $N\ln(N)$ que de $N^{3/2}$. Finalement le terme dominant dans la complexité est représenté par les entiers premiers.

2.4 Exercice 14

```
for nmax in [10**2, 10**4, 10**6]:
    cpt = 0
    for n in range(1,nmax+1):
        if est_premier(n):
            cpt += 1
    print(' {:d} entiers premiers <= {:d}'.format(cpt, nmax))
```

- 25 entiers premiers $\leqslant 100$
- 1229 entiers premiers $\leqslant 10000$
- 78498 entiers premiers $\leqslant 1000000$

2.5 Exercice 15

```
cpt = 0
for n in range(2, 10**6-1):
    if est_premier(n) and est_premier(n+2):
        cpt += 1
print(cpt)
```

8169 couples de premiers jumeaux inférieurs à 10**6

2.6 Exercice 16

```
n = 10**10+1
while not est_premier(n) or not est_premier(n+2):
    n += 1
print("Le plus petit couple d'entiers premiers jumeaux \
supérieurs à 10**10 est ",(n,n+2))
```

Le plus petit couple d'entiers premiers jumeaux supérieurs à 10^{10} est $(10000000277, 10000000279)$.

2.7 Exercice 17

```
def est_premier2(n):
    '''test de primalité'''
    global div  #compteur de divisions, variable globale pour l'exo 17
    if n <= 1:
        return False
    if n <= 3:
```

```
        return True
    for d in range(2, int(n**(1/2))+1):
        div += 1
        if n%d == 0:
            return False
    return True

cpt = 0 #compteur de couples de premiers jumeaux
div = 0 #compteur de divisions
for n in range(2,10**6):
    if est_premier2(n):
        cpt += 1
print('Il y a {} entiers  premiers  inférieurs à 10**6'.format(cpt))
print(div,'divisions ont été effectuées.')
```

Il y a 78498 entiers premiers inférieurs à 10^6 et 67740403 divisions ont été effectuées.

3 Observons une suite d'entiers

3.1 Exercice 18

```
def u(n):
    u = 42
    for k in range(1, n+1):
        u = (15091*u) % 64007
    return u


for n in [1, 10, 10**6]:
    print('u(%s)=%s'%(n,u(n)))
```

u(1)=57759 , u(10)=15421 et u(1000000)=14918.

3.2 Exercice 19

Pour éviter de répéter les memes calculs mieux vaut dresser une table de tous les premiers inférieurs à 64007 plutot que faire 10^7 tests de primalité. Quand on doit utiliser les mêmes données un grand nombre de fois dans une boucle, mieux vaut calculer ces données une seule fois en dehors de la boucle.

```
premier = [est_premier(k) for k in range(64007)]

"""
>>> print(premier[:8])
[False, False, True, True, False, True, False, True]
"""

c = [0]*6  #compteurs des six conditions
u = 42
for n in range(1, 10**7+1):
    u = (15091*u)%64007
    if u%2 == 0:
        c[0] += 1
```

```
    if premier[u]:
        c[1] += 1
    if u%3 == 1:
        c[2] += 1
    if u%3 == 1 and premier[u]:
        c[3] += 1
    if u%2 == 0 and premier[u]:
        c[4] += 1
    if n%2 == 0 and premier[u]:
        c[5] += 1
print(c)

"""
[4999968, 1001923, 3333434, 499050, 156, 493240]
"""
```

On n'a pas utilisé de `if elif else` car les clauses ne sont pas exclusives.

3.3 Exercice 20

```
"""
>>> ((17%10)*(923%10))%10
1
>>> ((123345678987654%10)*(836548971236%10))%10
4
"""
```

3.4 Exercice 21

```
res = 1
a = 123456
c = 1234567
for i in range(1, 654322):
    res = (res*a)%c
print(res)
"""
1075259
"""

print("{:d} est plus rapide à calculer que {:d}".format(res,(123456**654321)%1234567))
```

1075259 est plus rapide à calculer que 1075259.

3.5 Exercice 22 : Projet Euler problème 48

```
series, m  = 0, 10**10
for i in range(1,1001):
    res = 1
    for k in range(1, i+1):
        res = (res*i)%m
    series = (series+res)%m
print(series)
```

Réponse : 9110846700

4 Pour ceux qui s'ennuient

4.1 Exercice 23 Tombola

```
def gagnant(m):
    while m%9 == 0:
        m = m // 9
    return m%3 == 0

"""
>>> list(map(gagnant, [100,300,729,2016]))
[False, True, False, False]
"""

def nombre_gagnant(n):
    '''Retour le nombre d'entiers gagnants parmi
    les entiers entre 1 et n'''
    # on détermine d'abord le plus grand entier k tel que 3**k <= m
    p = 1
    e = 0
    while p <= n:
        p *= 3
        e += 1
    e = e - 1  #3**e <= m et 3**(e+1)>m
    gagnant = 0  #compteur de gagnant
    for k in range(1, e + 1):
        #pour les 3**k avec k impair
        #on rajoute le nombre de multiples de 3**k <= m
        # de la forme a*3**k avec a pas divisible par 3
        if k%2 == 1:
            q = n//3**k
            gagnant += q - q//3
    return gagnant

def nombre_gagnant2(m):
    '''Plus court mais plus long car complexité cachée'''
    return len([m  for n in range(1 , m + 1) if gagnant(m)])


from timeit import timeit
from math import log
```

La fonction `timeit` du module `timeit` affiche le temps cumulé d'exécution d'une commande pour un nombre fixé d'exécutions.

On observe ci-dessous que le temps d'exécution de `nombre_gagnant(n)` semble proportionnel à $\log(n)/\log(3)$ et que celui de `nombre_gagnant2(n)` semble proportionnel à `n`.

On peut conjecturer une **complexité temporelle** logarithmique pour `nombre_gagnant` et linéaire pour `nombre_gagnant2`.

```
bench = [10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8, 10**9]
for n in bench:
    temps = timeit('nombre_gagnant({})'.format(n),
                   'from __main__  import nombre_gagnant',number = 10000)
    print(log(n,3)/temps)

"""
130.16506757074666
158.44562663545946
157.35055637870508
169.62033933138346
167.83313335771103
162.74572022382512
168.88623818133289
161.82620193271757
"""

bench = [10**2, 10**3, 10**4]
for n in bench:
    temps = timeit('nombre_gagnant2({})'.format(n),
                     'from __main__  import nombre_gagnant2',number = 10000)
    print(n/temps)

"""
259.88979357801304
254.6282087040769
"""
```

4.2 Exercice 24 : Projet Euler problème 39

On recherche le nombre maximal de configurations de triangles rectangles de périmètre donné parmi tous les périmètres inférieur ou égal à n.

- Algorithme 1 :

  3 boucles imbriquées : sur le perimetre puis sur les côtés a, b et c avec $a \leqslant b \leqslant c$ où $c = p - a - b$.

  La complexité est **cubique** en $O(n^3)$.

```
def euler39_cubique(n):
    pmax, nbsolmax = 0, 0
    for p in range(4, n + 1):
        nbsol = 0
        for a in range(1, p//3 + 1):
            for b in range(a, 2*p//3 + 1):
                if (p - a -b)**2 == a**2 + b**2:
                    nbsol += 1
        if nbsol > nbsolmax:
            nbsolmax, pmax = nbsol,p
    return pmax, nbsolmax

def euler39_cubique2(n):
    pmax, nbsolmax = 0, 0
    for p in range(3, 1001):
```

```
        nbsol = 0
        for c in range(p//3, p):
            b = int(c/2**0.5)
            while b < c and p > b + c:
                a = p - c - b
                if a**2 + b**2 == c**2:
                    nbsol += 1
                b += 1
        if nbsol > nbsolmax:
            pmax, nbsolmax = p, nbsol
    return pmax, nbsolmax
```

- Algorithme 2 :

  On crée un tableau des nombres de solutions pour les n périmètres. On le crible avec deux boucles imbriquées qui parcourent tous les couples avec $a \leqslant b \leqslant 2n/3$ tels que $\sqrt{a^2+b^2}$ est un entier inférieur ou égal à n.

  La complexité est **quadratique** en $O(n^2)$.

```
def euler39_quadratique(n):
    #cribleperi[p] devra contenir le nombre de solutions pour le périmètre p
    cribleperi = [0]*(n+1)
    for a in range(1, n//3 + 1):
        for b in range(a, 2*n//3 + 1):
            c = (a**2 + b**2)**(1/2)
            if c == int(c):
                p = a + b + int(c)
                if p <= n:
                    cribleperi[p] += 1
    return max(enumerate(cribleperi), key = lambda couple : couple[1])

"""
In [1]: euler39_cubique(1000)
Out[1]: (840, 8)

In [2]: euler39_quadratique(1000)
Out[2]: (840, 8)

In [3]: %timeit euler39_quadratique(1000)
10 loops, best of 3: 184 ms per loop

In [4]: %timeit euler39_cubique(1000)
1 loops, best of 3: 53.4 s per loop

In [5]: 52.4/184e-3
Out[5]: 284.7826086956522
"""
```

On a vérifié ci-dessus que le rapport de temps entre les deux algorithmes était de l'ordre de grandeur de n (ici 1000) à une constante près.