Cours et Exercices de Python

niveau 1ère et Tale S

Propositions de correction

par

Philippe Moutou

Les questions algorithmiques envisagées ici sont résolues en langage Python 3. Le choix est un peu arbitraire car nous aurions tout aussi bien pu utiliser un autre langage (Ruby, C++, Java, etc.) ou une autre version de Python (la version 2 est encore utilisée) et parfois, sans doute, la solution tableur pourrait être essayée (elle a ses avantages) comme, ne l'oublions pas, la calculatrice qui est une nécessité dans certaines occasions (le bac par exemple) et dont il faut, parallèlement, poursuivre l'apprentissage. Mais comme il faut bien choisir, et que le choix de Python se révèle, dans bien des situations, un bon choix, alliant simplicité et richesse, qui sera un atout pour ceux qui en continueront l'apprentissage (en classes prépas notamment), nous en resterons là pour nos justifications.

Sur le fond, nous avons fait le choix de certains sujets au détriment d'autres qui auraient pu se révéler tout aussi intéressants, voire davantage. Ces choix ont été notamment dictés par les points de programmation que nous voulions illustrer, et aussi par des goûts personnels. Les notions abordées ne sont pas forcément toutes dans les programmes du Lycée mais elles sont compréhensibles, du moins nous l'espérons, avec les connaissances et le niveau de culture générale et mathématique auquel veut se situer ce cours. Comme il ne s'agit finalement que de programmation, le fond est secondaire, même si, dans nos motivations secrètes, nous cherchons à créer un intérêt pour ce que l'on programme. Les programmeurs professionnels sont souvent obligés de traiter des sujets qui ne les amusent pas beaucoup, mais ici, nous avons la liberté de choisir nos sujets, alors ne nous privons pas de découvrir quelques perles...

Un dernier point avant de commencer : nos propositions de correction ne sont que ce qu'elles sont. Il y a sûrement mieux. Plus court, plus efficace, plus élégant... Nous avons cherché à répondre aux questions posées avec un minimum de connaissances sur le langage Python et sur la programmation en général. Parfois la simplification d'un programme nécessite de le réécrire complètement alors que la complexification est le résultat d'ajouts progressifs (pour corriger des défauts ou pour ajouter des fonctionnalités). Nos corrections se situent entre ces deux extrêmes, et elles fonctionnent (ce qui est tout de même le but recherché). Nous invitons nos lecteurs à nous faire part de leurs commentaires et suggestions et leur souhaitons un bonne lecture active.

P. Moutou

Contenu

Partie n°1 : Primalité

a) Écrire un programme qui détermine si un nombre entier *n* est premier (si *n* n'a que deux diviseurs) ou s'il est composé. Dans ce dernier cas, donner la décomposition en facteurs premiers de *n*.

Pour déterminer si un nombre entier *n* est premier, il suffit de diviser ce nombre, successivement, par tous les entiers $a \geq 2$. Si une seule de ces divisions tombe juste alors le nombre est composé et on a déjà un des facteurs de la décomposition, sinon le nombre est premier. Cet algorithme n'est pas très raffiné car il oblige à faire des divisions inutiles : pourquoi tenter la division par 4 si la division par 2 ne tombe pas juste ? Mais, néanmoins, nous allons essayer ce programme, ne serait-ce que pour en mesurer les performances.

```
n=int(input(" Saisissez un nombre : "))
isPrem=True
listDiviseur=list()
for i in range(2,n):
    if n%i==0 :
        isPrem=False
        listDiviseur.append(i)   # liste des diviseurs de n
if isPrem : print("oui, "+str(n)+" est un nombre premier.")
else :
    s=decomposition(n,listDiviseur)
    print("non, "+str(n)+" n'est pas un nombre premier.")
    print("Voici sa décomposition :"+s)
```

```
 Saisissez un nombre : 7001
oui, 7001 est un nombre premier.

 Saisissez un nombre : 7003
non, 7003 n'est pas un nombre premier.
Voici sa décomposition :7003=47e1×149e1

 Saisissez un nombre : 7007
non, 7007 n'est pas un nombre premier.
Voici sa décomposition :7007=7e2×11e1×13e1
```

```
def decomposition(n,L) :# Décompose n en facteurs premiers
    s=str(n)+"="        # à partir de la liste de ces diviseurs
    decompo=list()
    Lpremier=[L[0]]     # liste contenant les facteurs premiers
    while len(L)!=0 :
        L=[a for a in L if a%Lpremier[-1]!=0] # expurgation de la liste
        if len(L)!=0 :Lpremier.append(L[0])   # des diviseurs
    i=0
    while n>1 :
        expo=0
        while n%Lpremier[i]==0 : # recherche de l'exposant pour
            expo+=1              # chaque facteur premier
            n//=Lpremier[i]
        decompo.append([Lpremier[i],expo])
        i+=1
    for rang,facteur in enumerate(decompo) :
        s+=str(decompo[rang][0])+"e"+str(decompo[rang][1])
        if rang<len(decompo)-1 :
            s+='\u00D7'          # écriture de la décomposition
    return s                     # dans une chaîne de caractères
```

L'algorithme principal (à gauche) examine si le nombre est premier et, s'il ne l'est pas, il stocke dans une liste les diviseurs. Cette liste est ensuite envoyée à la fonction *decomposition*() qui récupère tout d'abord les facteurs premiers. On stocke pour ce faire, le 1er diviseur (le plus petit, qui est forcément premier) dans une autre liste, et on expurge la liste des diviseurs de tous les diviseurs qui sont obtenus pas composition de ce facteur premier. On recommence jusqu'à avoir vidé la liste des diviseurs. On cherche alors la multiplicité (l'exposant) de chacun des diviseurs premiers.

Cet algorithme est très simple − il colle naïvement à la définition des nombres premiers − mais très laborieux. Il effectue *n* divisions : beaucoup de travail inutile. Testons-le avec un nombre plus grand que 7007 : pour $M_{11}=2^{11}-1=2047$ la réponse est instantanée, pour $M_{23}=2^{23}-1=8388607$, c'est beaucoup moins efficace : environ 8 secondes et pour $M_{29}=2^{29}-1=536870911$, il ne faut pas moins de 8 minutes. Ce nombre est 64 fois plus grand que le précédent, et il faut environ 64 fois plus de temps (il y a juste un peu plus de 536 millions de divisions à faire dans la partie principale du programme).

```
 Saisissez un nombre : 8388607
non, 8388607 n'est pas un nombre premier.
Voici sa décomposition :8388607=47e1×178481e1
```

8 secondes

```
 Saisissez un nombre : 536870911
non, 536870911 n'est pas un nombre premier.
Voici sa décomposition :536870911=233e1×1103e1×2089e1
```

8 minutes

La première amélioration va venir du fait que toutes les divisions ne sont pas nécessaires : s'il y a un facteur qui divise le nombre, ce facteur est forcément plus petit ou égal à la racine carrée du nombre . Car si $n=a\times b$, alors soit *a* soit *b* est inférieur ou égal à $\sqrt{n}$. On peut aussi supprimer tous les nombres pairs (sauf 2). Ces deux simples idées conduit à une amélioration qui diminue le nombre d'opérations à faire d'une manière très efficace. Pour examiner la primalité de M_{29}, on n'effectue les divisions que jusqu'à $\sqrt{M_{29}}\approx 23170$, et encore, on n'en effectue que la moitié, il y a donc environ 11585 divisions à effectuer au lieu de 536870911, ce qui revient à diviser le temps d'exécution d'un facteur 46342 environ. Les 8 minutes deviennent un centième de seconde !

Avec cette amélioration, il y a une adaptation à faire pour la fonction *decomposition*() qui prend en argument la liste des diviseurs du nombre *n* : en n'effectuant pas toutes les divisions, cette liste ne peut pas être complète ! Pour la compléter, nous avons mis au point la nouvelle fonction *complement*() qui ajoute les grands diviseurs (supérieurs à $\sqrt{n}$) obtenus en divisant *n* par la liste des premiers diviseurs. La liste complète des diviseurs a besoin d'être triée pour les besoins de la fonction *decomposition*(), nous utilisons donc cette méthode *sort*() de la classe *list*. Cette amélioration considérable ne suffit pourtant pas pour examiner, en temps raisonnable, la primalité de nombres beaucoup plus grands que M_{29}.

Pour commencer, notre instruction Ldiviseur=[2]+*list*(*range*(3,*int*(nRacine),2)) ne semblant pas plaire à Python pour certains grands nombres comme $M_{101}=2^{101}$ qui contient 31 chiffres (la liste à créer est trop

grande pour la mémoire), il fallait contourner ce problème technique. Une boucle *while* fait aussi bien l'affaire que cette liste et, cette fois, Python peut exécuter le programme mais sans nécessairement aboutir : il y a environ 796 131 459 065 721 opérations à effectuer pour examiner la primalité de M_{101} et 8 minutes ne suffisent pas, loin de là. Si on fait le calcul, le processeur de notre ordinateur effectuant environ 1 118 481 opérations par secondes, il faudrait environ 542 années... Un peu trop long à attendre. Soyons plus raisonnable et examinons le cas de $M_{43}=2^{43}-1=8796093022207$ qui ne nécessite que 1 482 910 opérations : cette fois 1,4 secondes suffisent.

amélioration 1 :

forme utilisable pour les petits nombres

```
def complement(n,L) :
    for diviseur in L :
        nouvDiv=n//diviseur
        if nouvDiv not in L : L.append(n//diviseur)
    return L

... même début que le programme initial ....
Ldiviseur=[2]+list(range(3,int(nRacine),2))
for i in Ldiviseur:
    if n%i==0 :
        isPrem=False
        listDiviseur.append(i)
if isPrem : print("oui, "+str(n)+" est un nombre premier.")
else :
    listDiviseur=complement(n,listDiviseur)
    listDiviseur.sort()
    s=decomposition(n,listDiviseur)
    ... même fin que le programme initial ....
```

```
Saisissez un nombre : 536870911
non, 536870911 n'est pas un nombre premier.
Voici sa décomposition :536870911=233e1×1103e1×2089e1
```

0.01 seconde

forme utilisable pour les grands nombres

```
... même début que le programme initial ....
if n%2==0 :
    isPrem=False
    listDiviseur.append(2)
i=3
while i<=nRacine :
    if n%i==0 :
        isPrem=False
        listDiviseur.append(i)
    i+=2
if isPrem : print("oui, "+str(n)+" est un nombre premier.")
else :
    listDiviseur=complement(n,listDiviseur)
    listDiviseur.sort()
    s=decomposition(n,listDiviseur)
    ... même fin que le programme initial ....
```

```
Saisissez un nombre : 8796093022207
non, 8796093022207 n'est pas un nombre premier.
Voici sa décomposition :8796093022207=431e1×9719e1×2099863e1
```

1.40 secondes

b) Utiliser ce programme pour montrer qu'à partir de k=5 les nombres de Fermat $F_k=2^{2^k}+1$ ne sont pas premiers.

```
oui, F1 = 5 est un nombre premier.
oui, F2 = 17 est un nombre premier.
oui, F3 = 257 est un nombre premier.
oui, F4 = 65537 est un nombre premier.
non, F5 = 4294967297 n'est pas un nombre premier.
Voici sa décomposition :4294967297=641e1×6700417e1
non, F6 = 18446744073709551617 n'est pas un nombre premier.
Voici sa décomposition :18446744073709551617=274177e1×67280421310721e1
```

Ces nombres, Pierre de Fermat (1601-1665) pensait qu'ils étaient tous premiers. Il avait calculé les quatre premiers et, effectivement, les nombres 5, 17, 257 et 65537 sont tous premiers (on peut ajouter le nombre $F_0=2^0+1=3$ qui est premier aussi). La conjecture qu'il énonça alors (en 1640) s'est avérée fausse car le 5ème nombre n'est pas premier. Fermat ne l'avait pas vérifié, il faut dire que 4 294 967 297 est un grand nombre et que son premier facteur premier est assez grand aussi (c'est 641, le 116ème nombre premier, il fallait faire 116 divisions...). C'est Euler qui montra, en 1732, que ce nombre $F_5=2^{2^5}+1=2^{32}+1$ est composé $F_5=641\times 6\,700\,417$. Pour prouver cela, Euler démontra ce théorème : tout facteur premier d'un nombre de Fermat F_n est de la forme $k\,2^{n+1}+1$ où k est un entier. Ainsi, pour F_5, il suffisait de diviser par des nombres de la forme $64k+1$, et pour k=10, il trouva le premier diviseur de F_5. Quant aux autres nombres de Fermat, on en cherche encore des premiers (de F_5 jusqu'à F_{32} ils sont tous composés). Voyons cela à l'aide de notre programme, en l'adaptant au problème : une petite boucle supplémentaire nous évitera d'entrer les valeurs successives de k. Nous n'irons pas bien loin car les nombres croissent très vite et pour k=6, le nombre a 20 chiffres, ce qui pose un problème de temps (plus de 45 minutes de calcul).

```
oui, M2 = 3 est un nombre premier.
oui, M3 = 7 est un nombre premier.
oui, M5 = 31 est un nombre premier.
oui, M7 = 127 est un nombre premier.
non, M11 = 2047 n'est pas un nombre premier.
Voici sa décomposition :2047=23e1×89e1
oui, M13 = 8191 est un nombre premier.
oui, M17 = 131071 est un nombre premier.
oui, M19 = 524287 est un nombre premier.
non, M23 = 8388607 n'est pas un nombre premier.
Voici sa décomposition :8388607=47e1×178481e1
non, M29 = 536870911 n'est pas un nombre premier.
Voici sa décomposition :536870911=233e1×1103e1×2089e1
oui, M31 = 2147483647 est un nombre premier.
non, M37 = 137438953471 n'est pas un nombre premier.
Voici sa décomposition :137438953471=223e1×616318177e1
non, M41 = 2199023255551 n'est pas un nombre premier.
Voici sa décomposition :2199023255551=13367e1×164511353e1
non, M43 = 8796093022207 n'est pas un nombre premier.
Voici sa décomposition :8796093022207=431e1×9719e1×2099863e1
non, M47 = 140737488355327 n'est pas un nombre premier.
Voici sa décomposition :140737488355327=2351e1×4513e1×13264529e1
non, M53 = 9007199254740991 n'est pas un nombre premier.
Voici sa décomposition :9007199254740991=6361e1×69431e1×20394401e1
non, M59 = 576460752303423487 n'est pas un nombre premier.
Voici sa décomposition :576460752303423487=179951e1×3203431780337e1
oui, M61 = 2305843009213693951 est un nombre premier.
```

c) Montrer, de même, que pour k=2, 3, 5, 7 les nombres de Mersenne, notés $M_k=2^k-1$, avec k premier, sont premiers (on les note alors *M1, M2, M3* et *M4*), alors que pour k=11, M_k n'est pas premier. Déterminer la valeur de k pour le 5ème nombre de Mersenne premier, noté *M5*.
Nous avons déjà examiné la primalité de quelques nombres de Mersenne. Pour les examiner dans l'ordre, nous allons adapter notre boucle pour les tester tous à partir de k=2, 3, 5, etc. Pour ce faire, nous écrivons la liste *Lprem* des nombres premiers jusqu'à 101 et nous utilisons cette fois une boucle for k *in* Lprem. Les premiers résultats sont très rapides à s'inscrire, mais, dès que les nombres sont plus grands, les temps de calcul s'allongent : c'est quasi-instantané jusqu'à $M_{37}=2^{37}-1=137\,438\,953\,471$, mais il faut une vingtaine de minutes pour aller jusqu'à la primalité de $M_{59}=2^{59}-1=576460752303423487$, donc nous ne pouvons pas espérer aller beaucoup plus loin avec cet algorithme.
Quoi qu'il en soit, contrairement à ce qu'avait conjecturé Marin Mersenne à l'époque de Fermat, les nombres dits « de Mersenne » ne sont pas tous premiers. Le premier qui ne l'est pas est relativement petit puisqu'il s'agit de $M_{11}=2^{11}-1=2047=23\times89$. C'est encore Euler qui démonta en 1732 cette fausse conjecture. Il donna aussi d'autres contre-exemples : M_{23}, M_{83}, M_{13}, M_{179}, M_{191} et M_{239} et prouva par contre que M_{31} était premier (c'était le 8ème nombre de Mersenne premier connu). Pourquoi s'intéressait-on tant aux nombres premiers de Mersenne à cette époque ? Parce qu'ils étaient liés aux nombres parfaits par la propriété suivante, connue depuis Euclide, au IVème siècle avant J.-C. : si $M=2^n-1$ est premier alors $\frac{M(M+1)}{2}=2^{p-1}(2^p-1)$ est un nombre parfait (l'intérêt pour les nombres parfaits est moins évident à comprendre). Ainsi, puisque $M4=M_7=127$ est premier, alors $\frac{127\times128}{2}=64\times127=2^6(2^7-1)=8128$ est un nombre parfait (le premier nombre parfait est 6=1+2+3, il correspond à $M1=M_2=3$). En effet, ses diviseurs propres sont 1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032 et 4064 et leur somme est égale au nombre lui-même :
1+2+4+8+16+32+64+127+254+508+1016+2032+4064=8128.
Jusqu'à l'avènement des ordinateurs, au milieu du XXème siècle, on découvrit jusqu'à 12 nombres de Mersenne premiers $M12=M_{127}$ (c'est un nombre qui s'écrit avec 39 chiffres). Aujourd'hui, on continue à s'intéresser aux grands nombres de Mersenne premiers car ils sont utiles en cryptographie. Le plus grand nombre premier connu est un nombre de Mersenne, il s'agit de $M48=M_{57885161}$ (il s'écrit avec 17 425 170 chiffres) qui fut découvert en 2013 par un projet de recherche collaborative, la GIMPS, qui élevait pour la 14ème fois le record du nombre premier le plus grand en seulement 17 ans d'existence.

d) Il est possible de se focaliser sur la primalité des nombres de Mersenne car un théorème dû au mathématicien français Lucas (1842-1891), permet d'accroître notablement l'efficacité du test : étant donnée la suite (s_n) définie par s_1=4 et, pour tout n>1, $s_n=(s_{n-1})^2-2$, le nombre $M_n=2^n-1$ est premier si et seulement si il divise s_{n-1}. La difficulté ici va être la croissance extrêmement rapide du nombre de chiffres des termes de la suite (s_n). Par exemple, pour tester si $M_{11}=2^{11}-1=2047$ est premier, il suffit de tester si ce nombre divise s_{10}. Le problème est que s_{10} s'écrit avec près de 300 chiffres ! Essayer de voir jusqu'où, avec Python sur une machine ordinaire et dans des temps raisonnables, cette simple suite permet d'examiner la primalité des nombres de Mersenne.
Pour ce programme nous allons oublier la décomposition des nombres de Mersenne non premiers, car la liste des diviseurs n'est pas accessible si on s'en tient au seul critère de Lucas. Nous nous bornons à calculer les termes de la suite (s_n) et à chercher le reste de la division de s_{n-1} par M_n. Si ce reste est nul, le nombre est premier. Lucas, à son époque sans ordinateur, parvint à prouver la primalité de M_{127} qui s'écrit avec 39 chiffres, sans faire le calcul explicite de s_{126}. Mais nous allons faire les calculs, enfin Python va s'en charger. Le programme est court, la difficulté est d'insérer une boucle pour le calcul de s_{n-1}. On doit veiller à utiliser les bons indices. Une mise au point est parfois utile : en affichant le terme utilisé de cette suite qui commence par 4, 14, 194, 37 634, 1 316 317 954, etc. on s'aperçoit d'une erreur éventuelle. Une remarque : pour n=2, le test ne fonctionne pas, car 4 n'est pas divisible par 3.
La bonne formulation du théorème de Lucas serait : « pour n>2, le nombre $M_n=2^n-1$ est premier si et seulement si il divise s_{n-1} » (ou bien pour n impair...). Avec ce test fabuleux, qui permet d'aller si loin dans l'étude de la primalité des nombres de Mersenne, nous n'avons pas réussi à aller plus loin que M_{23} ! Le nombre s_{22} utilisé possède déjà un million deux cent mille chiffres... Sans doute qu'il faudrait procéder autrement pour améliorer cette bien piètre performance.

```
Lprem=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,\
       53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101] #quelques nombres premiers
s=4               # terme s(1)
i=1
for k in Lprem :
    n=2**(k)-1
    while i<k-1 : # calcul du terme s(k-1) de la suite
        s=s**2-2
        i+=1
    isPrem=True
    if s%n!=0 : isPrem=False
    if isPrem : print("oui, M"+str(k)+" = "+str(n)+" est un nombre premier.")
    elif n==2 : print("oui, M2 = 3 est un nombre premier.")# petite exception au test de Lucas
    else :
        print("non, M"+str(k)+" = "+str(n)+" n'est pas un nombre premier.")
        print("car s"+str(i)+" qui s'écrit avec "+str(len(str(s)))+" chiffres, n'est pas un multiple de "+str(n))
```

```
    s10=68729682406644277238837486231747530924247154108646671752192618583088487405790957964732883069102561043436779663935595172042357306594916344606074564712868078287608055203024658359439017580883910978666185875717415541084494926500475167381168505927378181899753839260609452265365274850901879881203714
```

```
non, M2 = 3 n'est pas un nombre premier.
car s1 qui s'écrit avec 1 chiffres, n'est pas un multiple de 3
oui, M3 = 7 est un nombre premier.
oui, M5 = 31 est un nombre premier.
oui, M7 = 127 est un nombre premier.
non, M11 = 2047 n'est pas un nombre premier.
car s10 qui s'écrit avec 293 chiffres, n'est pas un multiple de 2047
oui, M13 = 8191 est un nombre premier.
oui, M17 = 131071 est un nombre premier.
oui, M19 = 524287 est un nombre premier.
non, M23 = 8388607 n'est pas un nombre premier.
car s22 qui s'écrit avec 1199461 chiffres, n'est pas un multiple de 8388607
non, M29 = 536870911 n'est pas un nombre premier.
```

Cette méthode naïve d'application du test de Lucas-Lehmer ne marche pas car les nombres sont trop grands. Pour la faire fonctionner, on n'a pas réellement besoin de calculer ces nombres. Puisqu'on ne se sert que du reste de la division de s_{n-1} par le nombre M_n, on peut se contenter des restes de la division des termes de la suite (s_n) par le nombre M_n. Justifier que le reste final (celui de s_{n-1} par le nombre M_n) sera bien le même si on garde les vrais termes de la suite (s_n) ou si on ne conserve que les résidus modulo M_n de ces termes n'est pas notre propos ici. Mais cette propriété va nous permettre d'appliquer efficacement le test. Nous pouvons tout de même essayer de comprendre ce qui se passe, en examinant un exemple : pour examiner la primalité de $M_7=127$, on doit examiner la divisibilité de s_6=2 005 956 546 822 746 114 par 127. Les termes précédents de la suite sont 4, 14, 194, 37 634 et 1 416 317 954. Si on examine la suite des restes (résidus) de la division par 127, on obtient 4, 14, 67 (194−127), ensuite on calcule $67^2-2=4\,487$ et le reste de 4 487 par 127 est 42 comme celui de 37 634 par 127. Ensuite on calcule $42^2-2=1\,762$ et le reste de 1 762 par 127 est 111 comme celui de 1 416 317 954 par 127 (et comme celui de $4\,487^2-2=20\,133\,167$ par 127 d'ailleurs). Enfin, on calcule $111^2-2=12\,319$ et le reste de 12 319 par 127 est 0 comme celui de

```
Lprem=[3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,\
       103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199]
r=1 #rang d'un nombre de Mersenne premier
for p in Lprem :
    s=4    # terme s(1)
    for k in range(3,p+1) :
        s=(s**2-2)%(2**p-1)
    if s==0 :
        r+=1
        print("oui, M"+str(k)+" = "+str(2**k-1)+" est un nombre premier. C'est le Mersenne no"+str(r))
    else :
        print("non, M"+str(k)+" = "+str(2**k-1)+" n'est pas un nombre premier. Le résidu est "+str(s))
```

```
oui, M3 = 7 est un nombre premier. C'est le Mersenne no2
oui, M5 = 31 est un nombre premier. C'est le Mersenne no3
oui, M7 = 127 est un nombre premier. C'est le Mersenne no4
non, M11 = 2047 n'est pas un nombre premier. Le résidu est 1736
oui, M13 = 8191 est un nombre premier. C'est le Mersenne no5
oui, M17 = 131071 est un nombre premier. C'est le Mersenne no6
oui, M19 = 524287 est un nombre premier. C'est le Mersenne no7
non, M23 = 8388607 n'est pas un nombre premier. Le résidu est 6107895
non, M29 = 536870911 n'est pas un nombre premier. Le résidu est 458738443
oui, M31 = 2147483647 est un nombre premier. C'est le Mersenne no8
non, M37 = 137438953471 n'est pas un nombre premier. Le résidu est 117093979072
non, M41 = 2199023255551 n'est pas un nombre premier. Le résidu est 856605019673
non, M43 = 8796093022207 n'est pas un nombre premier. Le résidu est 5774401272921
non, M47 = 140737488355327 n'est pas un nombre premier. Le résidu est 96699253829728
non, M53 = 9007199254740991 n'est pas un nombre premier. Le résidu est 5810550306096509
non, M59 = 576460752303423487 n'est pas un nombre premier. Le résidu est 450529175803834166
oui, M61 = 2305843009213693951 est un nombre premier. C'est le Mersenne no9
non, M67 = 147573952589676412927 n'est pas un nombre premier. Le résidu est 44350645312365507266
non, M71 = 2361183241434822606847 n'est pas un nombre premier. Le résidu est 271761692158955752596
non, M73 = 9444732965739290427391 n'est pas un nombre premier. Le résidu est 2941647823169311845731
non, M79 = 604462909807314587353087 n'est pas un nombre premier. Le résidu est 149246123283525944719226
non, M83 = 9671406556917033397649407 n'est pas un nombre premier. Le résidu est 74263932232114893531232218
oui, M89 = 618970019642690137449562111 est un nombre premier. C'est le Mersenne no10
non, M97 = 158456325028528675187087900671 n'est pas un nombre premier. Le résidu est 138799132171283648987055810555
non, M101 = 25353012004564588029934064107 51 n'est pas un nombre premier. Le résidu est 24574576398683058552749163448886
non, M103 = 10141204801825835211973625643007 n'est pas un nombre premier. Le résidu est 447327545995254516118850996511 8
oui, M107 = 162259276829213363391578010288127 est un nombre premier. C'est le Mersenne no11
non, M109 = 649037107316853453566312041152511 n'est pas un nombre premier. Le résidu est 80310482899578688674364643057506
non, M113 = 10384593717069655257060992658440191 n'est pas un nombre premier. Le résidu est 660079124374067013275891922799333 7
oui, M127 = 170141183460469231731687303715884105727 est un nombre premier. C'est le Mersenne no12
```

2 005 956 546 822 746 114 par 127. Le fait de pouvoir travailler avec les résidus de la suite modulo M_n plutôt qu'avec la suite directement évite les nombres plus grands que M_n. Pourquoi (194−127)²−2 et (194)²−2 ont-ils le même reste dans la division par 127 ? Si on note *R* ce reste, on a $\frac{194^2}{127}=Q+\frac{R}{127}$ (Q est le quotient entier de cette division) et

$$\frac{(194-127)^2}{127}=\frac{194^2-2\times127\times194+127^2}{127}=\frac{194^2}{127}-2\times194+127=Q+\frac{R}{127}-2\times194+127=Q'+\frac{R}{127}$$

(Q' est le nouveau quotient entier de cette division). Donc $\frac{194^2}{127}$ et $\frac{(194-127)^2}{127}$ ont le même reste dans la division par 127. Retrancher 2 à ces deux nombres fera diminuer d'autant le reste commun modulo 127, donc ces deux nombres ont le même reste, et la propriété se propage aux autres termes de la suite.

Cette fois, pour tester les nombres de Mersenne jusqu'à M_{127}, cela ne prend qu'une fraction de seconde. Cela ne pose plus de problèmes d'aller plus loin dans la recherche. Pour éviter d'écrire une liste démesurée de nombres premiers (la liste *Lprem*), on supprime le contrôle de la boucle par cette liste et on le remplace par une boucle sur les nombres impairs à partir de 3.

Voici maintenant notre ultime proposition. On y a supprimé les affichages des nombres de Mersenne qui ne sont pas premiers. L'objectif est de lister les nombres de Mersenne premiers jusqu'à *k*=1 000. Nous en avons trouvé 14 en 2 secondes, le dernier étant M_{607} qui comporte 183 chiffres.

```
r=1 #rang d'un nombre de Mersenne premier
p=3
while p < 1000 :
    s=4    # terme s(1)
    for k in range(3,p+1) :
        s=(s**2-2)%(2**p-1)
    if s==0 :
        r+=1
        print("M"+str(p)+" est un nombre premier. Il a "+str(len(str(2**p-1)))+" chiffres. C'est le Mersenne no"+str(r))
    p+=2
```

```
M3 est un nombre premier. Il a 1 chiffres. C'est le Mersenne no2
M5 est un nombre premier. Il a 2 chiffres. C'est le Mersenne no3
M7 est un nombre premier. Il a 3 chiffres. C'est le Mersenne no4
M13 est un nombre premier. Il a 4 chiffres. C'est le Mersenne no5
M17 est un nombre premier. Il a 6 chiffres. C'est le Mersenne no6
M19 est un nombre premier. Il a 6 chiffres. C'est le Mersenne no7
M31 est un nombre premier. Il a 10 chiffres. C'est le Mersenne no8
M61 est un nombre premier. Il a 19 chiffres. C'est le Mersenne no9
M89 est un nombre premier. Il a 27 chiffres. C'est le Mersenne no10
M107 est un nombre premier. Il a 33 chiffres. C'est le Mersenne no11
M127 est un nombre premier. Il a 39 chiffres. C'est le Mersenne no12
M521 est un nombre premier. Il a 157 chiffres. C'est le Mersenne no13
M607 est un nombre premier. Il a 183 chiffres. C'est le Mersenne no14
```

Comme nous pouvons facilement aller plus loin, nous essayons d'aller jusqu'à 10 000. Cela prend tout de même du temps, de plus en plus, au fur et à mesure de l'augmentation de la taille des nombres. Nous voyons progressivement la liste s'allonger : M_{1279}, M_{2203}, M_{2281}, M_{3217}, M_{4253}, M_{4423}. Le dernier nombre de Mersenne trouvé, le nombre M20, comporte 1332 chiffres ! Notre programme peine à en trouver davantage, nous en resterons donc là.

```
M1279 est un nombre premier. Il a 386 chiffres. C'est le Mersenne no15
M2203 est un nombre premier. Il a 664 chiffres. C'est le Mersenne no16
M2281 est un nombre premier. Il a 687 chiffres. C'est le Mersenne no17
M3217 est un nombre premier. Il a 969 chiffres. C'est le Mersenne no18
M4253 est un nombre premier. Il a 1281 chiffres. C'est le Mersenne no19
M4423 est un nombre premier. Il a 1332 chiffres. C'est le Mersenne no20
```

Partie n°2 : Liste de nombres premiers

a) Écrire un programme qui détermine la liste des nombres premiers inférieurs ou égaux à un entier *n* donné. On pourra afficher proprement cette liste (tableau) et donner des informations sur chaque nombre premier (son rang, sa valeur, son résidu modulo 4, modulo 6 ou modulo 10, etc.) ainsi qu'un récapitulatif statistique (effectif total des nombres premiers, pourcentages par résidu dans les différents modulo, etc.). Pour ce faire, on peut partir de rien, et utiliser la méthode du crible d'Ératosthène (un algorithme qui part de la liste des nombres entiers compris entre 2 et un entier *n* donné, et qui raye successivement tous les multiples du premier nombre non rayé, puis du 2ème nombre non rayé, etc.).

La méthode du crible d'Ératosthène est relativement simple à mettre en œuvre : on parcourt la liste des nombres entiers inférieurs ou égaux à *n* autant de fois qu'il est possible en « rayant » les multiples du premier nombre non rayé, puis du 2d, du 3ème, etc. Le verbe « rayer » dénote l'usage traditionnel qui est manuscrit, mais en informatique on procèdera au « rayage » en affectant 0 à la place du nombre, ou en supprimant purement et simplement les nombres. On arrive ainsi à la fin de la liste initiale après l'avoir parcourue autant de fois qu'il y a de nombres premiers inférieurs ou égaux à *n*. Il ne reste plus qu'à procéder à l'affichage qui comporte quelques petits traitements sans difficulté. Si on ne voit pas trop l'intérêt de ces statistiques, disons qu'elles nous obligeront seulement à organiser notre affichage. Dans un premier temps, on peut se contenter de n'afficher que la liste des nombres entiers comme Python sait le faire : par exemple, si on entre *n*=12, il nous affiche [2, 3, 5, 7, 11].

Une remarque est nécessaire, pour éviter de procéder à l'expurgation de valeurs inexistantes : le plus grand des nombres premiers qui peut avoir des multiples à expurger de la liste est inférieur ou égal à $\sqrt{n}$. Nous avions déjà fait bénéficier de cette remarque l'algorithme du 2a. Un nombre premier supérieur à $\sqrt{n}$, notons-le *P*, aura déjà vu ses premiers multiples expurgés (2*P* étant un multiple de 2 aura été expurgé dès le départ, idem pour 3*P*, 5*P*, etc.). Le premier multiples non expurgé de *P* est donc P^2 qui est forcément plus grand que *n*. Ceci évite donc de boucler dès que l'on doit s'interroger sur les multiples d'un nombre premier $p > \sqrt{n}$.

```
n=int(input("Saisissez un nombre : "))
Liste_premiers=list(range(2,n+1))
k=2
nRacine=n**0.5
while k<nRacine :
    Liste_premiers=[p for p in Liste_premiers if p<=k or p%k!=0]
    k=Liste_premiers[Liste_premiers.index(k)+1] # nouveau nombre premier
print("plus grand premier="+str(Liste_premiers[-1])+" nombre de premiers="+\
      str(len(Liste_premiers))+" liste premiers: ",Liste_premiers)
```

```
Saisissez un nombre : 12
plus grand premier=11 nombre de premiers=5 liste premiers:  [2, 3, 5, 7, 11]

Saisissez un nombre : 100
plus grand premier=97 nombre de premiers=25 liste premiers:  [2, 3, 5, 7, 11, 13,
17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Une fois que cette partie du programme est au point, il reste à réaliser l'affichage soigné et les statistiques. Nous avons le choix entre une sortie « console » (dans le *shell* de Python), sur un panneau graphique (utilisant le module *tkinter* de Python) ou sur un fichier externe (utilisant le module *os* de Python). Pour ce travail qui peut conduire à une liste assez longue, contenant des informations que l'on aimerait peut-être conserver pour une utilisation ultérieure, la dernière solution me semble la mieux adaptée. Concentrons nos efforts sur une écriture qui soit propre (des colonnes bien individualisées) afin d'être bien lisible comme si c'était un tableau. On pourrait peut-être s'arranger pour enregistrer un fichier qui respecte la syntaxe d'un tableur afin de profiter des fonctions d'édition du tableur, mais cela sera dans un 2ème temps.

Nous voulons écrire une ligne par nombre premier en calibrant les largeurs des colonnes sur le plus grand nombre à écrire (la fin de la liste). Quelque chose comme ce qui suit conviendrait :

Rang	Valeur	R mod4	R mod6	R mod10
1	2	2	2	2
2	3	3	3	3
3	5	1	5	5
4	7	3	1	7
5	11	3	5	1

statistiques, total :5
résidus mod4 :0(0%),1(20%),2(20%),3(60%)
résidus mod6 :0(0%),1(20%),2(20%),3(20%),4(0%),5(40%)
résidus mod10:0(0%),1(20%),2(20%),3(20%),4(0%),5(20%),6(0%),7(20%),8(0%),9(0%)

Voilà notre objectif, il n'y a plus qu'à réaliser le programme qui réalise cela.

```
from os import getcwd,chdir
chdir(getcwd())
fichier=open('nombresPremiers.txt','w')
n=int(input("Saisissez un nombre : "))
Liste_premiers=list(range(2,n+1))
k=2
nRacine=n**0.5
while k<nRacine :
    Liste_premiers=[p for p in Liste_premiers if p<=k or p%k!=0]
    k=Liste_premiers[Liste_premiers.index(k)+1]#nouveau nombre premier

while len(s)<len(str(Liste_premiers[-1])) :  # entête pour les valeurs
        s+=' '
fichier.write('Rang \t'+s+'\t mod4 \t mod6 \t mod10'+'\n')
```

```
for p in Liste_premiers :
    residu4[p%4]+=1                          #incrémentations des résidus
    residu6[p%6]+=1
    residu10[p%10]+=1
    nombrePremierS=str(p )                # 1er champ affiché : le rang
    while len(nombrePremierS)<len(str(Liste_premiers[-1])) :
        nombrePremierS+=' '          # 2ème champ : la valeur (formatée)
    fichier.write(str(k)+'\t'+nombrePremierS+\
'\t'+str(p%4)+'\t'+str(p%6)+'\t'+str(p%10)+'\n')
    k+=1                                 #ensuite les 3 résidus (ci-dessus)
n=len(Liste_premiers)
fichier.write('nombre de premiers : '+str(n)+'\n')# affichage du total
fichier.write('résidus :\t 0 \t 1\t 2\t 3\t 4\t 5\t 6\t 7\t 8\t 9\n')
s='mod04 : \t '                                  # entête pour les résidus
for i in range(4) :
    s+=str(residu4[i]*100.0/n)+'% \t'
fichier.write(s+'\n')                # les 4 résidus possibles modulo 4
s='mod06 : \t '
for i in range(6) :
    s+=str(residu6[i]*100.0/n)+'% \t'
fichier.write(s+'\n')                # les 6 résidus possibles modulo 6
s='mod10 : \t '
for i in range(10) :
    s+=str(residu10[i]*100.0/n)+'% \t'
fichier.write(s+'\n')             # les 10 résidus possibles modulo 10
fichier.close()
print('fichier écrit : nombresPremiers.txt')
```

fichier nombresPremiers.txt

```
Rang    Valeur   mod4    mod6    mod10
1       2        2       2       2
2       3        3       3       3
3       5        1       5       5
4       7        3       1       7
5       11       3       5       1
nombre de premiers : 5
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    20.0%   20.0%   60.0%
mod06 :          0.0%    20.0%   20.0%   20.0%   0.0%    40.0%
mod10 :          0.0%    20.0%   20.0%   20.0%   0.0%    20.0%   0.0%    20.0%   0.0%    0.0%
```

Il reste à tester ce programme de génération de la liste des nombres premiers avec un nombre *n* supérieur à 12. L'affichage doit pouvoir s'adapter à la largeur des colonnes qui va augmenter (surtout celle des valeurs). Au passage, on pourra retirer une statistique plus précise sur les résidus. Pour *n*=1000, la statistique est plus précise, certes, mais trop : les valeurs des pourcentages n'ayant pas été arrondis, ils sont affichés avec une précision ridicule qui met la pagaille dans nos colonnes.

```
....    ....
168     997     1       1       7
nombre de premiers : 168
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    47.619047619047619%     0.5952380952380952%     51.785714285714285%
mod06 :          0.0%    47.619047619047619%     0.5952380952380952%     0.5952380952380952%     0.0%    51.190476190476190%
mod10 :          0.0%    23.809523809523810%     0.5952380952380952%     25.0%   0.0%    0.5952380952380952%     0.0%    27.380952380952380%
```

Une fois ce petit détail corrigé, on peut obtenir un tableau bien ordonné et augmenter encore *n*. Voilà ce que cela donne pour *n*=1 000 000. Notre correction pour l'affichage a été d'ajouter [:5] après la chaîne *str*(residu*x*[*i*]*100.0/*n*) dans les lignes où l'on affecte les pourcentages de chaque résidu. Cette fonctionnalité de l'objet chaîne utilise la conversion implicite de la chaîne en liste, et opère une extraction de l'indice 0 (inclus) à l'indice 5 (exclu). De cette façon, on réalise une approximation par défaut (une troncature) du pourcentage. La 1ère de ces trois lignes devient : s+=*str*(residu4[*i*]*100.0/*n*)[:5]+'% \t'
Bien d'autres techniques peuvent être employées pour réaliser ce formatage des nombres flottants, comme aussi pour l'ajustement des colonnes à tailles variables, et plus généralement, comme toutes les options de ce programme.

```
Rang    Valeur   mod4    mod6    mod10          pour n=1 000 000
1       2        2       2       2
2       3        3       3       3
3       5        1       5       5       <=== affichage des 78498 nombres premiers
....    ....
78498   999983   3       5       3              affichage des statistiques
                                                        vvv
nombre de premiers : 78498
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    49.90%  0.001%  50.09%
mod06 :          0.0%    49.97%  0.001%  0.001%  0.0%    50.02%
mod10 :          0.0%    24.99%  0.001%  25.05%  0.0%    0.001%  0.0%    24.99%  0.0%    24.95%
```

Une petite remarque : le pourcentage du résidu 2 pour chacun des modulos est un nombre infime car, en fait, il n'y a que pour le nombre premier 2 que le résidu est égal à 2. Pour tous les autres nombres premiers, comme ces autres nombres sont tous impairs, le résidu ne peut jamais être égal à 2. On constate ainsi que, exception fait du résidu de 2, les résidus modulo 4, 6 et 10 (des nombres pairs) ne peuvent jamais être égaux à 0, 2, 4, 6 ou 8. De la même façon, les pourcentages infimes correspondants à 3 *mod* 6 ou 5 *mod* 10, sont imputables aux seules nombres premiers 3 et 5. Si l'on demande au programme de faire ces statistiques sans tenir compte des 3 premiers nombres premiers, les pourcentages infimes seraient réduits à 0%, très exactement (voir ci-dessous).

```
fichier.write('Rang \t'+s+'\t mod4 \t mod6 \t mod10'+'\n')
Liste_premiers=Liste_premiers[3:]        # pour supprimer les 2, 3 et 5
for p in Liste_premiers :
```

ajout de cette instruction

```
Rang    Valeur   mod4    mod6    mod10
1       7        3       1       7
2       11       3       5       1
3       13       1       1       3
  ....    ....
```

lorsqu'on enlève les nombres premiers 2, 3 et 5

```
78495   999983  3       5       3
nombre de premiers : 78495
résidus :         0       1       2       3       4       5       6       7       8       9
mod04 :           0.0%    49.90%  0.0%    50.09%
mod06 :           0.0%    49.97%  0.0%    0.0%    0.0%    50.02%
mod10 :           0.0%    24.99%  0.0%    25.05%  0.0%    0.0%    0.0%    24.99%  0.0%    24.96%
```

On peut conjecturer, au vu de ces résultats que :

- la moitié des nombres premiers s'écrit $4k+1$ et l'autre moitié s'écrit $4k+3$ (exception : 2),
- la moitié des nombres premiers s'écrit $6k+1$ et l'autre moitié s'écrit $6k+5$ (exceptions : 2 et 3),
- un quart des nombres premiers s'écrit $10k+1$, les trois autres quarts s'écrivant $10k+3$, $10k+7$, et $10k+9$ (exceptions : 2 et 5).

Les petites fluctuations autour de ces valeurs équitables sont dues vraisemblablement au fait qu'on a pris qu'un (petit) échantillon de nombres premiers : qu'est-ce que 78 498 par rapport à l'infini ?

Quel est l'intérêt de connaître les résidus modulo 4, 6 ou 10 ?

Les nombres premiers de la forme $4k+1$ (la moitié des nombres premiers) sont dits « de Pythagore » car ce sont les seuls nombres premiers que l'on peut écrire comme la somme de deux carrés, et la décomposition est unique. Par exemple, on a $5=4\times1+1=1^2+2^2$, $13=4\times3+1=2^2+3^2$, $17=4\times4+1=1^2+4^2$, $29=4\times7+1=2^2+5^2$. Le nombre 2 a une place à part car il s'écrit comme une somme de carrés ($2=1^2+1^2$) sans avoir un résidu égal à 1 modulo 4. Les autres nombres premiers n'ont pas cette faculté de s'écrire comme une somme de deux carrés : $3=4\times0+3$, $7=4\times1+3$ et $11=4\times2+3$ en sont les trois premiers exemples. Pour les nombres composés, il faut que les facteurs premiers de la forme $4k+3$ soient tous à des exposants pairs (car $(4k+3)^2=16k^2+24k+9=4(4k^2+6k+2)+1=4k'+1$) pour que le nombre puisse se décomposer en somme de deux carrés. Pour plus de précision sur ce sujet, voir le théorème des deux carrés.

Le fait que tous les nombres premiers (sauf 2 et 3) soient voisins d'un multiple de 6 permet de les mémoriser facilement : les voisins de 6 sont 5 et 7, ceux de 12 sont 11 et 13, ceux de 18 sont 17 et 19, ensuite on doit éliminer quelques multiples de 5 (ceux qui se terminent par 5 et qui sont voisins d'un multiple de 6, à commencer par 25, puis 35, 55, 65, 85, 95, etc.) et puis les multiples de 7 voisins d'un multiple de 6 (cela commence par 49, puis 77 et 91, etc.). Si on veut retrouver tous les nombres premiers jusqu'à 100, il n'y a rien à enlever de plus aux voisins des multiples de 6 (le prochain voisin à éloigner est le carré de 11 : 121). Cette technique s'appelle la « barre des 6 » : en vert, les nombres premiers (en vert clair ceux qui ne sont pas voisin d'un multiple de 6) voisins des multiples de 6 (en bleu) et en rouge (respectivement orange et rose) les multiples de 5 (respect. De 7 et de 11) qu'il faut enlever de cette liste.

2	5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113	119
	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
3	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109	115	121

Le fait que les nombres premiers se terminent par 1, 3, 7 ou 9 est une évidence car (sauf 2) aucun n'est pair (donc aucun ne se termine par 0, 2, 4, 6 ou 8) et (sauf 5) aucun n'est un multiple de 5 (donc aucun ne se termine par 0 ou 5). Certaines propriétés concernent plus particulièrement certains nombres, par exemple les nombres de la forme $N=4^n+n^2$. On montre facilement que si n s'écrit $10k+1$ ou $10k+9$ alors N est divisible par 5 ; de même on montre facilement que si n est pair alors N aussi ; on en déduit que pour que N soit premier, il faut que n s'écrive $10k+3$ ou $10k+7$.

D'une façon générale, on peut s'interroger sur la fréquence des résidus dans tous les modulos, ne serait-ce que par curiosité (une saine habitude en mathématiques). Avec quelques copiés/collés notre programme précédent répond à cette demande reformulée. Voici donc ces fréquences pour les modulos 2 à 10. Nous avons pris n=1 000 000 comme précédemment et nous avons enlevé le 7 de la liste des nombres premiers pour éviter de trouver une fréquence infime du 0 modulo 7.

Rang	Valeur	mod2	mod3	mod4	mod5	mod6	mod7	mod8	mod9	mod10
1	11	1	2	3	1	5	4	3	2	1
2	13	1	1	1	3	1	6	5	4	3
3	17	1	2	1	2	5	3	1	8	7
4	19	1	1	3	4	1	5	3	1	9
5	23	1	2	3	3	5	2	7	5	3
....	....	lorsqu'on enlève les nombres 2, 3, 5 et 7								
78494	999983	1	2	3	3	5	5	7	2	3

nombre de premiers : 78494

résidus :	0	1	2	3	4	5	6	7	8	9
mod02 :	0.0%	100.0%								
mod03 :	0.0%	49.97%	50.02%							
mod04 :	0.0%	49.90%	0.0%	50.09%						
mod05 :	0.0%	24.99%	24.99%	25.05%	24.96%					
mod06 :	0.0%	49.97%	0.0%	0.0%	0.0%	50.02%				
mod07 :	0.0%	16.64%	16.64%	16.69%	16.64%	16.69%	16.67%			
mod08 :	0.0%	24.90%	0.0%	25.03%	0.0%	24.99%	0.0%	25.05%		
mod09 :	0.0%	16.64%	16.68%	0.0%	16.65%	16.64%	0.0%	16.68%	16.68%	
mod10 :	0.0%	24.99%	0.0%	25.05%	0.0%	0.0%	0.0%	24.99%	0.0%	24.96%

On peut être intrigué par certains de ces résultats, comme par exemple l'absence de nombres premiers qui ont des résidus égaux à 3 ou 6 modulo 9. Ceci traduit pourtant strictement le fait qu'aucun nombre premier (à part 3) n'est un multiple de 3. Si on calcule la somme des chiffres d'un nombre premier, pour faire une preuve par 9 par exemple, on ne trouvera jamais 0, 3 et 6.

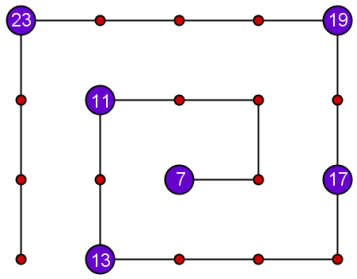
b) On peut aussi partir d'une première liste des premiers nombres premiers fournie en argument, et déterminer ceux qui manquent par une adaptation de la méthode précédente.
Ce dispositif peut s'avérer utile dans certaines situations où on ne sait pas à l'avance combien de nombres premiers seront utiles. Bien sûr, on pourrait se contenter d'utiliser la méthode précédente qui recalcule toute la liste des nombres entiers, mais il est plus judicieux, si l'on veut gagner en efficacité, d'étendre une liste existante lorsque le besoin s'en fait sentir.
Nous allons partir du premier nombre impair supérieur au dernier nombre premier de la liste (*Lprem*[-1]), baptisé *first*, et construire la liste *L* des nombres impairs inférieurs ou égaux à *n*. Avec une liste initiale *Lprem*=[2, 3, 5, 7, 11, 13, 17, 19, 23], la liste *L* est donc égale à [25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, etc.]. Dans cette liste, nous allons affecter 0 aux multiples des nombres premiers de la liste *Lprem* initiale, en commençant par le premier qui n'a pas déjà été supprimé. Pour cela, posons nous la question : quel est le rang de 7×7 dans *L* ? C'est 12 d'après notre liste L, on obtient ce nombre en effectuant (7×7−25)//2=(49−25)//2=24//2=12. Nous pouvons mettre 0 à la place de 49, et puis aussi à tous les multiples de 7 supérieurs à 7×7 qui sont dans *L*. Ce n'est pas la peine de s'occuper des multiples de 7 qui sont avant 7×7 car ils auront été effacés par la procédure appliquée aux nombres premiers inférieurs à 7. On ne s'occupe pas des nombres pairs car il n'y en a pas dans *L*. On commence par les multiples de 3 : le rang de 3×3 dans *L* est (3×3−25)//2=(9−25)//2=−16//2=−8. C'est normal de trouver un nombre négatif, car *L* commençant à 25 ne contient pas 9. Ce nombre se trouve virtuellement 8 rangs avant 25. Le prochain multiple de 3 n'étant pas pair est 9+6=15 qui se trouve au rang −8+3=−5 (on ajoute 3 car 6=3×2, et les nombres dans *L* vont de 2 en 2). Ensuite, il y a 15+6=21 qui se trouve au rang −5+3=−2 (toujours pas dans *L*), et puis 21+6=27 qui se trouve au rang −2+3=1. On trouve effectivement 27 au rang 1 dans L, et on peut l'écraser avec 0 car il n'est pas premier. On continue à écraser les multiples de 3 en ajoutant 3 à l'indice du précédent multiple, et ainsi tous les multiples de 3 sont marqués. On fait de même avec les multiples de 5 : le premier a chercher est 5×5 qui est au rang (5×5−25)//2=0 ; on le remplace par 0 ainsi que les multiples de 5 suivants que l'on trouve en ajoutant 5 à l'indice dans *L* (on va ainsi de 10 en 10, évitant les nombres pairs qui n'y figurent pas). Tout cela est plus long à expliquer qu'à faire, mais on n'en a pas encore fini avec les nombres composés de *L*.
Supposons que la liste *L* était [25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, … , 999]. Ce que nous venons de faire à modifié cette liste en [0, 0, 29, 31, 0, 0, 37, 0, 41, 43, 0, 47, 0, 0, … , 0]. Ne voyons-nous que des nombres premiers ici ? Non, bien sûr, il y a tous les nombres composés des nombres premiers supérieurs à ceux de la liste initiale, à commencer par 29×29=841, 29×31=899, 31×31=961 et c'est tout si la liste *L* s'arrêtait à 999. Ces nouveaux nombres composés n'ont pas été retirés par notre précédente boucle. Au lieu d'une boucle *for*, on utilise une boucle while ici, en limitant la recherche aux nombres composés construits avec les nouveaux nombres premiers inférieurs ou égaux à la racine carrée du nombre *n* (on a déjà expliqué pourquoi).
Après ce dernier passage dans la liste *L*, il ne reste plus que des zéros et des nombres premiers. Il suffit alors de recopier la liste sans les zéros, à la suite de la liste *Lprem* initiale.

```
from math import sqrt
def Lpremier(nb):#étend la liste des nombres premiers jusqu'à nb,
    global Lprem
    if nb%2==0 : nb-=1
    if Lprem[-1]+1>=nb : return 0
    first=Lprem[-1]+2
    L=[first]
    for i in range(1,(nb-first)//2+1):
        L.append(first+2*i)
    for prem in Lprem:
        if prem>2 :
            j=(prem*prem-first)//2
            while j<0 : j+=prem
            while j<len(L) :
                L[j]=0
                j+=prem
    nRacine=sqrt(nb)
    i=0
    nouveau=first
    while nouveau<=nRacine :
        if L[i]!=0 :
            j=(nouveau*nouveau-first)//2
        while j<len(L) :
            L[j]=0
            j+=nouveau
        i+=1
        nouveau+=2
    Lprem_suite=[p for p in L if p!=0]
    Lprem+=Lprem_suite
    return len(Lprem_suite)
```

```
Lprem=[2,3,5,7,11,13,17,19,23]
n=int(input("Jusqu'à quel nombre voulez-vous étendre la liste : "))
ajout=Lpremier(n)
print("Plus grand premier trouvé : {}, nombre total de premiers : {}, \
nombre de premiers ajoutés : {} ".format(Lprem[-1],len(Lprem),ajout))
print("Liste des nombres premiers inférieurs ou égaux à "+str(n)+" :",Lprem)
```

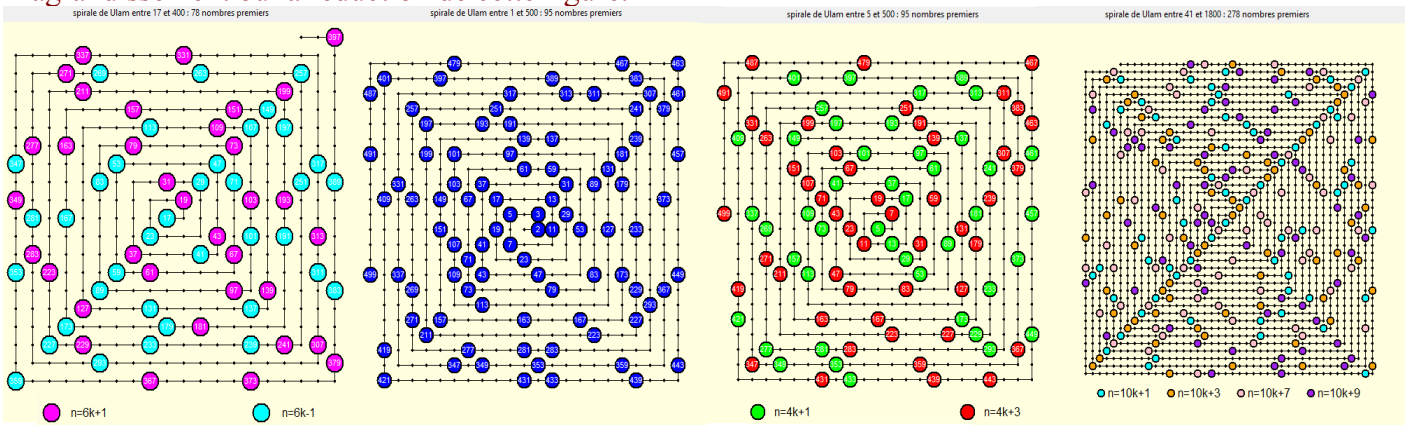
```
Jusqu'à quel nombre voulez-vous étendre la liste : 500
Plus grand premier trouvé : 499, nombre total de premiers : 95, nombre
de premiers ajoutés : 86
Liste des nombres premiers inférieurs ou égaux à 500 : [2, 3, 5, 7, 11
, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157
, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233
, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313
, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401
, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487
, 491, 499]
```

c) Application n°1 : Une représentation graphique amusante et instructive des nombres premiers consiste à enrouler ceux-ci à la manière d'un escargot autour d'un premier nombre (le germe). L'image ci-contre nous donne une idée de ce que l'on veut obtenir (le germe y est égal à 7). Pour bien identifier les nombres premiers des autres, nous allons dessiner un gros point de couleur pour les nombres premiers et un plus petit point d'une couleur différente pour les autres. Pour se donner quelques repères, on peut écrire les valeurs des nombres premiers ou seulement celles des nombres de la forme $10k+1$ (un quart des nombres premiers) par dessus le point correspondant.



Il s'agit donc ici davantage d'un travail graphique. L'utilisation d'un figuré (les gros points sont des disques) et de la couleur suggère d'utiliser une fenêtre *tkinter*. Le traitement mathématique a pour but de déterminer la position des points selon la valeur du nombre, les dimensions de la fenêtre, la valeur du germe. On remarquera que l'escargot qui s'enroule autour du germe a des côtés égaux à 1, 1, 2, 2, 3, 3, 4, 4, etc. Cette remarque est utile pour traduire la position d'un point par rapport au précédent, il suffit d'avoir un compteur i qui s'incrémente de 1 à k, avec un compteur j qui prend les valeurs 0 et 1 alternativement. Lorsque i arrive à k, si j=0 alors i recommence à aller de 1 à k avec j=1 et si j=1 alors k augmente de 1, j=0 et i recommence à aller de 1 à k. Quand k est pair et j=0 on va vers la droite, quand k est pair et j=1 on va vers le haut, quand k est impair et j=0 on va vers la gauche, quand k est impair et j=1 on va vers le bas.

Les dimensions de la fenêtre graphique sont supposées fixes, par contre, on aimerait que le graphique s'affiche en entier pour n compris entre le germe et le nombre maximum qui peut être demandé au début du programme à l'utilisateur (comme le germe d'ailleurs). Il va donc falloir déterminer la dimension c de la maille carrée (en pixel) pour que la longueur du plus grand bord de l'escargot (les bords horizontaux qui mesurent $k\times c$) entre dans la fenêtre. Le rayon des disques aussi doit être dimensionné pour suivre l'agrandissement ou la réduction de cette figure.



On peut ajouter une légende pour rappeler les options de l'affichage. Les résultats sont variables selon le germe choisi, mais en général on peut observer des alignements de nombres premiers : par exemple,

lorsqu'on choisi 17 comme germe (à gauche sur notre illustration), on obtient un alignement sans trou de 16 nombres premiers : 17, 19, 29, 47, 73, 107, 149, 199, 257, et puis aussi 23, 37, 59, 89, 127, 173 et 227. Si on écrit ses nombres dans l'ordre croissant, on obtient 17, 19, 23, 29, 37, 47, 59, 73, 89, 107, 127, 149, 173 199, 227 et 257. Les écarts entre ces nombres sont 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 28 et 30. Croyez-vous que cela soit le fruit du hasard, ou bien pensez-vous qu'une loi arithmétique se cache derrière cet alignement ? Cette disposition en escargot est due à Ulam (d'où le nom spirale de Ulam) qui traça en 1963 la première (à partir du germe 1) et remarqua ces alignements que l'on chercha ensuite à expliquer. L'alignement de la figure de droite contient 40 nombres premiers, à partir de 41, en suivant toujours la même croissance régulière. C'est Euler qui, longtemps avant Ulam, trouva cette suite de nombres premiers, comme images d'entiers consécutifs par le polynôme $P(x)=x^2-x+41$. Si on calcule ces images avec un tableur, voilà ce que l'on trouve. Pour x entier strictement positif, la 1ère valeur pour laquelle $P(x)$ n'est pas premier est 41, car $P(41)=41^2$, et ensuite il y a $P(42)=41\times43$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
41	43	47	53	61	71	83	97	113	131	151	173	197	223	251	281	313	347	383	421	461	503	547	593	641	691	743	797	853	911	971	1033	1097	1163	1231	1301	1373	1447	1523	1601	1681	1763	1847	1933

Nous avons colorié les nombres premiers de la même couleur que sur la spirale : il apparaît clairement une régularité avec aucun nombre premier en $10k$+9, 20% de nombres en $10k$+7 et 40% pour les nombres en $10k$+1 et $10k$+3. De très nombreux mathématiciens se sont penchés sur ces intéressantes dispositions des nombres premiers, mais nous en resterons là.

d) Application n°2 : Lorsqu'on additionne tous les diviseurs stricts (inférieurs au nombre) d'un nombre n, on fabrique un nouveau nombre n' qui peut lui-même suivre le même sort : on additionne tous ses diviseurs pour fabriquer un 3ème nombre n'', etc. Si n=10, les diviseurs stricts de 10 étant 1, 2 et 5, on fabrique le nombre n'=1+2+5=8, puis, comme les diviseurs stricts de 8 sont 1, 2 et 4 on fabrique le nombre n''=1+2+4=7. Comme 7 est premier, on se retrouve au nombre n'''=1 et on s'arrête là car 1 n'a pas de diviseur strict. Il se trouve qu'en essayant avec plusieurs valeurs n de départ, on s'arrête presque toujours sur 1. Prouver cela en écrivant un programme qui affiche cette suite de nombres partant d'un nombre n quelconque. Explorer le comportement des premiers entiers : longueur de la suite et sens de variation. Essayer de repérer les exceptions à la règle énoncée.
Cette situation nécessite de connaître les diviseurs d'un nombre. On a vu comment les obtenir très facilement, sans utiliser les nombres premiers (partie 2a). Mais ici, on va se servir des nombres premiers pour trouver la décomposition en facteurs premiers de n'importe quel nombre n (cette décomposition avait été obtenue dans la partie 2a). On va alors déterminer très facilement la somme des diviseurs stricts, sans même calculer ces diviseurs. Supposons que l'on parte du nombre $1176=2^3\times3^1\times7^2$. Les différents diviseurs de 1176 sont obtenus par combinaison de chacun des facteurs premiers à des exposants variants entre 0 et la multiplicité du facteur considéré. Par exemple $98=2^1\times3^0\times7^2$ est un des diviseurs de 1176, $12=2^2\times3^1\times7^0$ en est un autre. On peut trouver la somme des diviseurs de 1176 en écrivant le produit $(2^0+2^1+2^2+2^3)\times(3^0+3^1)\times(7^0+7^1+7^2)$ qui contient, dans chaque facteur, toutes les possibilités de contribution d'un des facteurs premiers de la décomposition (on fait varier l'exposant de 0 à la multiplicité du facteur considéré). On peut s'en tenir à cette formule qui est simple à programmer, ou bien utiliser un des résultats sur les suites géométriques qui permet de simplifier la somme $2^0+2^1+2^2+2^3=1+2+4+8=15$ en $\frac{2^4-1}{2-1}=\frac{16-1}{1}=15$, ou bien, plus généralement $a^0+a^1+a^2+...+a^n=\frac{a^{n+1}-1}{a-1}$. Quoi qu'il en soit, la recherche de la décomposition en facteurs premiers du nombre n suivie par le calcul de la somme des diviseurs stricts (on enlève n à la somme obtenue précédemment) va être confiée à la fonction *suivant*() et nous pourrons nous concentrer, dans le programme principal, sur ce que l'on veut faire de ce nombre suivant. À priori, d'après l'énoncé, en recommençant pour différents nombres, on doit tomber sur 1. Les autres comportements de la suite évoqués par le « presque toujours » de l'énoncé, correspondent à des cycles éventuels : on retombe sur un nombre qui a déjà été obtenu. Si on se rappelle de ce qu'est un nombre parfait (voir partie 1c), un nombre égal à la somme de ses diviseurs stricts, dont le plus petit représentant est 6 (6=1+2+3), on sait déjà qu'il va y avoir des cycles de longueur 1 (pour les nombres parfaits) mais on peut aussi penser qu'il peut y en avoir d'une longueur dépassant 1. Un autre comportement éventuel de la suite que l'on peut envisager est « qu'elle ne cesse jamais ». Ce genre de comportement entraînera le programme dans une boucle sans fin si il se rencontre. On peut prévoir une sortie de boucle en cas de dépassement d'un certain seuil.

```
def suivant(nb) :#détermine la somme des
    global Lprem  diviseurs stricts de nb
    decompo=decomposition(nb)
    prod=1
    for facteur in decompo :
        expo,som=0,0
        while expo<=facteur[1] :
            som+=facteur[0]**expo
            expo+=1
        prod*=som
    return prod-nb
```

```
n=int(input("Jusqu'à quel nombre voulez-vous aller : "))
for i in range(2,n+1) :
    suite=[i]
    successeur=suivant(i)
    seuil=1000000
    while successeur<seuil :
        if successeur in suite or successeur==1: break
        suite.append(successeur)
        successeur=suivant(successeur)
    suite.append(successeur)
    if suite[-1]==1 :
        print(suite)
    elif successeur>=seuil :
        print("suite indéterminée de longueur {}.".format(len(suite)),suite)
    else :
        rang=0
        while suite[rang]!=suite[-1] :
            rang+=1
        print("cycle de longueur {} : ".format(len(suite)-rang-1),suite)
```

```
Jusqu'à quel nombre voulez-vous aller : 100
[2, 1]
[3, 1]
[4, 3, 1]
[5, 1]
cycle de longueur 1 :  [6, 6]
[7, 1]
[8, 7, 1]
[9, 4, 3, 1]
[10, 8, 7, 1]
[11, 1]
[12, 16, 15, 9, 4, 3, 1]
[13, 1]
[14, 10, 8, 7, 1]
[15, 9, 4, 3, 1]
[16, 15, 9, 4, 3, 1]
[17, 1]
[18, 21, 11, 1]
[19, 1]
[20, 22, 14, 10, 8, 7, 1]
[21, 11, 1]
[22, 14, 10, 8, 7, 1]
[23, 1]
[24, 36, 55, 17, 1]
cycle de longueur 1 :  [25, 6, 6]
[26, 16, 15, 9, 4, 3, 1]
[27, 13, 1]
cycle de longueur 1 :  [28, 28]
[29, 1]
[30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
[31, 1]
[32, 31, 1]
[33, 15, 9, 4, 3, 1]
[34, 20, 22, 14, 10, 8, 7, 1]
[35, 13, 1]
[36, 55, 17, 1]
[37, 1]
[38, 22, 14, 10, 8, 7, 1]
[39, 17, 1]
[40, 50, 43, 1]
[41, 1]
[42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
[43, 1]
[44, 40, 50, 43, 1]
[45, 33, 15, 9, 4, 3, 1]
[46, 26, 16, 15, 9, 4, 3, 1]
[47, 1]
[48, 76, 64, 63, 41, 1]
[49, 8, 7, 1]
[50, 43, 1]
[51, 21, 11, 1]
[52, 46, 26, 16, 15, 9, 4, 3, 1]
[53, 1]
[54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
```

Tel que c'est écrit, nous pouvons explorer le comportement des nombres entiers en commençant à 1 jusqu'à *n*. On peut éviter de toujours commencer à 1 en demandant d'entrer l'entier par lequel on veut commencer. Mais déjà on s'aperçoit que certaines suites identifiées sont complétées par de nouveaux nombres : la suite de 10 est [10, 8, 7, 1] mais on la retrouve avec la suite de 14 qui est [14, 10, 8, 7, 1]. ensuite on trouve la suite de 20 qui prolonge encore cette suite [20, 22, 14, 10, 8, 7, 1], le prédécesseur de 20 est 34, etc. Il y a d'autres façon de parvenir à 1 que de passer par 7 : on peut y parvenir en passant par 3 (la suite [30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1] est la plus longue pour $n<100$ mais il y a aussi la suite [46, 26, 16, 15, 9, 4, 3, 1] qui, à partir de 15 fusionne), par 11 (la suite [18, 21, 11, 1]), par 13 (la suite [69, 27, 13, 1]), par 17 (la suite [36, 55, 17, 1]), etc. Il semblerait qu'on retrouve tous les nombres premiers sauf 2 et 5. Un prédécesseur de 2 devrait avoir pour diviseurs 1 et 1 ce qui n'est pas possible et un prédécesseur de 5 devrait avoir pour diviseurs 1 et 4 ce qui n'est pas possible non plus (car alors il devrait y avoir aussi 2) ou 1 2 et 2 ce qui est impossible également. Voilà une nouvelle approche des nombres premiers : ils sont les points d'accès à l'unité pour les nombres entiers qui se trouvent ainsi répartis par classe. La classe de 3 par exemple contient les nombres 4, 9, 15, 16, 26, 30, 33, 42, 45, 46, 54, 66, 78, 90, 144, 259, etc. (nous avons fusionné les deux listes citées précédemment), celle de 7 contient les nombres 8, 10, 14, 20, 22, 34, 62, 118, 148, 152, etc. Une idée de prolongement pour cette exploration serait de remplir une liste des éléments de chacune des classes engendrées par les nombres premiers. Cette idée conduit à la notion de graphe infini des suites aliquotes (voir à ce sujet le site *aliquotes.com* ou l'article de J.-P. Delahaye dans *Pour La Science*, février 2002).

L'existence de cycles de longueur *l* (nombres parfaits pour une *l*=1, nombres amicaux pour la *l*=2 ou sociaux pour une l>2) est une particularité intéressante de ce dispositif des suites « aliquotes » (l'ancien nom pour les diviseurs stricts est « parties aliquotes »). Les nombres parfaits que l'on y découvre sont recherchés depuis l'antiquité : 6, 28, 496, 8128, etc. et liés aux nombres de Mersenne comme on l'a dit dans la partie 2c. Le cycle du nombre parfait 6 est une fin possible pour toute une classe de nombres comme 25, 95, 119, 143, etc. Dans notre idée de prolongement, il faut donner à ces classes de nombres tombant sur un cycle la place qui leur convient. Les premiers cycles de longueurs 2 sont [220, 284], [1184, 1210], [2620, 2924], [5020, 5564], etc. ensuite on trouve des cycles de longueur 4 (on en connait une centaine, comme celui qui commence par 1 264 460) ou 5 (le seul connu commence à 12 496). Le plus long cycle trouvé jusqu'à aujourd'hui est de longueur 28 (il commence à 14 316 et a été trouvé en 1918, sans ordinateur !).

Autre chose qui peut faire l'objet d'un autre prolongement : trouver les nombres qui n'ont pas d'antécédent, les nombres « intouchables ». Nous avons déjà découvert l2 et 5, viennent ensuite 52, 88, 96, 120, etc. Mais le plus grand mystère est à rechercher du côté de ces nombres qui n'en finissent plus, les premiers sont 276, 552, 564, 660, etc. On a recherché leurs suites jusqu'à des nombres de 115 chiffres sans atteindre la fin.

Partie n°3 : Le petit théorème de Fermat

Ce théorème a été souvent utilisé pour montrer qu'un nombre n'est pas premier, voici comment. Ce théorème s'énonce en disant que, si p est un nombre premier et $a \geq 2$ un nombre entier non divisible par p, alors $a^{p-1}-1$ est un multiple de p.

a) Le nombre 7 est premier donc quelque soit l'entier $a \geq 2$ non divisible par 7, on doit avoir a^6-1 divisible par 7. Vérifier cela, en testant la divisibilité de a^6-1 par 7 pour toutes les valeurs inférieures à 14 qui ne soient pas dans la table de 7. Recommencer le même travail pour d'autres nombres premiers inférieurs à 100.

Commençons par 7. Nous allons générer la liste des valeurs de la forme a^6-1 pour tout $a \geq 2$ non divisible par 7. Le programme que nous venons d'écrire pour p=7 reste valable pour p=11 ou pour toute autre valeur de p premier.

```
p=7
L=list(range(2,2*p) )
L0=list(a for a in L if a%p!=0)
print(L0)                    #  liste des entiers a non multiples de p
L1=[a**(p-1)-1 for a in L0]
print(L1)                    #  liste des entiers b =2**(p-1)-1
L2=[b%p for b in L1]
print(L2)                    #  que des restes nuls d'après le théorème
L3=[b//p for b in L1]
print(L3)                    #  les quotients de b par p
print('a \t a**{}-1 \t division par {}'.format(p-1,p))
for rang,a in enumerate(L0) :
    puissance=str(L1[rang])  #  pour un affichage plus propre
    while len(puissance)<10 :# (toutes les colonnes alignées)
        puissance+=' '
    print(a,'\t',puissance,'\t ',str(L3[rang])+'\u00D7'+str(p)+"+0")
```

```
[2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13]
[63, 728, 4095, 15624, 46655, 262143, 531440, 999999, 1771560, 2985983, 4826808
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[9, 104, 585, 2232, 6665, 37449, 75920, 142857, 253080, 426569, 689544]
a        a**6-1          division par 7
2        63               9×7+0
3        728              104×7+0
4        4095             585×7+0
5        15624            2232×7+0
6        46655            6665×7+0
8        262143           37449×7+0
9        531440           75920×7+0
10       999999           142857×7+0
11       1771560          253080×7+0
12       2985983          426569×7+0
13       4826808          689544×7+0
```

Nous donnons ci-dessous les sorties « propres » uniquement pour les premières valeurs de p : 2, 3, 5, 7 et 11. On constate que les valeurs de $b=a^{p-1}-1$ sont très rapidement très grandes, mais néanmoins, la propriété énoncée est toujours vraie (cela ne constitue en aucun cas une démonstration. Ce théorème, énoncé en 1637 par Pierre Fermat, a été véritablement démontré par Leibniz et ensuite Euler, un siècle seulement plus tard).

```
a        a**1-1          division par 2
3        2                1×2+0


a        a**2-1          division par 3
2        3                1×3+0
4        15               5×3+0
5        24               8×3+0


a        a**4-1          division par 5
2        15               3×5+0
3        80               16×5+0
4        255              51×5+0
6        1295             259×5+0
7        2400             480×5+0
8        4095             819×5+0
9        6560             1312×5+0
```

```
a        a**10-1         division par 11
2        1023             93×11+0
3        59048            5368×11+0
4        1048575          95325×11+0
5        9765624          887784×11+0
6        60466175         5496925×11+0
7        282475248        25679568×11+0
8        1073741823       97612893×11+0
9        3486784400       316980400×11+0
10       9999999999       909090909×11+0
12       61917364223      5628851293×11+0
13       137858491848     12532590168×11+0
14       289254654975     26295877725×11+0
15       576650390624     52422762784×11+0
16       1099511627775    99955602525×11+0
17       2015993900448    183272172768×11+0
18       3570467226623    324587929693×11+0
19       6131066257800    557369659800×11+0
20       10239999999999   930909090909×11+0
21       16679880978200   1516352816200×11+0
```

b) Pour prouver qu'un nombre n n'est pas premier, il suffit de montrer que $a^{n-1}-1$ n'est pas divisible par n (c'est la forme contraposée du théorème) pour au moins une valeur de $a \geq 2$ non divisible par n. Par exemple, 91 n'est pas premier car $2^{90}-1=1237940039285380274899124223$ n'est pas divisible par 91 (le reste est 63). Et, en effet, $91=7\times13$, 91 est un nombre composé (on aurait pu le savoir bien plus facilement, sans utiliser ce théorème).

Mettre au point un algorithme qui teste, avec cette méthode la non-primalité d'un entier n. On pourra par exemple tenter de prendre a=2 et montrer que $b=2^{p-1}-1$ n'est pas divisible par n (utiliser la fonction % : reste de la division euclidienne), si ce n'est pas le cas, on tente la division par le nombre premier suivant, etc. Tester, par exemple, la non-primalité de $10^3+1=1001$ puis de $10^7+1=10000001$.

Nous donnons, pour simplifier, une liste des premiers nombres premiers. Nous examinons ensuite la divisibilité de b par p (en fait on teste si le reste de la division de a^{p-1} par p donne 1, ce qui revient au même). Dans le cas d'une divisibilité, le nombre est pseudo-premier en base a (voir plus loin) ; il faut alors continuer l'investigation en prenant le nombre premier suivant. C'est dans le cas contraire, que la contraposée est efficace : on sait tout de suite que le nombre n'est pas premier. Dans ce cas, rien ne sert de

continuer les calculs (qui sont très longs pour des nombres *p* très grands) mais on peut chercher le premier nombre premier qui divise *p* (c'est ce que nous avons fait pour les nombres premiers de notre liste, qui va jusqu'à 101). Pour cela notre liste risque de ne pas être assez longue... Cela pourrait être intéressant de coupler cet algorithme avec celui de la partie 2 qui prolonge la liste des nombres premiers.

```
p=int(input('Entrer un nombre entier impair : '))#tester avec 1001, 10403 puis 10000001
Lprem=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,\
       53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101]#quelques nombres premiers
for a in Lprem :
    if (a**(p-1))%p==1 : #  p est-il premier?
        print('{} est pseudo-premier en base {}'.format(p,a))
    elif p%a==0 :
        print('{} est composé de {} et de {}'.format(p,a,p//a))
        break
    else :
        print('{} est composé, mais pas de {}'.format(p,a))
```

```
Entrer un nombre entier impair : 341
341 est pseudo-premier en base 2
341 est composé, mais pas de 3
341 est composé, mais pas de 5
341 est composé, mais pas de 7
341 est composé de 11 et de 31
```

```
Entrer un nombre entier impair : 1001
1001 est composé, mais pas de 2
1001 est composé, mais pas de 3
1001 est composé, mais pas de 5
1001 est composé de 7 et de 143
```

```
Entrer un nombre entier impair : 10000001
10000001 est composé, mais pas de 2
10000001 est composé, mais pas de 3
10000001 est composé, mais pas de 5
10000001 est composé, mais pas de 7
10000001 est composé de 11 et de 909091
```

Les calculs sont longs pour *p*=10000001 car les nombres calculés sont très grands. Nous n'écrirons pas en entier ici le nombre $2^{1000000}$ (il est constitué de 301 030 chiffres, à raison de 60 lignes par pages et de 100 chiffres par ligne, il faudrait 50 pages pour l'écrire) mais, pour en avoir une idée, donnons juste 2^{1000} qui est calculé pour tester la non-primalité de 1001 (l'algorithme calcule aussi 3^{1000} et 5^{1000}) ; ce nombre ne s'écrit qu'avec 302 chiffres :
107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837881
569585812759467291755314682518714528569231404359845775746985748039345677748242309854210746050
623711418779541821530464749835819412673987675591655439460770629145711964776865421676604298316
52624386837205668069376.

c) Par contre, on ne peut utiliser directement la réciproque qui est fausse généralement : ce n'est pas parce qu'il existe un entier $a \geq 2$ premier avec l'entier impair *n* tel que $a^{n-1}-1$ soit un multiple de *n* (on écrit cela aussi $a^{n-1} \equiv 1 \,(mod\ n)$, que *n* est un nombre premier... De tels nombres impairs *n* sont appelés *nombres pseudo-premiers de base a* (on précise parfois que ce sont des nombres pseudo-premiers de Fermat car il existe d'autres sortes de nombres pseudo-premiers). Si *a* est égal à 2, et si *a* est quelconque (pas nécessairement premier avec *p*) de tels nombres pseudo-premiers sont appelés *nombres de Poulet*. Si la propriété est vérifiée pour tout entier *a* compris entre 2 et *n*, le nombre est appelé *nombre de Carmichael*.
Retrouver, à l'aide d'un algorithme, les dix premiers nombres de Poulet (la liste commence par 341, 561, 645, 1105 et 1387) et les dix premiers nombres de Carmichael (la liste commence par 561, 1105 et 1729).
Quels sont les premiers nombres pseudo-premiers de base 2, 3, 4, 5, etc. qui soient supérieurs à leur base ? (pour 2, 3 et 4 la réponse est 341, 91 et 15).
Pour les nombres de Poulet, nous avons besoin d'une liste de nombres premiers qui permette d'identifier si un pseudo-premier de base 2 (il passe positivement le test de Fermat pour la base *a*=2, il pourrait donc être premier) est un premier véritable. Nous employons donc la fonction *isPrem*() qui a été définie dans la partie 1. L'algorithme résultant est très simple et ne nécessite pas davantage de commentaires. Les premiers nombres de Poulet sont donc : 341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821, 3277, 4033, 4369, 4371, 4681, 5461, 6601, etc.

```
#recherche des nombres pseudo-premiers de Poulet
Lprem=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,\
       53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101]#quelques nombres premiers
nbrTrouve=0
n_teste=3
isPrem=False
while nbrTrouve<10:
    isPrem=isPremier(n_teste)
    if (2**(n_teste-1))%n_teste==1 and isPrem is False :
        nbrTrouve+=1
        print('{} est le nombre de Poulet n°{}'.format(n_teste,nbrTrouve))
    n_teste+=2
```

```
341 est le nombre de Poulet n°1
561 est le nombre de Poulet n°2
645 est le nombre de Poulet n°3
1105 est le nombre de Poulet n°4
1387 est le nombre de Poulet n°5
1729 est le nombre de Poulet n°6
1905 est le nombre de Poulet n°7
2047 est le nombre de Poulet n°8
2465 est le nombre de Poulet n°9
2701 est le nombre de Poulet n°10
```

On examine, dans cette recherche, tous les nombres impairs à partir de 3 mais il était précisé que la base *a*=2 et le nombre testé ne devait pas nécessairement être premier entre eux. Si l'on devient moins difficile pour les nombres testés et qu'on essaie tous les entiers à partir de 3 (on remplace *n-teste*+=2 par *n-teste*+=1), le résultat est le même : les premiers nombres entiers qui ne sont pas premiers mais qui paraisse

l'être avec la base a=2 selon le test de Fermat, sont les nombres précités.
Pour les nombres de Carmichael, nous procédons presque comme pour les nombres de Poulet. Il y a une boucle supplémentaire dans l'algorithme puisqu'on doit examiner tous les nombres impairs $a \geq 3$ comme des bases potentielles pour le test de Fermat. Nous supprimons de ces bases potentielles les nombres qui ne sont pas premiers avec le nombre testé (d'où l'utilisation de notre fonction *pgcd*() dans cet algorithme). Le mot-clef *continue* permet ici de poursuivre la boucle *for* lorsqu'on tombe sur une base potentielle qui n'est pas première avec le nombre testé. Les premiers nombres de Carmichael sont donc : 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, etc.

```
#recherche des nombres pseudo-premiers de Carmichael
Lprem=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,\
       53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101]#quelques nombres premiers
nbrTrouve=0
n_teste=3
isPrem=False
while nbrTrouve<10:
    isPrem=isPremier(n_teste)
    if isPrem is False :
        isCarmichael=True
        for i in range(2,n_teste) :
            if pgcd(i,n_teste)!=1 : continue
            if (i**(n_teste-1))%n_teste!=1 :
                isCarmichael=False
                break
        if isCarmichael is True :
            nbrTrouve+=1
            print('{} est le nombre de Carmichael n°{}'.format(n_teste,nbrTrouve))
    n_teste+=2
```

```
561 est le nombre de Carmichael n°1
1105 est le nombre de Carmichael n°2
1729 est le nombre de Carmichael n°3
2465 est le nombre de Carmichael n°4
2821 est le nombre de Carmichael n°5
6601 est le nombre de Carmichael n°6
8911 est le nombre de Carmichael n°7
10585 est le nombre de Carmichael n°8
15841 est le nombre de Carmichael n°9
29341 est le nombre de Carmichael n°10
```

On remarque que ces nombres sont tous des nombres de Poulet (c'est une évidence car la condition pour être un nombre de Poulet (passer le test pour la base 2) est contenue dans celle, plus sévère, pour être un nombre de Carmichael). Ainsi, le 10ème nombre de Carmichael est le 40ème nombre de Poulet (voir le tableau ci-dessous).
On a la preuve depuis 1994 qu'il y a une infinité de nombres de Carmichael, et qu'ils sont tous composés d'au moins trois nombres premiers (561=3×11×17, 1105=5×13×17, 1729=7×13×19, etc.). On sait aussi que les produits $(6k+1)(12k+1)(18k+1)$ quand ces trois facteurs sont premiers sont les nombres de Chernick, et

rang Poulet	nombre	rang Carmichael	rang Chernick	Décomposition	rang Poulet	nombre	rang Carmichael	rang Chernick	Décomposition
1	341			11×31	26	12 801			3×17×251
2	561	1		3×11×17	27	13 741			7×13×151
3	645			3×5×43	28	13 747			59×233
4	1 105	2		5×13×17	29	13 981			11×31×41
5	1 387			19×73	30	14 491			43×337
6	1 729	3	1	7×13×19	31	15 709			23×683
7	1 905			3×5×127	32	15 841	9		7×31×73
8	2 047			23×89	33	16 705			5×13×257
9	2 465	4		5×17×29	34	18 705			3×5×29×43
10	2 701			37×73	35	18 721			91×193
11	2 821	5		7×13×31	36	19 951			71×281
12	3 277			29×113	37	23 001			3×11×17×41
13	4 033			37×109	38	23 377			97×241
14	4 369			17×257	39	25 761			3×31×277
15	4 371			3×31×47	40	29 341	10		17×37×61
16	4 681			31×151	41	30 121			7×13×331
17	5 461			43×127	42	30 889			17×23×79
18	6 601	6		7×23×41	43	31 417			89×353
19	7 957			73×109	44	31 609			73×433
20	8 321			53×157	45	31 621			103×307
21	8 481			3×11×257	46	33 153			3×43×257
22	8 911	7		7×19×67	47	34 945			5×29×241
23	10 261			31×331	48	35 333			89×397
24	10 585	8		5×29×73	49	39 865			5×7×17×67
25	11 305			5×7×17×19	50	41 041	11		7×11×13×41

que ce sont tous des nombres de Carmichael. Par exemple, 1729=(6+1)(12+1)(18+1) est le 1er nombre de Chernick (il suffit de prendre k=1). Les nombres de Chernick sont beaucoup plus rares que les nombres de Carmichael, mais plus simples à obtenir ; le suivant est obtenu pour k=6 (il s'agit de 294 409) et ensuite il faut attendre k=35 pour trouver le 3ème (il s'agit de 56 052 361). Certains nombres de la forme $(6k+1)(12k+1)(18k+1)$ sont des nombres de Carmichael sans être des nombres de Chernick, le premier étant 172 081 qui s'écrit 31×61×91 avec 31 et 61 premiers, mais 91 n'est pas premier (91=7×13). On peut remarquer sur notre tableau que toutes les décompositions de nombres de Poulet ne font intervenir que des facteurs premiers différents (pas d'exposant supérieur à 1).

Partie n°4 : Récursivité et efficacité

a) Écrire un programme itératif (utilisant uniquement des boucles *for* et *while*) qui calcule le PGCD de deux nombres entiers *a* et *b* en traduisant l'algorithme d'Euclide. Écrire ce programme en mode récursif (utilisant une fonction qui s'appelle elle-même).

Le mode itératif (ou séquentiel) est le mode habituel, celui que l'on apprend au début : il utilise des boucles *for* et/ou des boucles *while* et c'est tout. La fonction *pgcd*() que nous avons écrit (à gauche) est une traduction de l'algorithme d'Euclide dans ce mode : on effectue des divisions euclidiennes sans se préoccuper du quotient, en utilisant juste le reste *reste=a%b*. Dans la boucle *while*, on remplace (*a,b*) par (*b,reste*) jusqu'à ce que *reste*==0 soit *True* (tant que le reste est différent de zéro). Le PGCD cherché est *b*, le dernier diviseur.

```
def pgcd(a,b): # forme séquentielle
    reste=a%b
    while reste!=0 :
        a,b=b,reste
        reste=a%b
    return b
```

```
def pgcd(a,b): # forme récursive
    reste=a%b
    if reste==0 : return b
    else : return pgcd(b,reste)
```

```
A=int(input('PGCD de A (un entier positif) : '))
B=int(input('et B (un autre): '))
if B>A : A,B=B,A
print('PGCD({},{})={}'.format(A,B,pgcd(A,B)))
```

```
PGCD de A (un entier positif) : 12345
et B (un autre): 67890
PGCD(67890,12345)=15
```

La récursivité simplifie souvent l'écriture d'une fonction, surtout quand celle-ci est, par nature, récursive et c'est le cas de l'algorithme d'Euclide avec la propriété PGCD(*a,b*)=PGCD(*b,reste*) tant que *reste*≠0. La traduction en mode récursif (à droite) de cet algorithme utilise cette propriété dans le cas général avec un test d'arrêt qui examine la valeur du reste et arrête le processus quand ce reste est nul. Ainsi pgcd(12,8) calcule 12%8=4 et comme 4≠0 retourne pgcd(8,4) qui calcule 8%4=0 et comme 0=0 retourne 4 qui se substitue à pgcd(8,4) puis à pgcd(12,8) comme valeur de retour.

Ces deux algorithmes donnent exactement le même résultat (heureusement!) et le programme principal fonctionne aussi bien avec l'un qu'avec l'autre. La boucle récursive est un peu plus courte à écrire mais si on mesure très précisément les temps d'exécution, elle est un peu plus longue à s'exécuter. Il faut tout de même prendre des valeurs assez grandes pour que cette différence soit sensible : un peu plus d'un dix millièmes de seconde d'écart pour déterminer que PGCD(573147844013817084101,354224848179261915075)=1 (nous avons pris le 101ème et le 100ème terme de la suite de Fibonacci car, à nombres de tailles égales, c'est pour les termes consécutifs de cette suite qu'il y a le plus de boucles à effectuer ; ici, il y a 99 divisions à effectuer pour les deux programmes).

```
PGCDitératif(48,6)=6, réponse en 8.062855086305534e-06 secondes
PGCDrécursif(48,6)=6, réponse en 9.52882873836089e-06 secondes

PGCDitératif(123456,789)=3, réponse en 1.2460776042472189e-05 secondes
PGCDrécursif(123456,789)=3, réponse en 2.272259160686005e-05 secondes

PGCDitératif(100000000001,10000001)=11, réponse en 1.1727789216444412e-05 secondes
PGCDrécursif(100000000001,10000001)=11, réponse en 1.759168382466439e-05 secondes

PGCDitératif(573147844013817084101,354224848179261915075)=1, réponse en 9.235634007949975e-05 secondes
PGCDrécursif(573147844013817084101,354224848179261915075)=1, réponse en 0.00023382279750286047 secondes
```

On peut dire que cette différence d'efficacité n'est pas significative. Elle ne pénalise pas le programme le plus lent, même si, lorsque les nombres augmentent, le retard s'accroit légèrement aussi.

b) Parfois l'écriture récursive est moins efficace que son équivalent itératif, car les appels multiples à la fonction créent autant d'environnements d'exécution qu'il en faut. Ces environnements s'ignorant les uns les autres peuvent être redondants. Pour mieux comprendre cela, programmer le calcul d'un terme de la célèbre suite de Fibonacci : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. (chacun des termes étant la somme des deux précédents) dans les deux modes (itératif et récursif). Mesurer les temps d'exécution avec la fonction *clock*() du module *time* : cette fonction renvoie un temps très précis de l'horloge interne en secondes ; pour mesurer une durée, il faut faire la différence entres le temps de fin et de début d'exécution.

La programmation en est simple dans les deux modes, mais le récursif est le plus simple des deux car il colle mieux à la définition du terme de rang *n* comme la somme des termes de rang *n*−1 et *n*−2. Par contre, l'exécution de la fonction récursive (*fibo2*() dans notre illustration) prend de plus en plus de retard par rapport à la fonction itérative quand *n* augmente. Pour le calcul du 120ème terme de la suite, la fonction

itérative répond presque instantanément, alors que la fonction récursive est au bord du dépassement de capacité et prend plusieurs dizaines de minutes pour s'exécuter. Pourquoi cela arrive t-il ? Parce qu'à chaque appel récursif, Python crée deux nouveaux environnements d'exécution qui chacun lancent deux nouveaux environnement d'exécution qui chacun… *fibo2*(120) demande le calcul de *fibo2*(119) et *fibo2*(118) ; jusque là tout va bien, sauf que *fibo2*(119) demande le calcul de *fibo2*(118) et *fibo2*(117), en ignorant complètement le fait que *fibo2*(118) a déjà été demandé… Et le processus continue créant des redondances : *fibo2*(118) est demandé 2 fois, *fibo2*(117) est demandé 3 fois, *fibo2*(116) est demandé 5 fois, … (cette progression semble emboîter le pas d'une suite de Fibonacci).

```
def fibo1(n): # forme itérative
    if n<2 : return n
    else :
        k,a,b=1,1,0
        while k<n :
            a,b=a+b,a
            k+=1
        return a

def fibo2(n): # forme récursive
    if n<2 : return n
    else : return fibo2(n-1)+fibo2(n-2)

from time import clock
def fiboDuree1(n) :
    t=clock()
    return fibo1(n),clock()-t
def fiboDuree2(n) :
    t=clock()
    return fibo2(n),clock()-t
```

```
Entrer le rang du terme de la suite de Fibonacci voulu : 10
FIBO(10)=55, réponse en 7.3298682602777594e-06 secondes
FIBO(10)=55, réponse en 0.00013193762868499853 secondes

Entrer le rang du terme de la suite de Fibonacci voulu : 20
FIBO(20)=6765, réponse en 1.2460776042472192e-05 secondes
FIBO(20)=6765, réponse en 0.016674717305305876 secondes

Entrer le rang du terme de la suite de Fibonacci voulu : 30
FIBO(30)=832040, réponse en 1.9790644302749945e-05 secondes
FIBO(30)=832040, réponse en 2.07831744340059 secondes

Entrer le rang du terme de la suite de Fibonacci voulu : 40
FIBO(40)=102334155, réponse en 2.565453891097215e-05 secondes
FIBO(40)=102334155, réponse en 255.03580347450418 secondes
```

```
n=int(input('Entrer le rang du terme de la suite de Fibonacci voulu : '))
print('FIBO({})={}, réponse en {} secondes'.format(n,fiboDuree1(n)[0],fiboDuree1(n)[1]))
print('FIBO({})={}, réponse en {} secondes'.format(n,fiboDuree2(n)[0],fiboDuree2(n)[1]))
```

Pour pallier à ce défaut majeur du mode récursif naïf (qui ne se rencontre pas pour toutes les fonctions récursives, il faut le dire), on doit éviter ce genre d'appels multiples, et réaliser des fonctions récursives qui ne s'appellent qu'une seule fois (pour éviter les redondances). Voici le calcul des termes de la suite de Fibonacci par un algorithme récursif de ce genre, qualifié ici de « terminal » (car les calculs s'effectuent directement en descendant dans la pile d'exécution, au lieu de se faire en remontant). Les temps de réponses sont donnés à droite pour comparer avec les deux autres méthodes.
L'astuce, pour ne lancer qu'une seule fonction *fibo3*() à chaque fois (au lieu de deux pour la fonction *fibo2*()) consiste en l'insertion de deux arguments supplémentaires (les deux termes qui précèdent celui qu'on doit calculer) qui sont calculés avant l'envoi de la fonction : au démarrage, *fibo3*(120) contient des arguments par défaut *a*=0 et *b*=1 (les deux premiers termes de la suite) et demande le calcul de *fibo3*(119,1,2) qui demande *fibo3*(118,2,3), etc. jusqu'à *fibo3*(1, 331164814351698201718008 1, 535835925499096664087184 0) qui renvoie le résultat final 120!=535835925499096664087184 0 car, alors, *n*=1.

```
def fibo3(n,a=0,b=1): # forme récursive terminale
    if n==1 : return b
    else : return fibo3(n-1,b,a+b)
```

```
FIBO(10)=55, réponse en 1.24607760424722e-05 secondes
FIBO(20)=6765, réponse en 2.638752573699832e-05 secondes
FIBO(30)=832040, réponse en 3.591635447541819e-05 secondes
FIBO(40)=102334155, réponse en 4.83771305178348 76e-05 secondes
```

La notation (*n*, *a*=0, *b*=1) dans les arguments de la fonction signifie que 0 et 1 seront affectés par défaut aux variables *a* et *b* si ces valeurs ne sont pas renseignées lors de l'appel. Quand nous appelons cette fonction par *fibo3*(*n*) en ne renseignant pas les deux derniers paramètres, ce sont les valeurs par défaut qui seront prises. Par la suite, les appels *fibo3*(*n-1,b,a+b*) renseignent les valeurs de *a* et de *b* ce qui désactive les affectations par défaut.

c) Écrire un programme itératif qui calcule la factorielle $n! = 1\times 2\times ...\times n = \prod_{k=1}^{n} k$ d'un nombre entier *n*.
En déduire un programme qui calcule les coefficients binomiaux $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, aussi appelés (en dénombrement) nombres de combinaisons de *k* éléments pris parmi *n*, ou coefficients du triangle de Pascal. Écrire ces deux programmes en mode récursif (utilisant une fonction qui s'appelle elle-même).
Disons le tout de suite : dans le module *math* de Python, la fonction factorielle existe ! Il suffit de l'appeler avec l'instruction *factorial*(3) (après avoir importé les fonctions de ce module par *from math import* *). Ce

n'est donc qu'à titre d'exercice (comme souvent, comme pour le PGCD qui existe aussi, bien sûr, dans les bibliothèques de Python) que nous chercherons à programmer par nous-même cette fonction.

```
>>> from math import *
>>> factorial(3)
6
```

En mode itératif, il suffit de faire varier *i* de 1 à *n* dans l'affectation *fact=fact*i* qui s'écrit aussi *fact*=i*.

```
fact=1
for i in range(1,11) : fact*=i
print(fact)
```
```
3628800
```

On ne doit pas non plus oublier d'initialiser *fact* à 1. Le programme ci-contre suffit pour calculer 10!
Pour un affichage plus propre qui nous rappelle la valeur de *n*, la variable *n* est nommée (affectée en mémoire) car elle est utilisée deux fois, par exemple dans une instruction qui demande à l'opérateur d'entrer cette variable. On peut étoffer la dernière ligne avec du texte : *print*('{}!={}'.*format*(n,fact)) ou *print*(*str*(n)+'!='+*str*(fact)). Pour une réutilisation ultérieure, on écrit une fonction *factorielle*() qui se charge du calcul, tandis que le programme principal gère les entrées-sorties.

```
def factorielle(nbr):
    if(nbr<=0 or int(nbr)!=nbr): return -1
    f=1
    for i in range(1,nbr+1) : f*=i
    return f

n=int(input('factorielle de ? '))
print('{}!={}'.format(n,factorielle(n)))
```
```
factorielle de ? 365
365!=25104128675558732292929443748812027705165520269876079766872595193901106138220937419666018009000254169376172314360982328660708071123369979853445367910653872383599704355532740937678091491429440864316046925074510134847025546014098005907965541041195496105311886173373435145517193282760847755882291690213539123479186274701519396808504940722607033001246328398800550487427999876690416973437861078185344667966871511049653888130136836199010529180056125844549488648615193390110613822093741966601800900025416937617231436098232866070807112336997985344536791065387238359970435553274093767809149142944086431604692507451013484702554601409800590796554104119549610531188617337343514551719328276084775588229169021353912347918627470151939680850494072260703300124632839880055048742799987669041697343786107818534466796687151104965388813013683619901052918005612584454948864861
```

```
def factorielle(nbr):# itératif
    if(nbr<=0 or int(nbr)!=nbr): return -1
    f=1
    for i in range(1,nbr+1) : f*=i
    return f
```

```
def coeffBi(n,k): # itératif amélioré
    coeff=1
    for i in range(1,k+1) : coeff*=(n-i+1)/i
    return int(coeff)
```

```
from time import clock
def coeffBiDuree1(n,k) :
    t=clock()
    coeff=int(factorielle(n)/(factorielle(k)*factorielle(n-k)))
    return coeff,clock()-t
```

```
def coeffBiDuree2(n,k) :
    t=clock()
    coeff=coeffBi(n,k)
    return coeff,clock()-t
```

```
n=int(input('Entrer n (le plus grand) : '))
k=int(input('Entrer k (le plus petit) : '))
coeff,duree=coeffBiDuree1(n,k)
print('coefficient binomial itera3({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
coeff,duree=coeffBiDuree2(n,k)
print('coefficient binomial amelio({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
```
```
Entrer n (le plus grand) : 150
Entrer k (le plus petit) : 29
coefficient binomial itera3(150,29)=79831699191308838887832372346880, réponse en 0.00019057657476722172 secondes
coefficient binomial amelio(150,29)=79831699191308883891431999717376, réponse en 6.963374847263754e-05 secondes
```

Pour le calcul du coefficient binomial en mode itératif $\binom{n}{k}=\frac{n!}{k!(n-k)!}$, c'est très facile : on utilise trois fois la fonction *factorielle*() précédente sans se demander si la fonction existe déjà dans un des modules de Python. Mais on peut faire légèrement mieux pour calculer $\frac{n!}{k!(n-k)!}=\frac{n\times(n-1)\times(n-2)\ldots\times(n-k+1)}{k\times(k-1)\times(k-2)\ldots\times(1)}$ car il y a autant de facteurs au numérateur qu'au dénominateur (il y en a *k*), par exemple $\frac{6!}{4!(6-4)!}=\frac{6\times5\times4\times3}{4\times3\times2\times1}$ (il y a 4 facteurs). On peut donc exécuter une seule boucle en faisant varier *i* de 1 à *k* dans l'affectation *coeff=coeff*(n-i*+1)/*i* qui s'écrit aussi *coeff*=(n-i*+1)/*i*. Cette nouvelle version itérative de calcul du coefficient binomial est plus efficace car elle se passe de la fonction *factorielle*() (dans la formule, les trois factorielles servent davantage à simplifier l'expression qu'à effectuer concrètement les calculs).

La traduction de la fonction *factorielle*() en une fonction récursive (qui s'appelle elle-même) ne pose pas de problème. C'est l'exemple typique (avec l'algorithme d'Euclide) qui est donné pour illustrer ce qu'est une fonction récursive. Le principe est que l'on doit mettre la condition d'arrêt au cœur de la fonction, généralement c'est une simple instruction conditionnelle qui réalise cela.

```
def factorielle(nb):
    if nb==1 : return 1
    else : return nb*factorielle(nb-1)
```

```
n=int(input('factorielle de ? '))
print('{}!={}'.format(n,factorielle(n)))
```

Si la récursivité est une nouveauté pour vous, prenez le temps d'analyser ce que fait cette fonction : prenons un petit exemple, pour *n*=3. Le programme demande *factorielle*(3), au test *nb*==1, la réponse est *False* donc la fonction retourne 3×*factorielle*(2), mais le fait d'écrire *factorielle*(2) lance la fonction qui, au test *nb*==1 répond *False*, retourne 2×*factorielle*(1), mais le fait d'écrire *factorielle*(1) lance la fonction qui, au test *nb*==1 répond ***True*** et retourne 1. Ce 1 renseigne le calcul 2×*factorielle*(1) qui vaut maintenant 2, ce qui renseigne le calcul 3×*factorielle*(2) qui vaut maintenant 6, ce qui est retourné au programme principal.

```
def factorielle(nbr):
    if(nbr<=0 or int(nbr)!=nbr): return -1
    f=1
    for i in range(1,nbr+1) : f*=i
    return f

n=int(input('Entrer n (le plus grand) : '))
k=int(input('Entrer k (le plus petit) : '))
coeff=factorielle(n)/(factorielle(k)*factorielle(n-k))
print('coefficient binomial({},{})={}'.format(n,k,coeff))
```

```
Entrer n (le plus grand) : 37
Entrer k (le plus petit) : 5
coefficient binomial(37,5)=435897.0

Entrer n (le plus grand) : 100
Entrer k (le plus petit) : 36
coefficient binomial(100,36)=1.977204582144933e+27

Entrer n (le plus grand) : 365
Entrer k (le plus petit) : 250
coefficient binomial(365,250)=2.6547215504137518e+97
```

Pour calculer les coefficients binomiaux en mode récursif, on va se servir d'une de leurs propriétés : $\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$.

Cette propriété est souvent illustrée par la disposition du « triangle de Pascal » qui dispose les coefficients binomiaux en un tableau triangulaire. On y calcule un des nombres en faisant la somme des deux nombres situés à la ligne d'avant, même colonne et colonne de gauche.

n/k	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Par exemple, $\binom{7}{2}=\binom{6}{1}+\binom{6}{2}=6+15=21$ (21 est au croisement de la ligne 7 colonne 2, il est égal à la somme de 6 qui se trouve au croisement de la ligne 6 colonne 1 et de 15 qui se trouve au croisement de la ligne 6 colonne 2. Bien sûr, pour remplir le tableau ainsi, il faut savoir que sur la colonne 0 il n'y a que des 1comme, de même, sur la diagonale du tableau où le n° de ligne égale le n° de colonne.

Bien sûr, on pourrait utiliser cette propriété sur un mode itératif (un peu compliqué à mettre en place avec les deux indices de ligne et de colonne), mais ici, nous allons l'employer en mode récursif car la traduction en est immédiate (on n'a pas besoin de gérer des indices). Cela s'écrit comme la propriété l'énonce car il s'agit d'un relation de récurrence qui définit un terme d'une suite en fonction de termes voisins qui ont eux mêmes été calculés de la même façon, à partir des premiers termes qui sont connus.

```
def coeffBin(li,col): #mode récursif
    if col==0 or col==li : return 1
    return coeffBin(li-1,col-1)+coeffBin(li-1,col)

n=int(input('Entrer n (le plus grand) : '))
k=int(input('Entrer k (le plus petit) : '))
print('coefficient binomial({},{})={}'.format(n,k,coeffBin(n,k)))
```

L'inconvénient de cette définition récursive se fait sentir si on veut calculer des termes d'indices trop élevés. Regardez les temps d'exécution de cette méthode récursive *coeffBin3*() pour le calcul de $\binom{25}{13}$: 10,5 secondes ! Les méthodes 1 et 2 qui utilisent les définitions itératives et récursives de la factorielle mettent respectivement 0,00005 et 0,0002 secondes approximativement pour faire ce calcul. L'explication de ce temps très long pour la méthode récursive est la même que pour le calcul des termes de la suite de Fibonacci : des appels redondants qui ralentissent la fonction de plus en plus quand *n* augmente jusqu'à l'empêcher (si les nombres en Python ne sont pas limités, la pile d'exécution, qui se chargent des calculs effectués dans tous les environnements cloisonnés d'exécution, a une taille limitée qui provoque l'arrêt des calculs en cas de débordement). Il faut écrire une fonction récursive qui ne s'appelle qu'une fois. Nous avons écrit cette fonction « terminale », *coeffBin4*(), en se servant de la relation déjà employée pour améliorer l'algorithme itératif entre deux coefficients voisins d'une même colonne : $\binom{n}{k}=\binom{n}{k-1}\times\frac{n-k+1}{k}$.

La condition d'arrêt est l'arrivée dans les colonnes 1 ou 0 dans lesquelles on trouve toujours *n* (pour la colonne 1) car $\binom{n}{1}=\frac{n!}{1!(n-1)!}=n$ ou 1 (pour la colonne 0).

Notons au passage que nos fonctions *factorielle*() ne donnent pas une valeur correcte pour 0 ! (l'itérative *factorielle1*() donne -1 alors que la récursive *factorielle2*() conduit à une belle boucle infinie). La valeur correcte de 0 ! est 1. Avec cette valeur, la formule du coefficient binomial donne toujours une valeur correcte.

```
def coeffBin1(li,col): # mode itératif
    return factorielle1(n)//(factorielle1(k)*factorielle1(n-k))
def factorielle1(nbr): # factorielle itératif
    if(nbr<=0 or int(nbr)!=nbr): return -1
    f=1
    for i in range(1,nbr+1) : f*=i
    return f

def coeffBin2(li,col): # mode pseudo-récursif
    return factorielle2(n)//(factorielle2(k)*factorielle2(n-k))
def factorielle2(nb): # factorielle récursif
    if nb==1 : return 1
    else : return nb*factorielle2(nb-1)

def coeffBin3(li,col): # mode récursif
    if col==0 or col==li : return 1
    return coeffBin3(li-1,col-1)+coeffBin3(li-1,col)

def coeffBin4(li,col): # mode récursif terminal
    if col==0 : return 1
    elif col==1 : return li
    return coeffBin4(li,col-1)*(li-col+1)//col
```

```
from time import clock
def coeffDuree1(li,col) :
    t=clock()
    return coeffBin1(li,col),clock()-t
def coeffDuree2(li,col) :
    t=clock()
    return coeffBin2(li,col),clock()-t
def coeffDuree3(li,col) :
    t=clock()
    return coeffBin3(li,col),clock()-t
def coeffDuree4(li,col) :
    t=clock()
    return coeffBin4(li,col),clock()-t

n=int(input('Entrer n (le plus grand) : '))
k=int(input('Entrer k (le plus petit) : '))
coeff,duree=coeffDuree1(n,k)
print('coeff1 iteratif({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
coeff,duree=coeffDuree2(n,k)
print('coeff2 pseudore({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
coeff,duree=coeffDuree3(n,k)
print('coeff3 recursif({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
coeff,duree=coeffDuree4(n,k)
print('coeff4 terminal({},{})={}, réponse en {} secondes'.format(n,k,coeff,duree))
```

```
Entrer n (le plus grand) : 25
Entrer k (le plus petit) : 13
coeff1 iteratif(25,13)=5200300, réponse en 5.3508038300027636e-05 secondes
coeff2 pseudore(25,13)=5200300, réponse en 0.0001847126801590021 secondes
coeff3 recursif(25,13)=5200300, réponse en 10.514033399277714 secondes
coeff4 terminal(25,13)=5200300, réponse en 4.397920956122903e-05 secondes
```

d) Le mode récursif est réputé plus efficace que le mode itératif dans certains cas. Pour trier une liste de nombres initialement dans le désordre, on peut procéder ainsi : choisir un des nombres comme pivot, comparer les nombres de la liste au pivot et les ranger dans deux listes distinctes (les nombres inférieurs dans *listInf* et les nombres supérieurs dans *listSup*), appliquer la même méthode de tri sur les deux listes. Écrire le programme récursif qui effectue le tri de cette façon. Tester ce programme sur une liste de *n* nombres tirés au hasard. Écrire un programme itératif qui réalise le même travail et comparer les deux sur leurs performances respectives.

L'algorithme de tri rapide proposé (*quick sort* en anglais) demande de choisir un pivot qui peut être choisi au hasard (pivot=*random.choice*(liste)) ; on peut aussi bien choisir le pivot toujours au début (pivot=liste[0]), ou bien au milieu approximativement de la liste (pivot=liste[*len*(liste)//2]). On pourra tester ces trois options. La procédure suivie est par nature récursive : on compare les éléments de la liste au pivot ; s'ils y sont inférieurs, on les range dans une *listeInf*, sinon dans une *listeSup*. S'ils sont égaux au pivot on les range dans une 3ème liste (*listeMid*). Ensuite, on renvoie le résultat de la concaténation des 3 listes triées (la *listeMid* n'a pas besoin d'être triée). Donc on relance la fonction de tri d'une liste, mais avec des listes plus petites. La condition d'arrêt du tri porte sur la taille de la liste à trier qui ne peut être égale à 0 ou 1 (il n'y aurait alors rien à trier) : dans ces deux derniers cas, on renvoie la liste reçue. En remontant dans la pile d'exécution, les listes triées se concatènent pour former la liste triée finale.

```
def tri1(liste): #  mode itératif
    global nbComp
    listInf,listSup,listMid=list(),list(),list()
    pivot=liste[len(liste)//2]
    for element in liste :
        nbComp+=1
        if   element<pivot :
            rang=0
            for inf in listInf :
                nbComp+=1
                if inf>element : break
                rang+=1
            listInf.insert(rang,element)
        elif element>pivot :
            rang=0
            for sup in listSup :
                nbComp+=1
                if sup>element : break
                rang+=1
            listSup.insert(rang,element)
        else : listMid.append(element)
    return listInf+listMid+listSup
```

```
def tri2(liste): #  mode récursif
    global nbComp
    listInf,listSup,listMid=list(),list(),list()
    if len(liste)<2 :return liste
    pivot=liste[len(liste)//2]
    for element in liste :
        nbComp+=1
        if   element<pivot :
            listInf.append(element)
        elif element>pivot :
            listSup.append(element)
            nbComp+=1
        else :
            listMid.append(element)
            nbComp+=1
    return tri2(listInf)+listMid+tri2(listSup)
```

```
Entrer la taille de la liste à trier : 20
liste initiale de taille 20 :
[18, 18, 11, 1, 18, 5, 6, 20, 15, 13, 11, 14, 15, 14, 7, 15, 17, 15, 11, 19]
Mode itératif : liste triée en 0.00021403215320011054 secondes après 120 comparaisons
[1, 5, 6, 7, 11, 11, 11, 13, 14, 14, 15, 15, 15, 15, 17, 18, 18, 18, 19, 20]
Mode récursif : liste triée en 0.00026094331006588575 secondes après 48 comparaisons
[1, 5, 6, 7, 11, 11, 11, 13, 14, 14, 15, 15, 15, 15, 17, 18, 18, 18, 19, 20]
```

```
Entrer la taille de la liste à trier : 100
Méthode du pivot - 1: choix du pivot au milieu de la liste
Mode itératif : 400 listes de 100 nombres triées en 0.00147100377414917 s/liste et 1841.3875 comparaisons/liste en moyenne.
Mode récursif : 400 listes de 100 nombres triées en 0.001229882260326133 s/liste et 976.3125 comparaisons/liste en moyenne.


Entrer la taille de la liste à trier : 500
Méthode du pivot - 1: choix du pivot au milieu de la liste
Mode itératif : 400 listes de 500 nombres triées en 0.0295309507352224 s/liste et 42089.1 comparaisons/liste en moyenne.
Mode récursif : 400 listes de 500 nombres triées en 0.008065242790891304 s/liste et 7136.35 comparaisons/liste en moyenne.
```

Le mode itératif copie le fonctionnement de ce tri par pivot. Mais il est difficile, voire impossible, de faire exactement la même chose qu'en mode récursif. Nous avons opté pour une solution qui compare les éléments de la liste au pivot et les insère dans la liste qui convient (*listeInf, listeSup* ou *listeMid*), au bon endroit, créant ainsi trois listes triées qu'il suffit de concaténer.
Le programme récursif est le plus efficace, et cette efficacité augmente quand la taille de la liste augmente. Cela vient du nombre de comparaisons à faire qui devient énorme en mode itératif comparativement au mode récursif 42 089 comparaisons en moyenne pour 500 nombres à trier en mode itératif alors qu'en mode récursif, il faut 7136 comparaisons en moyenne seulement (6 fois moins). Avec 100 nombres à trier, le rapport n'est que de 1,9 mais pour 2500 nombres, le rapport est de 22 (récursif : 47 561 comparaisons - itératif : 1 038 642 comparaisons). Ces nombres moyens de comparaisons sont à rapprocher des durées moyennes qui suivent globalement la même progression. Voici les chiffres obtenus lorsque le pivot est choisi au milieu de la liste (tableau ci-dessous).

1: choix du pivot au milieu				
Longueur de la liste	20	100	500	2500
Mode itératif				
nombre moyen de comparaisons	90	1841	42089	1038642
Durée moyenne (s)	0,0001067	0,001471	0,02953	0,7336
Mode récursif				
nombre moyen de comparaisons	112	976	7136	47561
Durée moyenne (s)	0,0001834	0,001229	0,00807	0,05072
Rapport itératif/récursif				
a=nombres moy. de comparaisons	0,80	1,89	5,90	21,84
b=durées moyennes	0,58	1,20	3,66	14,46
a/b	1,38	1,58	1,61	1,51

Nous pouvons recommencer ces mesures pour un autre choix du pivot. Est-ce qu'il y a une place optimale pour le pivot ou bien est-ce approximativement toujours la même efficacité moyenne (le nombre de comparaisons et la durée du tri dépendent évidemment de la liste initiale : si elle est déjà triée, il y aura forcément beaucoup moins de comparaisons). Il semblerait que les deux méthodes ont des efficacités plus proches lorsque le pivot est fixé au rang 0 (comparer les lignes a/b pour les trois options de choix du pivot) et plus éloignées lorsque le pivot est choisi de façon aléatoire. L'option 1 (choix au milieu de la liste) donne des résultats intermédiaires, mais c'est la meilleure option pour la méthode itérative (du moins pour des listes longues). La meilleure option pour la méthode récursive est moins flagrante : en durée c'est plus rapide avec l'option 3 mais en nombre de comparaisons, ce serait plutôt l'option 1 qui l'emporterait ; disons que les trois options sont sensiblement à égalité.

2: choix aléatoire du pivot				
Longueur de la liste	20	100	500	2500
Mode itératif				
nombre moyen de comparaisons	90	1802	43193	1044953
Durée moyenne (s)	0,0001119	0,001446	0,03017	0,7409
Mode récursif				
nombre moyen de comparaisons	112	973	7181	47390
Durée moyenne (s)	0,0002521	0,001588	0,00989	0,0603
Rapport itératif/récursif				
a=nombres moy. de comparaisons	0,80	1,85	6,01	22,05
b=durées moyennes	0,44	0,91	3,05	12,29
a/b	1,81	2,03	1,97	1,79

3: choix du pivot fixé au rang 0				
Longueur de la liste	20	100	500	2500
Mode itératif				
nombre moyen de comparaisons	90	1796	42358	1059473
Durée moyenne (s)	0,0001047	0,001455	0,02981	0,7598
Mode récursif				
nombre moyen de comparaisons	114	963	7145	47923
Durée moyenne (s)	0,0001766	0,001196	0,00784	0,05
Rapport itératif/récursif				
a=nombres moy. de comparaisons	0,79	1,87	5,93	22,11
b=durées moyennes	0,59	1,22	3,80	15,20
a/b	1,33	1,53	1,56	1,45

Nous n'avons envisagé qu'un seul algorithme de tri mais il en existe de nombreux, la situation de tri étant très fréquente dans les traitements informatiques. Python a, bien sûr, implémenté un algorithme de tri pour les listes. Nous avons souvent utilisé la méthode *sorted*(). Par exemple *sorted*([2,6,8,1,6,7,2,9,88,1]) renvoie la liste triée : [1, 1, 2, 2, 6, 6, 7, 8, 9, 88]. Mesurons la performance de l'algorithme utilisé par Python dans cette implémentation du tri : les chiffres parlent d'eux-mêmes, pour une liste de 2500 nombres à trier, la méthode *sorted*() est 25 fois plus rapide que notre meilleur choix en mode récursif ! Nous devons mentionner un handicap pour nos méthodes : afin de mesurer le nombre de comparaisons effectuées, nous avons incrémenté un compteur *nbComp* à chaque comparaison. Cela peut-il suffire à expliquer le fossé qu'il y a entre ces deux méthodes ? Pour répondre, il n'y a qu'à supprimer ce compteur.

Tri de Python avec sorted()				
Longueur de la liste	20	100	500	2500
Durée moyenne (s)	0,00001227	0,000064	0,0004	0,0026

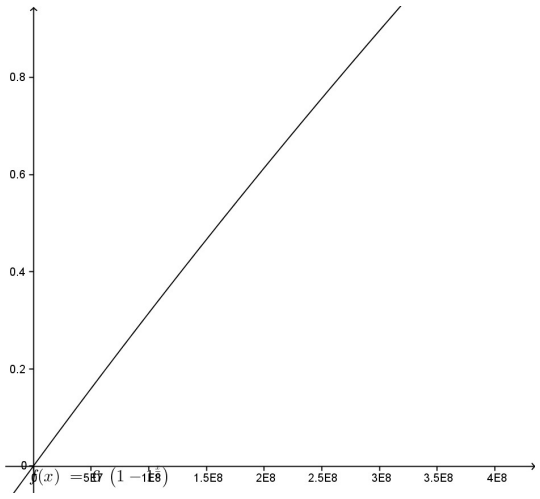
```
Entrer la taille de la liste à trier : 2500
Méthode du pivot - 3: choix du pivot fixé au rang 0
Mode récursif : 400 listes de 2500 nombres triées en 0.03211547327859863 s/liste en moyenne.
Tri de Python par sorted(): 400 listes de 2500 nombres triées en 0.0025310584842859053 s/liste en moyenne.
```

Au lieu de 0,0500 seconde par liste de 2500 nombres, notre méthode récursive traite maintenant une liste en 0,0321 seconde quand *sorted*() de Python ne prend que 0,0025 seconde. Il y a eu un gain d'efficacité en enlevant ce compteur mais la méthode native de Python est tout de même près de 13 fois plus rapide...

e) Pour prolonger la réflexion sur les méthodes récursives de programmation, nous avons vu qu'il faut éviter les redondances qui sont la plaie des programmes récursifs (du moins pour des traitements de grande ampleur). Comment alors programmer la situation suivante où on a calculé qu'une probabilité p_n vaut $p_n = p_{n-1} + \frac{1-p_{n-6}}{26^6}$? Il s'agit de la probabilité d'écrire un mot spécifique de six lettres comme « COGITO » (*je pense*, en latin) lorsqu'on tape n lettres au hasard[1]. La situation est aggravée par le fait que n est nécessairement grand (n=2000 est un minimum envisageable) sinon p_n va être ridiculement petit.
La définition qui nous est donnée de p_n est une définition par récurrence, comme B. Rittaud le souligne dans son livre. Elle permet, connaissant les premiers termes (ici on doit connaître p_1, p_2, p_3, p_4, p_5 et p_6), de déterminer les suivants. Ici, le 5 premiers termes de cette suite sont nuls (on ne peut pas écrire COGITO qui contient 6 lettres en n'écrivant que 5, 4, 3, 2 ou 1 lettre). Par contre p_6 vaut $p_6 = \frac{1}{26^6} \approx 3{,}237 \times 10^{-9}$ puisqu'il y a $26^6 = 308915776$ mots de 6 lettres choisies au hasard parmi 26 (on suppose ici qu'on ne tape que des lettres, le clavier de la machine à écrire de notre singe dactylographe ne doit pas contenir de chiffres, ni d'autres symboles typographiques).
Cette définition par récurrence conduit naturellement à l'écriture d'une fonction récursive qui va générer de nombreuses redondances empêchant son exécution pour des valeurs trop élevées de n. On peut écrire cette fonction récursive *proba1*() pour évaluer jusqu'à quelle valeur de n on peut aller (sans parler du temps d'exécution qui va s'allonger, c'est la pile d'exécution qui va déborder, arrêtant le processus). Les premières valeurs sont obtenues en des temps raisonnables (inférieurs à 1 seconde tant que n ne dépasse pas 55). Mais ce n'est pas exactement ce que l'on souhaite car une ligne de 80 caractères c'est trop court (les lignes

```
Mode récursif (calcul en 5.13090778219443e-06 secondes) : p(6)= 3.2371282973906776e-09
Mode récursif (calcul en 1.9790644302749945e-05 secondes) : p(10)= 1.618564148695339e-08
Mode récursif (calcul en 0.00015539320711788847 secondes) : p(20)= 4.8556923989305195e-08
Mode récursif (calcul en 0.24591561415866672 secondes) : p(50)= 1.4567076520896105e-07
Mode récursif (calcul en 3.0867042786639995 secondes) : p(60)= 1.7804204351971324e-07
Mode récursif (calcul en 38.555863491465466 secondes) : p(70)= 2.1041332078256563e-07
Mode récursif (calcul en 469.656663106794 secondes) : p(80)= 2.4278459699751825e-07
```

de ce document contiennent une centaines de caractères), la probabilité de l'évènement considéré est trop faible : environ 0,0000002478 soit une chance sur quatre millions.
Dans la nouvelle de B. Rittaud, c'est au moment où le lecteur inconnu du journal commence son exposé sur la façon d'obtenir une expression de p_n en fonction de n, que le rédacteur en chef de ce journal décide de mettre sa lettre au panier. Ce faisant, il nous prive de l'étude de cette suite récurrente d'ordre 6 que le lecteur inconnu avait semble t-il effectuée. Nous ne nous lancerons pas dans ce genre d'étude ici, mais pour évaluer cette probabilité, une formule approchée, un peu moins rigoureuse sans doute que la formule de récurrence (voir les explications données p.185-186), mais beaucoup plus simple à employer est donnée. Cette formule est $p_n = 6 \times \left(1 - \left(1 - \frac{1}{26^6}\right)^{\frac{n}{6}}\right)$. Elle ne justifie même pas un programme en Python tellement elle est aisée à mettre en œuvre. Avec une calculatrice ou un tableur, on trouve des valeurs sensiblement identiques aux résultats calculés avec notre programme récursif jusqu'à n=80, mais on peut aller bien au-delà.



n	6	10	20	50	60	70	80	100	1000	10000	100000	1000000	10000000	100000000	1000000000
p_n	1,94E-08	3,24E-08	6,47E-08	1,62E-07	1,94E-07	2,27E-07	2,59E-07	3,24E-07	3,24E-06	3,24E-05	3,24E-04	3,24E-03	3,23E-02	3,15E-01	2,50E+00

La valeur obtenue pour un milliard de caractères est assez étrange tout de même puisqu'elle dépasse 1 ! Le tracé de la courbe donné par Geogebra nous confirme ce dépassement de 1 à partir de $n \approx 300\,000\,000$ (voir

1 Cette situation d'*un singe dactylographe*, imaginée par Émile Borel au début du XX^e, est mise en scène dans le livre de Benoît Rittaud : *L'assassin des échecs et autres fictions mathématiques* (coll. Plumes de Sciences, ed. Le Pommier, 1992), pp.165-179.

la courbe). Le nombre $1-\frac{1}{26^6}$ est très proche de 1 (il vaut approximativement 0,9999999967628717), élevé à la puissance $\frac{10^9}{6}\approx 166667$, ce nombre vaut environ 0,5830272319, ôté de 1 cela fait à peu près 0,416972768, et en multipliant cela par 6, on trouve bien davantage que 1. Que doit-on conclure ? La probabilité est-elle tellement élevée que l'évènement est plus que certain ? Certainement pas. La formule est-elle fausse ? Pour cette tranche de valeurs, sans doute, ce qui ne l'empêche pas de donner des valeurs acceptables dans un domaine restreint.

En l'absence d'une formule satisfaisante, alors que l'écriture récursive immédiate ne donne rien de bien satisfaisant, une autre écriture récursive, plus efficace car ne s'appelant qu'une seule fois, semble assez tentante. Il suffit d'entrer les six premières valeurs de p_n et la formule de récurrence donnée plus haut donne l'idée de ce mécanisme récursif : au lieu de l'appel *proba1*(n-1)+(1-*proba1*(n-6))/(26**6) qui conduit à l'explosion rapide de l'espace alloué à la pile d'exécution (aux alentours de n=80), nous avons écrit la fonction *proba2*() qui appelle *proba2*(n-1,b,c,d,e,f,f+(1-a)/(26**6)). La formule $p_n = p_{n-1} + \frac{1-p_{n-6}}{26^6}$ peut, en effet, conduire à calculer directement un terme dès lors que l'on conserve les six termes précédents : $p_7 = p_6 + \frac{1-p_0}{26^6}$, dès que l'on connait p_7, on peut se passer de p_0, et on calcule $p_8 = p_7 + \frac{1-p_1}{26^6}$, etc.

```
def proba1(n):
    if n==6 :return 1/(26**6)
    elif n<6 : return 0
    else : return proba1(n-1)+(1-proba1(n-6))/(26**6)
```

```
def proba2(n,a=0,b=0,c=0,d=0,e=0,f=1/(26**6)):
    if n==6 :return f
    elif n<6 : return 0
    else : return proba2(n-1,b,c,d,e,f,f+(1-a)/(26**6))
```

```
Mode récursif (calcul en 0.0012365487755088577 secondes) : p(250)= 7.930961323230796e-07
Mode récursif (calcul en 0.0032185451530879635 secondes) : p(500)= 1.6023772517724784e-06
Mode récursif (calcul en 0.0066525884330280935 secondes) : p(1000)= 3.220937525862455e-06
```

La programmation récursive est plus astucieuse car elle ne crée pas de redondance, cependant elle encombre rapidement l'espace alloué à l'exécution car chaque environnement doit conserver 7 nombres en mémoire (la valeur de n et les six valeurs de p_n). Aux alentours de n=1000, cette fois, on assiste au débordement de la pile. Il faut donc renoncer à obtenir les valeurs recherchées (on s'était fixé un minimum à n=2000).

Une autre approche a été étudiée en classe de seconde. Ce n'est pas le calcul de la probabilité qui est visé mais son estimation grâce à l'intervalle de confiance dans lequel elle se situe. Nous allons procéder à la simulation de m échantillons de taille n et au calcul de la fréquence expérimentale de l'évènement : « au moins une occurrence de la séquence « COGITO » (les six lettres tapées par le singe dactylographe) » a été décelée dans l'échantillon. La fréquence f_e obtenue pour nos m échantillons de taille n permet d'affirmer, avec moins de 5% de chance de se tromper, que la probabilité p_n se situe dans l'intervalle $[f_e - \frac{1}{\sqrt{m}} ; f_e + \frac{1}{\sqrt{m}}]$. La mise en œuvre de cette méthode ne pose pas de problèmes d'écriture (le programme de simulation est simple), mais les durées nécessaires pour générer des échantillons de grande taille sont prohibitives : 8 secondes pour générer un texte de un million de caractères choisis au hasard dans l'alphabet et examen de ce texte pour tenter d'y trouver la séquence « COGITO ». Nous allons cependant le lancer m=400 fois (pour avoir une amplitude de l'intervalle de confiance égal à 0,1) en allant jusqu'à n=2000 caractères (et davantage si c'est réalisable). Pour vérifier les valeurs obtenues par cette méthode statistique, nous pourrons lancer également notre programme *simul*() pour les valeurs n=250, 500 et 1000 pour lesquelles nous avons un calcul exact de probabilité.

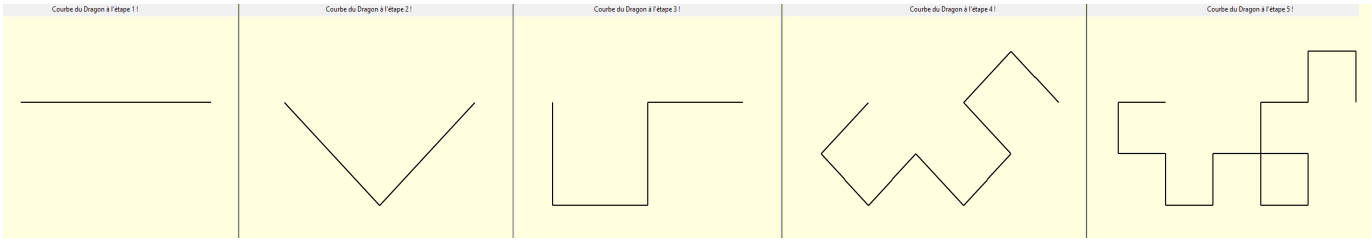
```
from random import *
def simul1(n): #  simulation d'un échantillon de n lettres tirées au hasard,
    alphabet=['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z']
    global s #renvoie l'échantillon et True si la séquence COGITO est présente
    echantillon=''
    presence=False
    for i in range(n):
        echantillon+=choice(alphabet)
    if echantillon.count(s)>0 : presence=True
    return presence

n=int(input('Entrer le nombre de lettres tapées : '))
m=int(input('Entrer le nombre d\'échantillons voulu : '))
s='COGITO'
f,d=simulDuree1(n,m)
print('La séquence {} cherchée est présente dans {}% des {} échantillons de taille {}'.format(s,f/m,m,n))
print('Simulation effectuée en {} secondes.'.format(d))
```

```
from time import clock
def simulDuree1(n,m) : # simulation
    freq=0
    for i in range(m):
        if simul1(n) is True : freq+=1
    t=clock()
    return freq,clock()-t
```

```
La séquence COGITO cherchée est présente dans 0.0% des 10000 échantillons de taille 1000
Simulation effectuée en 83.61153823882324 secondes.

La séquence PI cherchée est présente dans 0.769% des 1000 échantillons de taille 1000
Simulation effectuée en 8.268127313947785 secondes.
```

L'inconvénient majeur de cette méthode est qu'elle ne fonctionne bien que lorsque la fréquence expérimentale n'est ni trop grande (supérieure à 0,8), ni trop petite (inférieure à 0,2). Pour notre situation, la fréquence est bien trop petite, ridiculement petite. Nous ne pourrons donc pas enrichir notre connaissance de cette probabilité qui est infime par cette méthode.
Cette situation mérite t-elle un tel acharnement de méthodes, toutes plus infructueuses les unes que les autres ? Qui peut le dire... Pour le plaisir, ou à titre d'exercice, j'ai envie d'essayer une autre méthode de simulation : générer des échantillons de taille variable contenant au moins une séquence recherchée. Je m'explique, il s'agit de tirer des lettres au hasard jusqu'à obtenir les six lettres de « COGITO » (ce pourrait être n'importe quelle séquence). Les lettres sont oubliées au fur et à mesure des tirages, on n'en garde à chaque fois que cinq pour tester la présence de la séquence lorsqu'on tire la 6ème lettre. Pour pouvoir calculer une moyenne acceptable, nous devons recommencer de nombreuses fois ce dispositif qui n'est pas destiné à évaluer, même indirectement, p_n pour une valeur particulière de n. On arrivera plutôt à un nombre que l'on peut évaluer par cette approche intuitive : le « C » arrive 1 fois sur 26 en moyenne, après le « C » on aura un « O » dans 1 cas sur 26 en moyenne (soit la séquence « CO » dans 1 cas sur 262=676 en moyenne), etc. Cela nous conduit à penser que dans 1 cas sur 26^6, soit dans 1 cas sur 308 915 776, on trouvera le mot « COGITO » écrit. Il faudra donc attendre trois millions de caractères en moyenne pour, enfin, obtenir la séquence tant attendue... Non, finalement je renonce à ce projet qui ne répond plus du tout à la problématique d'origine.

f) Pour finir notre incursion au pays des fonctions récursives, et pour montrer que leur domaine d'application n'est absolument pas limité aux nombres, laissons nous aller à dessiner la *courbe du dragon*. Cette courbe fractale a de nombreuses propriétés comme celle, inattendue, de paver le plan avec des répliques d'elle-même. Pour la construire, on part d'un segment à la 1ère étape, que l'on remplace par un chevron (si le segment initial est la diagonale d'un carré, le chevron est constitué par un des demi-carrés que découpe cette diagonale), et on fait ensuite de même avec les deux nouveaux segments. Les chevrons nouvellement créés sont alternativement tournés de part et d'autre de la courbe dont ils sont issus. Pour mieux comprendre cela, on peut observer le début de la construction sur l'illustration ci-dessous. Si D symbolise un virage à Droite et G un virage à Gauche : à la 2ème étape on a G, à la 3ème on a GGD (nous indiquons avec la couleur verte la séquence précédente, et avec la couleur rose la séquence doublement inversée qui s'ajoute, après un G médian). La séquence GGD devient, à la 4ème étape, GGDGGDD. La 1ère inversion (le début devenant la fin) de GGD conduit à DGG, et il s'ajoute une 2de inversion (les D se changeant en G) qui donne finalement le GDD final. Le processus continue ainsi jusqu'à l'infini, du moins en théorie, mais nous voulons obtenir l'étape n.



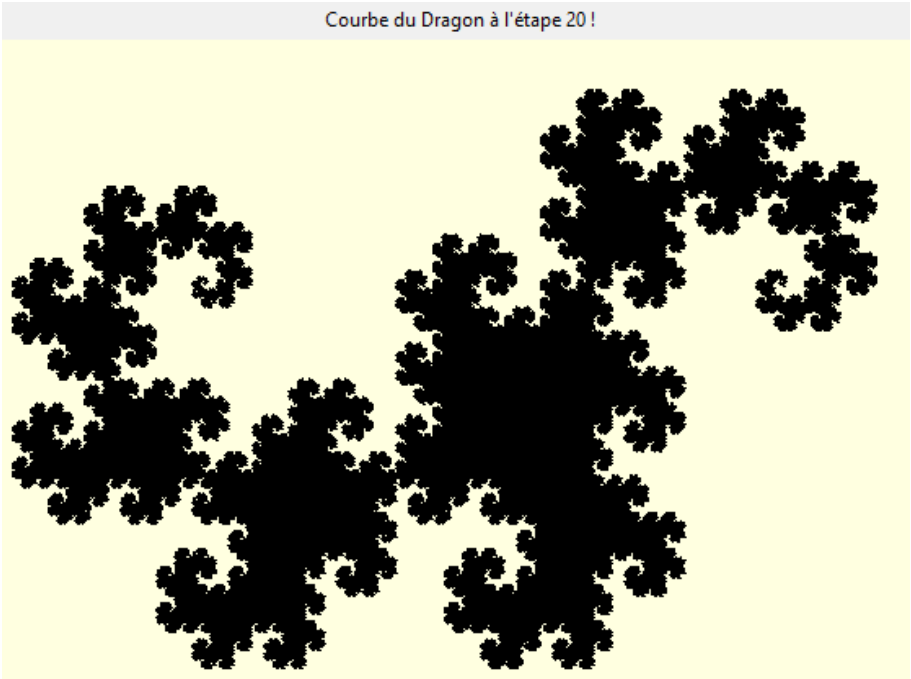
Voici le programme récursif qui a réalisé ces courbes pour les valeurs n=1, 2, 3, 4 et 5. Il est très largement inspiré du programme proposé par Didier Müller sur son site *nymphomath.ch*. La fonction récursive

dragon() prend deux arguments : le 1er est l'indice d'étape et le 2d est un nombre qui oriente les angles de rotation (1 oriente dans un sens et −1 oriente dans l'autre). La longueur du segment initial étant divisée par le coefficient $\sqrt{2}$ à chaque étape, nous calculons dès le départ la longueur du segment final ($\frac{500}{\sqrt{2}^n}$) puisque, de toutes les façons, ce n'est qu'à la fin du processus récursif que la courbe est tracée. Pour que le dragon soit toujours orienté de la même façon (sur ses pattes), nous déterminons l'angle initial selon la valeur de *n*. Cette construction récursive crée des redondances en lançant deux fois la courbe *dragon*(*n*−1,±1) car, si on lance *dragon*(10,1), cela renvoie *dragon*(9,1) et *dragon*(9,−1) qui, respectivement, lancent à leur tour *dragon*(8,1) et *dragon*(8,−1) soit deux *dragon*(8,1) et deux *dragon*(8,−1). Il y aura ensuite quatre *dragon*(7,1) et quatre *dragon*(7,−1), etc.

```
from math import *
from tkinter import *
def dragon(n,s):     #  dessin du dragon récursif
    global x,y,angle #  n:étapes - s:signe
    if n==1 :
        x2,y2=x+long*cos(angle),y+long*sin(angle)
        toile.create_line(x,y,x2,y2,width=2,fill='black')
        x,y=x2,y2
    else :
        dragon(n-1,1)
        angle-=radians(s*90)
        dragon(n-1,-1)
```

```
n=int(input('Entrer le nombre d\'étapes désiré : '))
x,y,long=150,150,500/(sqrt(2))**n
fenetre=Tk()
texte=Label(fenetre,text="Courbe du Dragon à l'étape {} !".format(n),fg='black')
texte.pack ()
toile=Canvas(fenetre,bg='light yellow',height=420,width=610)
toile.pack()
angle=radians((n-1)*45)
dragon(n,1)
fenetre.mainloop()
```

Nous voudrions réécrire une fonction qui traduise la propriété notée dans l'énoncé : à une séquence de segments donnée (l'étape *n*−1), on ajoute la même séquence doublement inversée (après avoir inséré une rotation supplémentaire de même nature que la 1ère rotation). La fonction *inverse*() réalise cela : elle prend en argument une liste d'éléments pris dans l'ensemble {0;1} qu'elle inverse doublement selon le schéma prescrit. Les angles sont juste stockés sous la forme d'un indicateur qui est converti en degré par l'opération (a×2−1)×90 Ainsi 0 donne −90° alors que 1 donne +90°. Cette méthode ne crée pas de redondance (une seule fonction appelée), mais son inconvénient est d'encombrer la mémoire avec une liste d'indicateurs d'angles de plus en plus longue. L'angle qui est ajouté au début de la liste (sous la forme de l'indicateur assez mystérieux (3-*n*)/4 est destiné à donner l'orientation du 1er segment tracé afin que, comme précédemment, le dragon repose sur ses pieds.




```
from math import *
from tkinter import *
def dragon2(k,listeAngle): # dessin du dragon récursif n°2
    global n,x,y,angle     # k:étapes
    if k==1 :
        listeAngle=[(3-n)/4]+listeAngle
        for a in listeAngle:
            angle-=radians((a*2-1)*90)
            x2,y2=x+long*cos(angle),y+long*sin(angle)
            toile.create_line(x,y,x2,y2,width=2,fill='black')
            x,y=x2,y2
    else :
        dragon2(k-1,inverse(listeAngle))
```

```
def inverse(liste01):
    liste10=[(a+1)%2 for a in reversed(liste01)]
    liste01110=liste01+[1]+liste10
    return liste01110

n=int(input('Entrer le nombre d\'étapes désiré : '))
x,y,long=150,150,500/(sqrt(2))**n
fenetre=Tk()
texte=Label(fenetre,text="Courbe du Dragon à l'étape {} !".format(n),fg='black')
texte.pack ()
toile=Canvas(fenetre,bg='light yellow',height=420,width=610)
toile.pack()
angle=0
dragon2(n,[])
fenetre.mainloop()
```

Partie n°5 : Modules et classes

Les fractions permettent d'écrire les nombres rationnels à partir de deux nombres entiers, le numérateur et le dénominateur, le second devant être non nul. Parmi toutes les fractions égales, il existe une unique fraction plus simple que toutes les autres : la *fraction irréductible*. Les nombres rationnels ont une autre propriété : tous ont une écriture décimale qui est périodique à partir d'un certain rang. Pour $\frac{2}{3}=0{,}66666\ldots$, c'est le chiffre 6 qui se répète à partir du rang des dixièmes (−1), précédé par le nombre 0 ; pour $\frac{13}{11}=1{,}181818\ldots$ c'est la suite de chiffres 18 qui se répète à partir du rang des dixièmes (−1), précédée par le nombre 1 ; pour $\frac{15}{14}=1{,}071428571428571\ldots$, c'est la suite de chiffres 714285 qui se répète à partir du rang des centièmes (−2), précédée par le nombre 1,0 (le 0 est important ici).

a) Écrire les fonctions nécessaires à l'obtention de l'écriture irréductible d'un nombre rationnel donné par deux nombres *a* et *b* (le numérateur et le dénominateur de la fraction) ainsi que celles donnant la suite de chiffres se répétant, le rang à partir duquel cette suite apparaît dans la partie décimale et la partie précédent cette suite (appelé partie non périodique du quotient). Toutes ces informations doivent pouvoir être obtenues à partir d'une classe que vous appellerez « Frac » (on ne se sert pas encore de la classe « Fraction » qui existe déjà dans Python).

Pour déterminer la fraction irréductible, il faut diviser le numérateur et le dénominateur par le PGCD de ces deux nombres. Notre fonction *pgcd*() ayant déjà été écrite dans la partie 4a (en mode récursif et itératif), nous l'utiliserons en la copiant dans notre fichier *fractions.py* qui contiendra la classe *Frac*, ou bien en l'important grâce à l'instruction *from utilitaire import* * (en supposant que le fichier *utilitaire.py* soit

utilitaire.py

```
def pgcd(a,b): # forme récursive
    reste=a%b
    if reste==0 : return b
    else : return pgcd(b,reste)
```

fractions.py

```
from utilitaire import *
A=int(input('Numérateur (un entier) : '))
B=int(input('Dénominateur (non nul): '))
a,b=A,B # pour garder la mémoire des nombres entrés
if A*B<0 and B<0 : A,B=-A,B # le signe - au numérateur
elif A*B>0 and B<0 : A,B=-A,-B # simplification des signes
div=pgcd(abs(A),abs(B)) # la fonction pgcd dans utilitaire
print('PGCD({},{})={}'.format(a,b,div))
print('Irréductible({},{})=({},{})'.format(a,b,A//div,B//div))
```

```
Numérateur (un entier) : 15
Dénominateur (non nul): 25
PGCD(15,25)=5
Irréductible(15,25)=(3,5)

Numérateur (un entier) : -1200
Dénominateur (non nul): -850
PGCD(1200,850)=50
Irréductible(1200,850)=(24,17)
```

situé dans le même répertoire que *fractions.py* et qu'il contienne la fonction *pgcd*()).

Pour déterminer la suite qui se répète, on peut essayer de transcrire le processus de la division euclidienne en repérant à quelle moment le reste obtenu fait partie des restes qui ont déjà été obtenus. Pour cela, on écrit une petite boucle de recherche sur cette liste, ou bien on utilise la méthode *count*() de la classe *list* qui fait cela (seule la classe *Fonction* de Python nous est interdite ici). La méthode *index*() de la classe *list* est utile aussi car elle donne l'indice de la 1ère occurrence d'un élément donné dans une liste. Remarque : elle pourrait se substituer ici à la méthode *count*() puisqu'au cas où l'élément est dans la liste (*count*() renvoie un nombre supérieur à 0, égal à 1 dans notre situation) elle renvoie l'indice de cet élément mais au cas où elle n'y est pas elle lève un erreur. Si l'on sait traiter les erreurs, on peut se suffire de cette méthode (mais l'interception d'une erreur est un peu trop compliquée pour qu'on développe ce point ici). Nous utilisons donc ces deux méthodes : *count*() pour repérer la pésence d'un reste dans liste des restes, et *index*() pour nous donner l'indice de reste. Pendant tout le processus, nous écrivons les chiffres du développement décimal de *a/b* dans *quotient* qui est une chaîne de caractères. Quand on connait l'*indice* de reste, on peut séparer la chaîne quotient en deux : la longueur de la suite périodique est égale à *len*(restes)–*indice* et donc

utilitaire.py

```
divDecimale(a,b):
# renvoie la partie non périodique du quotient (str),
#         la suite périodique (str) et le rang (int)
restes=[]
quotient=str(a//b)
reste=a%b
quotient+='.'
if reste==0 : return quotient,'0',1 # cas a/b:entier
while restes.count(reste)==0 :
    restes.append(reste)
    a=reste*10
    quotient+=str(a//b)
    reste=a%b
indice=restes.index(reste)
suite=quotient[indice-len(restes):]
quotient=quotient[:indice-len(restes)]
rang=len(quotient)-quotient.find('.')
return quotient,suite,rang
```

Shell Python

```
Numérateur (un entier) : 157
Dénominateur (non nul): 34
157/34=4.617647058823529411764705882352941...
La suite de 16 chiffres se répète à partir du rang 2 : 1764705882352941
```

fractions.py

```
from utilitaire import *
A=int(input('Numérateur (un entier) : '))
B=int(input('Dénominateur (non nul): '))
a,b,s=A,B,'' # pour garder la mémoire des nombres entrés
if A*B<0 and B<0 : A,B=-A,B # le signe - au numérateur
elif A*B>0 and B<0 : A,B=-A,-B # simplification des signes
if A*B<0 : s='-'
qNP,sP,rg=divDecimale(abs(A),abs(B))#qNP:quotientNonPeriodique sP:suitePeriodique rg:rang
print('{}/{}={}{}{}{}...'.format(a,b,s,qNP,sP,sP))
pluriel="s"
if len(sP)==1 : pluriel=""
print('La suite de {} chiffre{} se répète à partir du rang {} : {}'.format(len(sP),pluriel,rg,sP))
```

le début de la chaîne quotient jusqu'à l'indice –(*len*(restes)–*indice*) est la partie non périodique (on donne l'indice en partant de la fin en utilisant un indice négatif), alors que l'autre partie de la chaîne (celle qui commence à cet indice) est la suite périodique.
Maintenant que l'on a réussi à écrire ces deux fonctions, nous pouvons créer notre classe *Frac* qui prend deux entiers comme argument et qui détermine les caractéristiques de la fraction définie par ces deux entiers (fraction irréductible, partie non périodique, suite périodique et rang de cette suite dans l'écriture décimale). Notre classe pourrait faire bien d'autres choses que cela avec les fractions mais nous ne voulons pas nous substituer à la classe *Fraction* de Python. On peut néanmoins compléter notre classe *Frac* en ajoutant une méthode *compare*(Fract) qui renvoie 0 (respectivement 1 et −1) si la fraction passée en argument est égale (respectivement supérieure et inférieure) à la fraction à laquelle on applique cette

fractions.py

```
from utilitaire import *
class Frac :
# Classe pour fractions.
# Attributs : num,denom (numérateur de la fraction irréductible (int),
#                        dénominateur de la fraction irréductible (int positif)),
# decimales,suite,rang  (l'écriture décimale avec deux suites suivies de ...(str),
#                       la suite des chiffres qui se répètent (str),
#                       le rang à partir duquel commence la suite (int))
# Compare deux fractions avec la méthode compare(Fract)
    def __init__(self,A,B):
        if A*B<0 and B<0 : A,B=-A,B # le signe - au numérateur
        elif A*B>0 and B<0 : A,B=-A,-B # simplification des signes
        div=pgcd(abs(A),abs(B))
        self.num,self.denom=A//div,B//div
        s=''
        if A*B<0 : s='-'
        qNP,sP,rg=divDecimale(abs(self.num),abs(self.denom))
        self.decimales=s+qNP+sP+sP+'...'
        self.suite=sP
        self.rang=rg
    def compare(self,f):
        if self.num*f.num>=0 : #même signe
            if self.num*f.denom-self.denom*f.num<0 : return -1
            elif self.num*f.denom-self.denom*f.num>0 : return +1
            else : return 0
        else :  #signe contraire
            if self.num<0 : return -1
            else : return +1

A=int(input('Numérateur (un entier) : '))
B=int(input('Dénominateur (non nul): '))
frac1=Frac(A,B)
print('Irréductible({},{})=({},{})'.format(A,B,frac1.num,frac1.denom))
print('{}/{}={}'.format(A,B,frac1.decimales))
pluriel="s"
if len(frac1.suite)==1 : pluriel=""
print('La suite de {} chiffre{} se répète à partir du rang {} : {}'.format(len(frac1.suite),pluriel,frac1.rang,frac1.suite))
```

exécution de fractions .py dans le Shell

```
Numérateur (un entier) : 15
Dénominateur (non nul): 11
Irréductible(15,11)=(15,11)
15/11=1.3636...
La suite de 2 chiffres se répète à partir du rang 1 : 36
```

instructions entrées directement dans le Shell

```
>>> A=Frac(15,7)
>>> B=Frac(11,5)
>>> A.compare(B)
-1
>>> A.decimales
'2.142857142857...'
>>> B.decimales
'2.200...'
>>> A.num=16
>>> A.num
16
>>> A.decimales
'2.142857142857...'
```

méthode. Cette méthode additionnelle est juste proposée pour montrer que l'on peut faire autre chose que de déterminer les caractéristiques de l'objet auquel on l'applique.
On a gardé un morceau de programme exécutable à la suite de la classe *Frac* pour continuer à obtenir une sortie semblable à ce qui précède, mais cette partie doit être supprimée si l'on veut conserver le programme *fractions.py* comme module à importer par un autre programme. Notre classe est rudimentaire : elle possède une méthode *__init__*() qui est le « constructeur » de la classe (une fonction obligatoire qui est appelée à chaque fois qu'un objet de type *Frac* est créé). Dans ce constructeur, nous avons lancé le calcul des caractéristiques qui nous intéressent et avons renseignés les cinq attributs que l'on peut interroger/modifier pour un objet de ce type. Dans notre exécution (à droite), on voit que la modification de l'attribut *num* ne modifie pas la valeur de l'attribut *decimales*, ce qui est tout de même assez gênant...

```
from utilitaire import *
class Frac : # Classe pour fractions.

def __init__(self,A,B=1):
    if B<0: A,b=-A,-B  # simplification des signes
    div=pgcd(abs(A),abs(B))
    self.num,self.denom=A//div,B//div
    s=''
    if A*B<0: s='-'
    qNP,sP,rg=divDecimale(abs(self.num),abs(self.denom))
    self.decimales=s+qNP+sP+sP+'...'
    self.suite=sP
    self.rang=rg
def setNum(self,nouveauNum):
    f=Frac(nouveauNum,self.denom)
    self.num=nouveauNum
    self.decimales=f.decimales
    self.suite=f.suite
    self.rang=f.rang
def setDenom(self,nouveauDenom):
    f=Frac(self.num,nouveauDenom)
    self.denom=nouveauDenom
    self.decimales=f.decimales
    self.suite=f.suite
    self.rang=f.rang
```

```
def compare(self,f):
    if self.num*f.num>=0:
        if self.num*f.denom-self.denom*f.num<0: return -1
        elif self.num*f.denom-self.denom*f.num>0: return 1
        else: return 0
    else:
        if self.num<0: return -1
        else: return 1
def __add__(self,f):
    num1=self.num*f.denom+self.denom*f.num
    denom1=self.denom*f.denom
    return Frac(num1,denom1)
def __str__(self):
    pluriel="s"
    if len(self.suite)==1 : pluriel=""
    return "{}/{}={}. La suite de {} chiffre{} ({}) se répète à partir du rang {}".\
           format(self.num,self.denom,self.decimales,len(self.suite),pluriel,self.suite,self.rang)
```

```
>>> a=Frac(115,275)
>>> print(a)
23/55=0.41818.... La suite de 2 chiffres (18) se répète à partir du rang 2
>>> b=Frac(15,35)
>>> print(b)
3/7=0.428571428571.... La suite de 6 chiffres (428571) se répète à partir du rang 1
>>> print(a+b)
326/385=0.8467532467532.... La suite de 6 chiffres (467532) se répète à partir du rang 2
```

Notre deuxième version de cette classe va apporter quelques améliorations pour pallier à ce défaut en la dotant des fonctions *setNum*() et *setDenom*() qui peuvent modifier le numérateur et le dénominateur de la fraction irréductible, entrainant une réactualisation des attributs *decimales*, *suite* et *rang*. Pour la réactualisation, nous avons choisi la solution d'instancier un nouvel objet *Frac* avec les nouvelles valeurs et d'affecter à notre ancien objet, les attributs de ce nouvel objet. Pour cette nouvelle version de notre classe, nous voulons pouvoir entrer un entier sans préciser que le dénominateur est 1 (cela se fait en donnant au dénominateur une valeur par défaut égale à 1). Nous voulons aussi définir une méthode qui permette d'ajouter deux objets *Frac* : cela passe par la fonction spéciale *__add__*(). Ainsi, on peut obtenir un résultat à la somme de objets *Frac*. Par exemple, si *a=Frac*(1,3) et que *b=Frac*(1,6), alors *c=a+b* est un objet *Frac* qui contient les attributs de ½. Sans cette méthode *__add__*(), l'addition *a+b* aurait déclenché une erreur. On peut faire cela pour toutes les opérations en redéfinissant les méthodes spéciales *__sub__*(), *__mul__*(), *__truediv__*(), *__pow__*(), etc. pour la soustraction, la multiplication, la division décimale, l'élévation à la puissance, etc. On peut aussi redéfinir (on dit aussi sucharger) les méthodes de comparaison, par exemple pour que *a<b* retourne *True* ou *False* selon les cas, au lieu du message d'erreur : TypeError: unorderable types: Frac() < Frac(). Une autre méthode intéressante à surcharger est la méthode *str*() qui transforme un objet en chaine de caractère. On écrit pour cela la fonction *__str__*() qui sera notamment appelée si on utilise la fonction *print*() avec un objet *Frac*.

b) Notre classe *Frac* peut s'étoffer encore un peu : nous voulons pouvoir obtenir l'écriture de notre fraction sous sa forme de fraction continue. Par exemple $\frac{105}{8}=13+\frac{1}{8}$ ou $\frac{51}{7}=7+\frac{1}{3+\frac{1}{2}}$. On notera ces fractions avec une notation linéaire [13,8] et [7,3,2]. Ainsi, toutes les fractions peuvent s'écrire sous la forme d'une liste finie [a_0, a_1, a_2, …, a_n], cette notation ayant un intérêt qui dépasse largement ce niveau anecdotique. Comment obtient-on les coefficients de cette notations : avec l'algorithme d'Euclide (encore), les quotients partiels de chaque étape de la recherche du PGCD nous les fournit : 51=7×7+2 (donc a_0=7) puis 7=2×3+1 (donc a_1=3) et enfin 2=1×2+0 (donc a_2=2). Une fois définie cette fonction *fracContinue*(), nous voudrions pouvoir instancier un objet de la classe *Frac* avec une liste d'entiers. Par exemple, nous voudrions pouvoir faire *a=Frac*([7,3,2]) et que dans *a.num* on trouve 51.

fractions.py

```
from utilitaire import *
class Frac : # Classe pour fractions.
    def __init__(self,A,B=1):
        .....
        self.fcontinue=fracContinue(abs(self.num),abs(self.denom))
    .....
    def __str__(self):
        pluriel="s"
        if len(self.suite)==1 : pluriel=""
        return "{}/{}={}. \nLa suite de {} chiffre{} ({}) se répète à \
partir du rang {}. \nDécomposition en fraction continue : {}.".\
format(self.num,self.denom,self.decimales,len(self.suite),pluriel,\
       self.suite,self.rang,self.fcontinue)
```

utilitaire.py

```
def fracContinue(a,b):
    # renvoie la liste des quotients partiels de la
    #         fraction continue égale à a sur b (list)
    lst=[]
    quotient=a//b
    reste=a%b
    lst.append(quotient)
    if reste==0 : return lst # cas a/b:entier
    while reste!=0 :
        a,b=b,reste
        quotient=a//b
        reste=a%b
        lst.append(quotient)
    return lst
```

```
>>> a=Frac(51,7)
>>> print(a)
51/7=7.285714285714....
La suite de 6 chiffres (285714) se répète à partir du rang 1.
Décomposition en fraction continue : [7, 3, 2].
```

```
>>> a=Frac(233,144)
>>> print(a)
233/144=1.618055....
La suite de 1 chiffre (5) se répète à partir du rang 5.
Décomposition en fraction continue : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2].
```

La première partie de ce projet est réalisé en écrivant la nouvelle fonction *fracContinue*() dans le fichier *utilitaire.py*. Ensuite, dans le constructeur de la classe *Frac*, on ajoute une ligne qui appelle cette méthode

pour renseigner le nouvel attribut *fcontinue* de notre fraction, à savoir, la liste des coefficients de la décomposition en fraction continue. Il nous reste juste à modifier la fonction __*str*__() pour tenir compte de ce nouvel attribut et vérifier au passage que la fonction *fracContinue*() fait bien ce qu'elle doit faire. Nos deux exemples, sur l'illustration, montrent que c'est bien le cas puisque [7,3,2] est bien la décomposition de $\frac{51}{7}$. Pour la fraction suivante, qui est égale au rapport entre le 13ème et le 12ème nombre de la suite de Fibonacci, la vérification est moins immédiate. On doit calculer le nombre :
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/2)))))))))) !
On a envie de faire confiance à l'algorithme, non ? Sinon, on peut toujours procéder par étape :
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+2/3)))))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+3/5)))))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+5/8)))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+8/13))))).
A=1+1/(1+1/(1+1/(1+1/(1+13/21)))).
A=1+1/(1+1/(1+1/(1+21/34))).
A=1+1/(1+1/(1+34/55)).
A=1+1/(1+55/89).
A=1+89/144).
A=233/144).
Youpi ! Cela fonctionne bien, on dirait. Il ne nous reste plus qu'à modifier le constructeur pour qu'il accepte de définir un objet *Frac* à partir d'autre chose que deux nombres entiers. Nous devons lui faire accepter une liste pour pouvoir définir une fraction à partir de sa décomposition en fraction continue. Comme le typage des variables n'est pas fort en Python, on peut fournir en premier argument un entier ou une liste, il suffit de mettre le test *A is int* qui renvoie *True* si *A* est entier. Nous réécrivons donc le constructeur de notre classe *Frac* pour qu'il accepte une liste comme argument. Ensuite, bien sûr, il faut écrire la fonction *composition*() dans le module *utilitaire* qui réalise la recomposition du nombre rationnel à partir de sa liste de décomposition sous la forme d'un numérateur et d'un dénominateur. Nous avons choisi d'écrire cette fonction de façon récursive car si f=L[0]+1/f' où L est la liste des coefficients de la décomposition, f' est défini de la même manière f'=L[1]+1/f'' et le dernier terme de la liste est un entier qui permet d'arrêter la récursivité et de remonter dans les calculs. En considérant que f' est une fraction, donnée par son numérateur n et son dénominateur d, nous avons calculé le numérateur et le dénominateur de f en remarquant que f=L[0]+1/(n/d)=(L[0]×n+d)/n.

fractions.py
```
def __init__(self,A,B=1):
    if type(A) is int :   # constructeur a/b
        if B<0: A,b=-A,-B  # simplification des signes
        div=pgcd(abs(A),abs(B))
        self.num,self.denom=A//div,B//div
        s=''
        if A*B<0: s='-'
        qNP,sP,rg=divDecimale(abs(self.num),abs(self.denom))
        self.decimales=s+qNP+sP+sP+'...'
        self.suite=sP
        self.rang=rg
        self.fcontinue=fracContinue(abs(self.num),abs(self.denom))
    else:                 # constructeur [a0,a1,...an]
        a,b=composition(A)
        f=Frac(a,b)
        self.num=f.num
        self.denom=f.denom
        self.decimales=f.decimales
        self.suite=f.suite
        self.rang=f.rang
        self.fcontinue=f.fcontinue
```

utilitaire.py
```
def composition(liste):
    # renvoie le numérateur et le dénominateur d'une fraction
    # définie par sa décomposition en fraction continue
    if len(liste)==1 : return liste[0],1
    a,b=composition(liste[1:])
    return liste[0]*a+b,a
```

Shell Python
```
>>> a=Frac([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2])
>>> b=Frac(233,144)
>>> a.compare(b)
0
>>> print(a)
233/144=1.618055....
La suite de 1 chiffre (5) se répète à partir du rang 5.
Décomposition en fraction continue : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2].
```

```
>>> c=Frac([1, 2, 3, 4, 5])
>>> print(c)
225/157=1.433121019108280254777070063694267515923566878980891719745222929936305732484076....
La suite de 78 chiffres (433121019108280254777070063694267515923566878980891719745222929936305732484076) se répète à partir du rang 1.
Décomposition en fraction continue : [1, 2, 3, 4, 5].
```

c) Notre classe *Frac* peut sans doute accepter encore un autre type de déclaration : nous voulons pouvoir définir une fraction par la partie non-périodique de son développement décimal et la suite de chiffres qui se répète (deux chaînes de caractères). Par exemple *a*=*Frac*("2.1","6") doit être interprété comme le nombre 2,16666... qui est égal à la fraction $\frac{13}{6}$. Pour réaliser cela, on calcule $10^1 a - a = 9a$ (1 car il n'y a qu'un seul chiffre qui se répète) qui vaut autant que 21,6−2,1 (les chiffres d'après sont identiques dans les écritures décimales) soit 19,5. Il ne reste plus qu'à simplifier $\frac{19,5}{9}=\frac{195}{90}=\frac{13\times15}{6\times15}=\frac{13}{6}$. Une fois que la classe *Frac* est au point, la transposer comme une classe héritée de la classe *Fraction* du module *fractions* de Python (*from fractions import Fraction*).
L'algorithme concernant cette transformation a été étudié en seconde (voir TD algorithmique de seconde III-algo3). Les arguments *A* et *B* sont identifiés par leur type (*str*). La fonction *composition2*() de *utilitaire*

va se charger du calcul des entiers qui vont définir la fraction (l'autre fonction *composition*() a été renommée *composition1*()). À partir de *N*=*len*(*B*), le nombre de chiffres qui se répètent, elle enlève *float*(*A*+*B*) à $10^N \times float(A+B+B)$ pour trouver le numérateur *num*. Le dénominateur *denom* est alors 10^N-1. Il ne reste plus qu'à simplifier mais le constructeur de la classe *Frac* sait faire cela.
Pour la fraction qui s'écrit 0,142857142857... on la construit en écrivant *f*=*Frac*('0.','142857').
On a alors *N*=6 et $10^N \times float(A+B) = 10^6 \times 0.142857142857 = 142857.142857$,
d'où *num*= $142857.142857-0.142857=142857.0$ et *denom*= $10^N-1=999999$. Il faut convertir en entiers les deux nombres (le numérateur n'est pas toujours un entier) en les multipliant chacun par la puissance de dix qui convient (cela dépend de la partie décimale de *num*). On trouve alors $\frac{142857}{999999}$ qui est converti en $\frac{1}{7}$ lors de l'instanciation de l'objet *Frac* correspondant.
Voilà notre classe *Frac* complétée. Elle offre des fonctionnalités intéressantes qui ne sont pas prévues dans la classe *Fraction* de Python et d'autres moins intéressantes comme l'addition qui est déjà implémentée (ainsi que toutes les autres opérations habituelles sur les fractions, sans oublier les opérateurs de comparaison ainsi que d'autres fonctionnalités importantes).

fractions.py
```
from utilitaire import *
class Frac : # Classe pour fractions.
  .........
def __init__(self,A,B=1):
    if type(A) is int:   # constructeur a/b
     .........
    elif type(A) is list: # constructeur [a0,a1,...an]
        a,b=composition1(A)
        .........
    else:             # constructeur pnd+sd+sd...
        a,b=composition2(A,B)
        f=Frac(a,b)
        self.num=f.num
        self.denom=f.denom
        self.decimales=f.decimales
        self.suite=f.suite
        self.rang=f.rang
        self.fcontinue=f.fcontinue
 .........
```

utilitaire.py
```
def composition2(pnd,sp):
    # renvoie le numérateur et le dénominateur d'une fraction
    # définie par sa décomposition en partie non-périodique (pnd)
    # et suite périodique (sp) deux chaines de carctères
    N=len(sp)
    num=10**N*float(pnd+sp+sp)-float(pnd+sp)
    denom=10**N-1
    while not num.is_integer():
        num*=10
        denom*=10
    return int(num),int(denom)
```

Shell Python
```
>>> a=Frac('2.1','6')
>>> print(a)
13/6=2.166....
La suite de 1 chiffre (6) se répète à partir du rang 2.
Décomposition en fraction continue : [2, 6].
```

Si nous voulons pouvoir utiliser les fonctionnalités de notre classe *Frac* combinées avec celles de la classe *Fraction* de Python, il faut faire dériver (on dit hériter) les propriétés de *Frac* de la classe mère *Fraction*. Pour commencer, on déclare la classe Frac comme une enfant de Fraction avec, au début, *class Frac*(*Fraction*) : mais ensuite, on ne peut pas continuer à initialiser un objet *Frac* de trois façons différentes, car la superclasse *Fraction* refuse de s'instancier avec une liste par exemple. Nous avons trouvé une sorte de parade qui fait un peu penser à du bricolage, en conservant dans le constructeur la partie concernant un premier argument de type *int* (celui-là ne pose pas de problème). Nous avons ensuite fabriqué une méthode de la classe *Frac* qui prend en argument une liste (*list*), deux chaines (*str*) ou même une fraction (*Fraction*) et qui renvoie un objet de type *Frac* selon les arguments envoyés. L'utilisation de cette méthode est un peu lourde, par exemple en faisant *a*=*Frac*().*setFrac*([1,2,5]) *a* est un objet *Frac*. On peut s'en convaincre en tapant *print*(*a*) qui renvoie, non pas seulement 16/11 (comme l'instruction *print*() de la classe *Fraction* le prévoit), mais les trois lignes donnant les caractéristiques calculées pour un objet *Frac* : 16/11=1.4545.... La suite de 2 chiffres (45) se répète à partir du rang 1. etc.

fractions.py
```
from utilitaire import *
from fractions import Fraction
class Frac(Fraction) : # Classe pour fractions.
def __init__(self,A=0,B=1):# constructeur a/b
    if B<0: A,b=-A,-B  # simplification des signes
    div=pgcd(abs(A),abs(B))
    self.num,self.denom=A//div,B//div
    s=''
    if A*B<0: s='-'
    qNP,sP,rg=divDecimale(abs(self.num),abs(self.denom))
    self.decimales=s+qNP+sP+sP+'...'
    self.suite=sP
    self.rang=rg
    self.fcontinue=fracContinue(abs(self.num),abs(self.denom))
    Fraction.__init__(self,self.num,self.denom)
def setFrac(self,A,B='0'):
    if type(A) is list:  # constructeur [a0,a1,...an]
        a,b=composition1(A)
        return Frac(a,b)
    elif type(A) is str: # constructeur pnd+sd+sd...
        a,b=composition2(A,B)
        return Frac(a,b)
    else:                # constructeur Fraction
        a,b=A.numerator,A.denominator
        return Frac(a,b)
 .........
```

Shell Python
```
>>> a=Frac().setFrac([1,2,5])
>>> print(a)
16/11=1.4545....
La suite de 2 chiffres (45) se répète à partir du rang 1.
Décomposition en fraction continue : [1, 2, 5].
>>> b=Frac().setFrac('2.15','458')
>>> print(b)
215243/99900=2.15458458....
La suite de 3 chiffres (458) se répète à partir du rang 3.
Décomposition en fraction continue : [2, 6, 2, 7, 1, 1, 4, 2, 1, 10, 3].
>>> print(a-b)
-769273/1098900
>>> c=Frac().setFrac(a-b)
>>> print(c)
-769273/1098900=-0.70003913003913....
La suite de 6 chiffres (003913) se répète à partir du rang 3.
Décomposition en fraction continue : [0, 1, 2, 2, 1, 254, 1, 6, 20, 3].
>>> a<c
False
>>> a.compare(c)
1
>>> listeFraction=[a,b,c]
>>> for f in listeFraction :print(f.decimales)
1.4545...
2.15458458...
-0.70003913003913...
```

Cette nouvelle classe *Frac*, héritant des caractéristiques de la classe mère *Fraction*, peut se permettre

maintenant des choses qu'elle ne savait pas faire. Plus besoin de redéfinir l'addition, la soustraction, etc. les comparaisons, etc. tout cela, les objets de la classe *Frac* peuvent le faire. Il y a cependant un bémol : si *a* et *b* sont des objets *Frac*, *a-b* n'en est pas un directement (il faudrait redéfinir la fonction *__sub__*() pour cela) mais peut le devenir en tapant *c=Frac*().*setFrac*(*a-b*) car nous avons prévu dans notre méthode *setFrac*() qu'une fraction soit transformée en objet *Frac*. La comparaison de deux objets *Frac* ne pose cependant pas de problème (pas besoin de redéfinir les méthodes qui permettent normalement cela), ni le parcours d'une liste d'objets de type *Frac*.

d) Pour compléter notre étude des classes Python et enrichir le domaine de nos investigations (il n'y a pas que les nombres!), intéressons nous à un jeu. Il est assez simple de résoudre un sudoku quand on envisage toutes les possibilités, les unes après les autres. L'algorithme de *backtracking* réalise cela : on essaie méthodiquement le remplissage de la grille (sans réfléchir), chiffre après chiffre, et si cela conduit à une impasse, on revient au point précédent où l'on augmente le chiffre, jusqu'à ce qu'on ai essayé toutes les possibilités, dans ce cas on revient encore plus en arrière, etc. En principe une bonne grille de sudoku ne doit avoir qu'une seule solution, nous la cherchons. L'objectif fixé ici : la grille initiale (une liste de 81 éléments donnée par un fichier) devient une instance de la classe *Grille* qui contient les méthodes nécessaires à la recherche de la solution et à son affichage.
Le programme principal effectue la lecture du fichier *sudoku.txt* qui contient la grille initiale sous une forme lisible (pour faciliter sa mise au point) qui doit être transformée en une liste de 81 entiers, cette liste étant l'argument qui permet d'instancier l'objet *probleme* de la classe *Grille*.

classe Grille

```
from tkinter import *
class Grille : # Classe pour Sudoku.
    global grille
    def __init__(self,g):
        self.nChiffres=81-g.count(0)
        self.grille=g
    def meme(self,i,j):
        return (i-j)%9==0 or i//9==j//9 \
               or 3*(i//27)+(i%9)//3==3*(j//27)+(j%9)//3
    def resoudre(self,g):
        if g.count(0)==0: return g
        i=g.index(0)
        impossible=[g[j] for j in range(81) if self.meme(i,j)]
        for n in range(1,10):
            if n not in impossible:
                avance=self.resoudre(g[:i]+[n]+g[i+1:])
                if avance is not None: return avance
    def affiche(self):
        for i in range(1,9):
            cadre.create_line(10,10+i*40,370,10+i*40,width=2,fill='black')
            cadre.create_line(10+i*40,10,10+i*40,370,width=2,fill='black')
        for i in range(4):
            cadre.create_line(8,10+i*120,372,10+i*120,width=4,fill='black')
            cadre.create_line(10+i*120,10,10+i*120,370,width=4,fill='black')
        for i in range(81):
            couleur="blue"
            if self.grille[i]!=grille[i]: couleur="red"
            if self.grille[i]!=0: #affiche les chiffres dans la grille
                cadre.create_text(30+i%9*40,30+i//9*40,text=str(self.grille[i]),\
                                  font="Arial 16",fill=couleur)
```

programme principal

```
from time import clock
from os import getcwd,chdir
chdir(getcwd())
fichierSudoku=open('sudoku2.txt','r')
grille=[]
for ligne in fichierSudoku.readlines() :
    if ligne[0]==' ': break
    grille+=ligne.split(' ')
    grille[-1]=grille[-1][:1]#pour enlever le \n de fin de ligne
grille=[int(a) for a in grille]
probleme=Grille(grille)
fen=Tk()
cadre=Canvas(fen,width=376,height=376,bg='light yellow')
cadre.pack()
print('Il y a {} chiffres dans la grille. \nIl faut en trouver encore {}.'.\
      format(probleme.nChiffres,81-probleme.nChiffres))
t=clock()
solution=Grille(probleme.resoudre(probleme.grille))#lancement
t-=clock()
solution.affiche()       #affiche la grille solution trouvée
print('Recherche achevée en {} secondes.'.format(-t))
fen.mainloop()
```

sudoku2.txt

```
8 0 0 0 0 0 0 0 0
0 0 3 6 0 0 0 0 0
0 7 0 0 9 0 2 0 0
0 5 0 0 0 7 0 0 0
0 0 0 0 4 5 7 0 0
0 0 0 1 0 0 0 3 0
0 0 1 0 0 0 0 6 8
0 0 8 5 0 0 0 1 0
0 9 0 0 0 0 4 0 0
```

Shell Python

```
>>>
Il y a 21 chiffres dans la grille.
Il faut en trouver encore 60.
Recherche achevée en 20.74227743404768 secondes.
```

tk

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

Sudoku diabolique, 10 étoiles

L'instruction *solution=Grille*(*probleme.resoudre*(*probleme.grille*)) lance la recherche (méthode *resoudre*() de la Grille *probleme* avec la grille initiale comme argument, et nomme *solution* la Grille renvoyée par cette méthode. Ainsi, il ne reste plus qu'à utiliser la méthode *affiche*() de cette classe pour obtenir notre grille solution.
Le fonctionnement de la méthode *resoudre*() est récursif : on place un chiffre *n* dans la 1ère case contenant un 0 (d'abord le chiffre 1, les autres ensuite) et on lance la méthode *resoudre*() avec pour argument, la liste contenant ce nouveau chiffre. Lorsque la méthode *resoudre*() active ne réussit pas à placer un nouveau chiffre, elle ne renvoie rien (None) ce qui conduit la méthode *resoudre*() appelante à essayer de placer le chiffre suivant, jusqu'à épuisement des possibilités (ce qui la conduit à renvoyer *None* et produit une rétrogradation) ou jusqu'au succès... Le cœur de cette méthode *résoudre*() est la liste *impossible* des chiffres impossibles pour la case examinée. Cette liste est constituée par l'examen des 20 cases liées à la case examinée (8 cases liées dans le même bloc, plus 6 dans la même ligne et 6 dans la même colonne). En fait, ce sont les 81 cases de la grille qui sont examinées, et la méthode *meme*() est chargée d'identifier si les deux cases examinées sont liées pour au moins une de ces contraintes :
(*i-j*)%9==0 est *True* si *i* et *j* sont dans la même colonne ; *i*//9==*j*//9 est *True* si *i* et *j* sont dans la même

ligne ; 3*(*i*//27)+(*i*%9)//3==3*(*j*//27)+(*j*%9)//3 est *True* si *i* et *j* sont dans le même bloc. Ce dernier test est sans doute moins facile à comprendre : *i*%9 représente la colonne (de 0 à 8) donc (*i*%9)//3 représente la colonne de blocs (de 0 à 3), *i*//27 représente la ligne de blocs (0, 1 ou 2) donc 3*(*i*//27) représente la ligne de blocs (0, 3 ou 6), il suffit ensuite d'ajouter ligne et colonne du bloc.

e) Vous noterez que l'algorithme employé ici pour résoudre les sudokus n'est pas du tout intelligent : il essaie bêtement toutes les possibilités jusqu'à trouver une solution. S'il n'y a en a pas, il va produire une erreur (rien n'est prévu pour ce cas) et s'il y en a deux ou plus, il ne nous le dira pas (et c'est pourtant disqualificatoire pour un sudoku). Nous voulons améliorer notre programme et compter les solutions (aller, tout aussi bêtement, jusqu'au bout du processus). Pour prolonger cette étude, nous pouvons essayer de générer une grille de sudoku valide (a une solution unique) et dont le nombre de chiffres de la grille initiale n'est pas trop grand (entre 17 qui est la valeur minimum et 25, une valeur raisonnable arbitraire) .
Pour la première question posée (compter les solutions d'une grille), nous devons modifier la condition d'arrêt de la fonction récursive. Le fait d'avoir complété une grille (il n'y a plus de 0) ne doit pas arrêter la description de l'arbre des possibilités. Nous devons au moins aller jusqu'à une 2ème solution, si elle existe, pour pouvoir éliminer une grille qui n'est pas valide. À cette fin, nous allons enregistrer les solutions trouvées dans une liste (une liste contenant des listes, les solutions) et c'est tout ! La fonction appelante ne recevant aucune valeur de retour (None), le processus d'exploration de l'arbre des possibilités continue jusqu'à la fin. Le programme principal a été légèrement modifié, la fonction *cherche*() contient toutes les instructions pour effectuer le travail de recherche des solutions, alors que le reste (à droite) se charge de la lecture de la grille initiale. Dans la classe *Grille*, il y a la fonction *isValide*() qui examine si une grille est valide. Cette fonction apparaît inutile si on part d'une grille valide, mais comme l'intérêt de ce programme est de tester des grilles nouvelles, il vaut mieux pouvoir les identifier dès le départ (sinon, elle seraient reconnues comme insolubles (sans solution), ce qui n'est pas la même chose). Nous n'avons pas remis les fonctions *meme*() et *affiche*() de la classe *Grille* dans l'illustration, pour gagner de la place.

Classe Grille

```
from tkinter import *
class Grille : # Classe pour Sudoku.
global solutions
def __init__(self,g):
    self.nChiffres=81-g.count(0)
    self.grille=g
def isValide(self,g):
    for i in range(81):
        for j in range(i+1,81):
            if self.meme(i,j) and g[i]!=0 and g[i]==g[j]: return False
    return True
.....
def resoudre(self,g,k=0):
    if g.count(0)==0:
        solutions.append(g)
    else:
        i=g.index(0)
        impossible=[g[j] for j in range(81) if self.meme(i,j)]
        for n in range(1,10):
            if n not in impossible:
                avance=self.resoudre(g[:i]+[n]+g[i+1:],k+1)
                if avance is not None:
                    return avance
.....
```

Sorties

tk

1	6	5	4	9	8	7	2	3
9	2	4	3	5	7	1	8	6
8	7	3	2	1	6	4	5	9
4	9	8	7	3	1	2	6	5
3	5	7	6	8	2	9	4	1
2	1	6	5	4	9	3	7	8
7	3	2	1	6	5	8	9	4
5	8	1	9	2	4	6	3	7
6	4	9	8	7	3	5	1	2

Entrées

sudoku7.txt

```
1 0 0 4 0 0 7 0 0
0 2 0 0 5 0 0 8 0
0 0 3 0 0 6 0 0 9
4 0 0 7 0 0 0 0 0
0 5 0 0 8 0 0 0 0
0 0 6 0 0 9 0 0 0
7 0 0 1 0 0 0 0 0
0 8 0 0 2 0 0 0 0
0 0 9 0 0 0 0 0 0
```

Shell Python

```
Il y a 20 chiffres dans la grille, il faut en trouver 61.
Il y a 1518 solutions, voici la première.
Recherche achevée en 53 secondes.
```

Programme principal

```
from time import clock
from os import getcwd,chdir
def cherche():
    global probleme,case,solutions
    if probleme.isValide(probleme.grille) :
        print('Il y a {} chiffres dans la grille, il faut en trouver {}.'.\
              format(probleme.nChiffres,81-probleme.nChiffres))#affiche le nombre de chiffres
        t=clock()
        probleme.resoudre(probleme.grille)#lancement de la recherche
        t-=clock()
        if len(solutions)!=0 :
            solution=Grille(solutions[0])
            if len(solutions)==1:
                print('Une seule solution. Recherche achevée en {} secondes.'.format(-t))
            else:
                print('Il y a {} solutions, voici la première. \nRecherche achevée \
en {} secondes.'.format(len(solutions),-t))
        else :
            solution=probleme
            print('Pas de solution. Recherche achevée en {} secondes.'.format(-t))
        solution.affiche()       #affiche la grille solution trouvée
    else:
        print('Grille invalide.')
        probleme.affiche()
chdir(getcwd())
fen=Tk()
cadre=Canvas(fen,width=376,height=376,bg='light yellow')
cadre.pack()
fichierSudoku=open('sudoku7.txt','r')
grille,solutions,case=[],[],81
for ligne in fichierSudoku.readlines() :
    if ligne[0]==' ': break
    grille+=ligne.split(' ')
    grille[-1]=grille[-1][:1]#pour enlever le \n
grille=[int(a) for a in grille]
probleme=Grille(grille)
cherche()
fen.mainloop()
```

La fonction *resoudre*() est presque identique à la précédente. Ce qui est changé, c'est la réponse au fait qu'une solution soit trouvée : on l'ajoute ici à la liste. Dans la fonction *cherche*(), cette liste des *solutions* est retournée et sa longueur différencie les grilles valides (une solution) de celles qui ont trop de solutions. La grille inventée, proposée dans le fichier *sudoku7.txt* possède ainsi 1518 solutions.

Pour le prolongement suggéré, la difficulté augmente. Notre programme actuel trouve la ou les solutions d'une grille valide, mais ne permet pas de modifier la grille pour en faire une grille ayant une solution unique. Cela sera notre approche pour rechercher une grille valide : choisir aléatoirement, jusqu'à 25 chiffres, en ajoutant un nouveau chiffre lorsque la grille valide en cours est surabondante. La plupart du temps, la grille trouvée sera invalide (dernier chiffre incompatible), insoluble (pas de solution) ou surabondante (trop de solutions). En cas d'invalidité, nous augmentons le dernier chiffre (il y en a forcément un qui rend la grille valide (sans incompatibilité) avec l'instruction *n*=*n*%9+1. En cas d'insolubilité, nous enlevons un chiffre au hasard et en cas de surabondance nous en ajoutons un. On peut raffiner cette méthode ou la transformer en procédure de *backtracking* (exploration systématique), mais l'objectif dépasse les limites que nous nous sommes fixés pour ce document. La génération d'une grille de sudoku valide n'est d'ailleurs pas une fin en soi, car il faut encore se confronter au challenge d'évaluer la difficulté de la grille. On ne peut, en effet, remettre entre les mains d'un joueur, une grille non évaluée. Le nombre de chiffres présents sur la grille n'est pas une bonne mesure de la difficulté. Certaines grilles à *n* chiffres sont faciles et d'autres extrêmement difficiles (comme ce sudoku qui a été résolu en 20 secondes par notre programme sur la 1ère illustration). L'évaluation de la difficulté d'un sudoku passe nécessairement par les techniques intelligentes développées par les joueur.

La 1ère des techniques consiste à compléter la grille lorsqu'il n'y a aucun autre choix pour une case. Cette technique parfois suffit, à elle seule, pour compléter la grille entière. Le problème est alors classé « très facile ». La plupart du temps, il est nécessaire de faire la liste des choix possibles pour toutes les cases de la grille, et l'on peut alors discerner les cases où il n'y a qu'un seul choix. Une grille qui serait réalisable avec ces deux seules techniques peut être qualifiée de « facile ». Les niveaux de difficulté augmentent encore lorsque ces méthodes élémentaires ne suffisent plus, et l'on peut discerner, dans le vaste ensemble des sudokus valides, des problèmes de difficulté très élevée, qui résistent aux différentes techniques employées. Les sudokus présentés ci-dessus ont tous les deux 23 chiffres « révélés » et, pourtant, ils sont de difficultés très différentes : celui de gauche ne nécessitant que les deux techniques évoquées plus haut (unicité des chiffres « candidats » pour une case ; la méthode *evalue*() de la classe *Grille* ci-dessous nous montre une implémentation de ces deux techniques de base) est facile alors que celui de droite, inventé en 2006 par le finlandais Arto Inkala et surnommé « Al Escargot », a été réputé le « sudoku le plus difficile au monde » jusqu'à ce qu'on lui conteste cette suprématie. Cette discussion d'expert ne nous concerne pas, mais il va sans dire que pour mesurer la difficulté du sudoku le plus difficile, il faut connaître et avoir programmé toutes les techniques logiques applicables, ce qui paraît irréalisable.

		2		3				8
						6	5	
4		9			1			
7					5		2	
			9					
				6			8	4
8			2			3		
2	5				8	9	7	

23 chiffres - facile

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

23 chiffres - le plus difficile(?)

Telle qu'elle est présentée, cette méthode *evalue*() teste d'abord les techniques *verifLigne*(), *verifColonne*(), et *verifBloc*(), qui, si elles suffisent à résoudre la grille (*if self.grille.count*(0)!=0), attribuent la difficulté 0 au sudoku. Si le sudoku résiste à ce traitement, on lui applique la méthode *verifCase*() qui examine tous les chiffres « candidats » possibles pour une case et remplit la case qui n'a qu'un seul candidat. Pour l'application de ces deux techniques, le programme a besoin d'une 2ème liste, en plus de celle des chiffres, qui contient les différentes possibilités pour chaque case. Cette liste, nommée *possible* dans ce programme, contient les listes des 9 valeurs booléennes (*True* ou *False*) correspondant à la possibilité d'un chiffre dans une case. Par exemple, pour le sudoku facile à 23 chiffres ci-dessus, cette liste commence par :

[[*True, False, False, False, True, True, False, False, False*], [*True, False, False, False, False, True, True, False, False*], etc.]

Le 1er *True* signifie que le chiffre 1 peut se mettre dans la case 0 (en haut à gauche). Dans cette case, on ne peut mettre que les chiffres 1 et 5, alors que dans la case 1 suivante, on peut mettre le 1 ou le 7... La liste *possible* est initialisée au départ à *True* pour tous les chiffres dans toutes les cases vides où ils peuvent se loger et à *False* ailleurs, puis, elle est mise à jour après chaque placement d'un chiffre par la méthode *placeElement*().

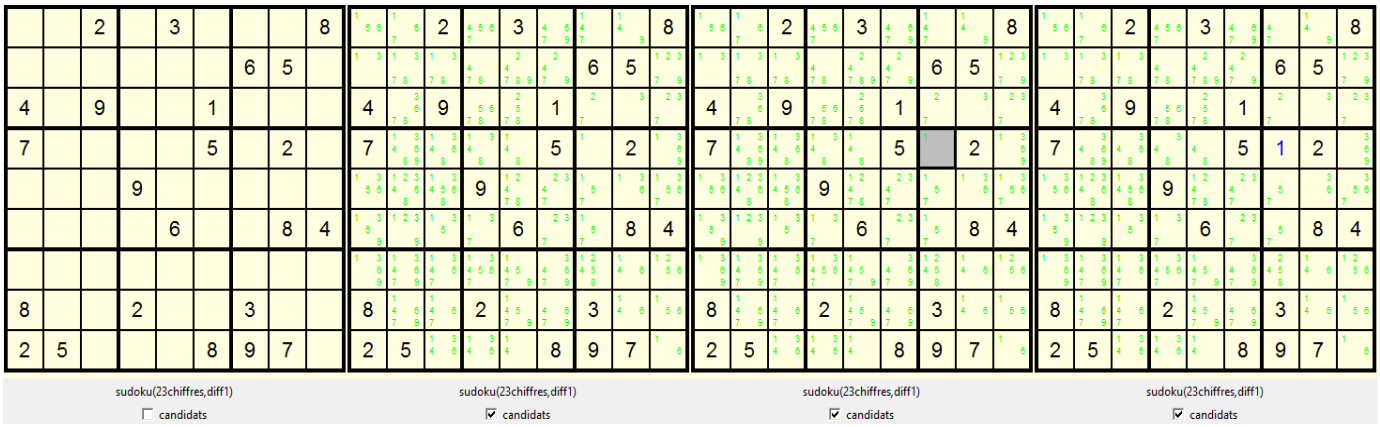
```
global solutions,grille,fichierGrille
def __init__(self,g):
    self.nChiffres=0
    self.grille=g
    self.possible=[]#contient la liste des possibilités pour une case
    for i in range(81):
        if self.grille[i]==0:self.possible.append(9*[True])
        else:self.possible.append(9*[False])
    for i in range(81):
        if self.grille[i]!=0:
            self.placeElement(i,self.grille[i])
def evalue(self):
    diff,possible=0,True
    while possible:
        possible=False
        if diff==1:#2ème méthode à utiliser
            for i in range(81):# possibilité unique dans 1 case
                if self.verifCase(i):possible=True
        for i in range(9):#1ère méthode utilisée
            for n in range(1,10):#un seul choix possible sur une zone
                if self.verifLigne(i,n):possible=True
                if self.verifColonne(i,n):possible=True
                if self.verifBloc(i,n):possible=True
        if diff==0 and not possible and self.grille.count(0)!=0:
            diff=1
            possible=True
    if self.grille.count(0)!=0:diff+=1
    return diff
def verifCase(self,c):
    compteur,candidat=0,0
    for n in range(1,10):
        if self.possible[c][n-1]:
            compteur+=1
            candidat=n
    if compteur==1:
        self.placeElement(c,candidat)
        return True
    return False
def verifLigne(self,lig,n):
    compteur,rang=0,0
    for i in range(9):
        if self.possible[lig*9+i][n-1]:
            compteur+=1
            rang=i
    if compteur==1:
        self.placeElement(lig*9+rang,n)
        return True
    return False
def verifColonne(self,col,n):
    compteur,rang=0,0
    for i in range(9):
        if self.possible[col+i*9][n-1]:
            compteur+=1
            rang=i
    if compteur==1:
        self.placeElement(col+rang*9,n)
        return True
    return False
def verifBloc(self,blo,n):
    compteur,rang=0,0
    for i in range(9):
        if self.possible[(blo//3)*27+(blo%3)*3+(i//3)*9+i%3][n-1]:
            compteur+=1
            rang=i
    if compteur==1:
        self.placeElement((blo//3)*27+(blo%3)*3+(rang//3)*9+rang%3,n)
        return True
    return False
def placeElement(self,c,n):
    self.grille[c]=n
    self.nChiffres+=1
    for i in range(81):#mise à jour des possibles dans
        if i%9==c%9: self.possible[i][n-1]=False # la même colonne
        if i//9==c//9: self.possible[i][n-1]=False # la même ligne
        if 3*(i//27)+(i%9)//3==3*(c//27)+(c%9)//3: self.possible[i][n-1]=False # le même bloc
    for i in range(1,10):self.possible[c][i-1]=False #mise à jour dans la case
```

Partie n°6 : Interactivité

Nous avons déjà rencontré quelques situations interactives – où l'utilisateur peut converser avec le programme – quand nous avons utilisé le module *tkinter* et découvert quelques uns de ses nombreux dispositifs graphiques. Mais l'interactivité est également présente dans un simple programme qui demanderait à l'opérateur d'entrer son nom jusqu'à ce que le nom entré ne contienne aucun chiffre. Cela pour dire que la notion d'interactivité recouvre des fonctionnalités très diverses, pas forcément compliquées, qui permettent de moduler les actions du programme.

a) Pour continuer avec les sudokus, nous aimerions disposer d'un programme qui affiche une grille initiale et offre la possibilité de la compléter avec des chiffres entrés au clavier. La fenêtre d'affichage doit être rendue sensible (la méthode *bind*() de la classe *Tk* réalise cela) et réagir aux entrées du clavier en lançant un évènement (*KeyPress*) qui permet à la commande *ajoute*() de se servir du chiffre entré. Le programme doit reconnaître une grille complète et valide en affichant le message « Bravo ! La solution a été trouvée en … secondes »). On peut prévoir aussi une case à cocher *Checkbutton*() qui, lorsqu'elle est cochée, déclenche l'affichage des chiffres « candidats » dans les cases vides.

Pour commencer, nous reprenons notre classe *Grille* avec ses deux attributs *grille* et *possible* ainsi que la procédure de lecture d'un fichier *sudoku.txt*. La classe *Grille* doit s'enrichir de quelques méthodes qui permettront de placer les nouveaux chiffres dans les cases. Placer un chiffre dans une case se fera par *setCase*() qui appelle *placeElement*() si le chiffre appartient à {1,2,3,4,5,6,7,8,9} et *placeZero*() si le chiffre est 0 – ce qui équivaut à un effacement du chiffre de la case. La méthode *affiche*() doit être enrichie pour tenir compte des options d'affichage : lorsqu'on clique dans une case, on peut montrer que la case a été sélectionnée en la colorant en gris ; lorsque le chiffre entré est invalide, on peut l'afficher en rouge alors qu'il sera affiché en bleu s'il est valide et en noir s'il fait partie de la grille initiale ; lorsque l'option « candidats » a été cochée, on doit afficher les chiffres candidats en petit (selon une grille 3×3).



L'interactivité principale est l'entrée des chiffres dans la grille. Elle est déclenchée par l'instruction fen.*bind*("<KeyPress>", *modifie*) où *ajoute*() est la fonction qui est lancée lorsqu'on presse une touche du clavier. La fonction *ajoute*() prend l'évènement *event* en argument, et l'utilise par l'instruction touche=*event.keysym* qui a pour effet d'affecter à la variable *touche* le caractère entré (sous la forme d'une chaîne, par exemple '3' si on presse la touche 3 et 's' si on presse la touche s). On pourrait effectuer un contrôle sur le caractère entré (pour savoir s'il s'agit bien d'un chiffre) mais nous considérons que l'utilisateur sait ce qu'il fait... Avant d'entrer le chiffre, l'utilisateur doit sélectionner la case de la grille qu'il veut compléter : cela se fait par l'ajout de cette instruction qui sensibilise le programme aux clics de souris cadre.*bind*("<*Button-1*>", *surligne*) où *surligne*() est la fonction qui est lancée lorsqu'on clique avec le bouton gauche de la souris (*Button-1*). La fonction *surligne*() prend l'évènement *event* en argument, et en utilise les attributs de coordonnées par les instructions xcurseur=*event.x* et ycurseur=*event.y*. Les coordonnées servent à déterminer quelle case de la grille a été sélectionnée (pour éviter les erreurs d'exécution si on clique en dehors de la grille, nous avons disposée les instructions dans un bloc *try*) et à lancer la méthode *surligneCase*() de la classe *Grille* (si la case était bien vide dans la grille initiale) puis le réaffichage de la grille. L'interaction optionnelle d'affichage des chiffres candidats, assurée par le cochement de la case, est réalisée par les instructions choix=*IntVar*() et aide=*Checkbutton*(*fen*, *text="candidats", variable=choix,command=affiche*) où *affiche*() est la fonction qui est lancée lorsqu'on coche ou décoche la case *aide*. Cette fonction se contente d'effacer la grille et de lancer la méthode *affiche*() de la classe *Grille*, où on a pris soin d'utiliser la valeur du paramètre *choix* qui vaut 0 quand la

case est désélectionnée et 1 lorsqu'elle est sélectionnée. La grille est donc ré-affichée avec la bonne valeur du paramètre. Voilà pour l'essentiel ! Il suffit, pour maintenir l'affichage des candidats à jour, de relancer cet affichage à chaque modification de la grille.

```
from tkinter import *
class Grille : # Classe pour Sudoku. basée sur sudoku3
    global solutions,choix, label
    def __init__(self,g):
        self.nChiffres=0
        self.grille=g
        self.surligned=81
        self.invalide=81
        self.possible=[]#contient la liste des possibilités pour une case
        for i in range(81):
            if self.grille[i]==0:self.possible.append(9*[True])
            else:self.possible.append(9*[False])
        for i in range(81):
            if self.grille[i]!=0:
                self.placeElement(i,self.grille[i])#print(self.possible)
    def surligneCase(self,c):
        self.surligned=c
    def isValide(self,g):
        for i in range(81):
            for j in range(i+1,81):
                if self.meme(i,j) and g[i]!=0 and g[i]==g[j]: return False
        return True
    def setInvalide(self,n):
        self.invalide=n
    def setCase(self,i,n):
        if n==0: self.placeZero(i)
        else: self.placeElement(i,n)
        self.surligned=81
    def meme(self,i,j):
        return (i-j)%9==0 or i//9==j//9 \
               or 3*(i//27)+(i%9)//3==3*(j//27)+(j%9)//3
    def placeZero(self,c):
        self.possible=[]
        self.grille[c]=0
        self.nChiffres=0
        for i in range(81):
            if self.grille[i]==0:self.possible.append(9*[True])
            else:self.possible.append(9*[False])
        for i in range(81):
            if self.grille[i]!=0:
                self.placeElement(i,self.grille[i])
    def placeElement(self,c,n):
        self.grille[c]=n
        self.nChiffres+=1
        for i in range(81):#mise à jour des possibles dans
            if i%9==c%9: self.possible[i][n-1]=False # la même colonne
            if i//9==c//9: self.possible[i][n-1]=False # la même ligne
            if 3*(i//27)+(i%9)//3==3*(c//27)+(c%9)//3: self.possible[i][n-1]=False # le même bloc
        for i in range(1,10):self.possible[c][i-1]=False #mise à jour dans la case
    def affiche(self):
        for i in range(1,9): #affiche les grosses lignes de la grille
            cadre.create_line(10,10+i*40,370,10+i*40,width=2,fill='black')
            cadre.create_line(10+i*40,10,10+i*40,370,width=2,fill='black')
        for i in range(4):   #affiche les grosses lignes de la grille
            cadre.create_line(8,10+i*120,372,10+i*120,width=4,fill='black')
            cadre.create_line(10+i*120,10,10+i*120,370,width=4,fill='black')
        if self.surligned!=81 :
            c=self.surligned
            cadre.create_rectangle(11+c%9*40,11+c//9*40,48+c%9*40,48+c//9*40,width=1,fill='grey')
        for i in range(81):
            couleur="black"
            if self.grille[i]!=grille[i]: couleur="blue"
            if self.invalide==i: couleur="red"
            if self.grille[i]!=0: #affiche les chiffres dans la grille
                cadre.create_text(30+i%9*40,30+i//9*40,text=str(self.grille[i]),\
                                  font="Arial 16",fill=couleur)
        if choix.get()==1:
            for i in range(81):
                for j in range(9):
                    if self.possible[i][j]:
                        cadre.create_text(17+i%9*40+j%3*12,19+i//9*40+j//3*12,text=str(j+1),font="Arial 7",fill="green")
```

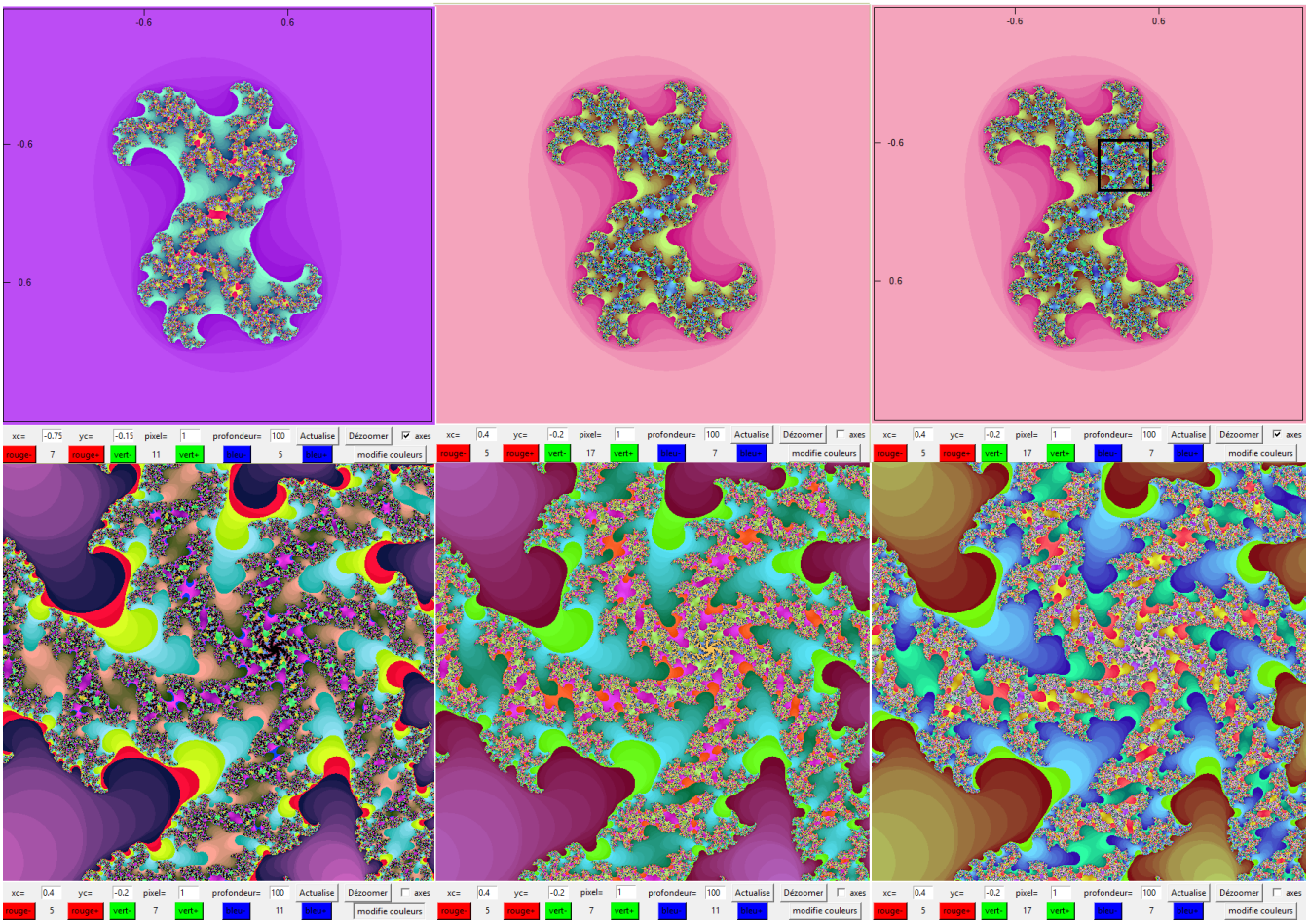
```
from time import clock
from os import getcwd,chdir
def affiche():
    cadre.delete(ALL)
    probleme.affiche()
def surligne(event):
    global probleme,case
    xcurseur=event.x
    ycurseur=event.y
    case=(xcurseur-10)//40+9*((ycurseur-10)//4
    if case in range(81) and grille[case]==0:
        probleme.surligneCase(case)
        cadre.delete(ALL)
        probleme.affiche()
def ajoute(event):
    global probleme,case
    t=event.keysym
    try:
        touche=int(t)
        if touche in range(10) and grille[case]==0:
            probleme.setCase(case,touche)
            cadre.delete(ALL)
            controle(case)
            probleme.affiche()
    except:None
def controle(c=81):
    global probleme,t,label
    if probleme.grille.count(0)==0:
        if probleme.isValide(probleme.grille) :
            t-=clock()
            print('Bravo! La solution a été trouvée en {} secondes.'.format(-t))
            label.configure(text="Bravo! Problème résolu en {} s!".format(int(10
        else:print('Il y a un petit problème, la solution trouvée est incorecte!
    else:
        if probleme.isValide(probleme.grille) :
            print('Il reste {} chiffres à trouver.'.format(81-probleme.nChiffres
            probleme.setInvalide(81)
        else:
            print('La grille n\'est pas valide.')
            probleme.setInvalide(c)


chdir(getcwd())
fen=Tk()
nomSudoku='sudoku(23chiffres,diff1)'
cadre=Canvas(fen,width=376,height=376,bg='light yellow')
cadre.pack()
label=Label(fen,text=nomSudoku)
label.pack()
choix=IntVar()
aide=Checkbutton(fen,text="candidats",variable=choix,command=affiche)
aide.pack()
grille,solutions,case=[],[],81
fichierSudoku=open(nomSudoku+'.txt','r')
for ligne in fichierSudoku.readlines() :
    if ligne[0]==' ': break
    grille+=ligne.split(' ')
    grille[-1]=grille[-1][:1]#pour enlever le \n de fin de ligne
fichierSudoku.close()
grille=[int(a) for a in grille]
probleme=Grille([int(a) for a in grille])
controle()
probleme.affiche()
t=clock()
cadre.bind("<Button-1>",surligne)
fen.bind("<KeyPress>",ajoute)
fen.mainloop()
```

Ce programme fonctionne à peu près comme nous le souhaitions. Ce n'est sans doute pas le mieux que l'on puisse faire. Il faudrait songer à implémenter d'autres fonctionnalités interactives telles que : proposer de résoudre d'autres grilles (en cherchant parmi les grilles présentes dans un dossier), afficher un chronomètre, proposer la solution, ou une option qui corrige les erreurs (si un chiffre entré ne correspond pas à la solution), etc. Ce faisant, nous avons un peu progressé en utilisant une case à cocher et en rendant le programme sensible à deux types d'évènements différents (un clic gauche de la souris et une saisie au clavier).

Un autre dispositif de programmation a été utilisé pour la 1ère fois : le bloc *try* qui a une syntaxe simple constitué des deux lignes consécutives : *try* :< instructions> et *except* :< instructions>. On essaie de faire quelque chose dans le bloc *try*, et si ce quelque chose conduit à une erreur, on n'arrête pas le programme

pour autant, mais on exécute ce qui suit la commande *except*. Ici, la conversion de la touche pressée avec *int*() peut échouer (si la touche est une lettre, ou bien la touche *Enter*, ou *Alt Gr* ou encore une autre touche) et conduire à une erreur. Dans notre cas, on ne fait rien si une erreur arrive, mais dans d'autres situations, on pourrait intercepter le type d'erreur et l'afficher ou lancer d'autres traitements spécifiques de l'erreur. La commande complète serait *except type_de_l_exception as* type : qui conduirait à mettre le type de l'erreur (NameError, TypeError, ZeroDivisionError, ValueError, etc.) dans la variable *type*. En l'absence du bloc *try*, notre programme génèrerait une erreur de type *ValueError* en cas de saisie de la touce *Enter* avec le message suivant : *invalid literal for int() with base 10: 'Return'*.

b) Changeons de sujet et faisons un pas de plus dans le monde merveilleux des fractales ! Les ensembles de Julia sont des images fractales construites selon un procédé assez simple. À chaque pixel P de coordonnées $(x;y)$ nous allons associer une couleur $coul(r,b,v)$ selon la 1ère valeur de l'entier n pour laquelle l'image P_n du point P s'écarte de l'origine $O(0;0)$ du repère d'une différence supérieure ou égale à 2. Pour trouver les coordonnées $(x';y')$ de l'image P_n, à partir de celles $(x;y)$ de P_{n-1} on applique les relations : $x'=x^2-y^2+x_I$ et $y'=2xy+y_I$ où $(x_I\,;y_I)$ sont les coordonnées d'un point I du plan. Lorsque le point P_n ne sort pas (en essayant les valeurs de n jusqu'à une certaine valeur maximum appelée *prof* pour profondeur) du disque de rayon 2 centré sur O, la couleur du point est noire, sinon la couleur est définie par un choix arbitraire, par exemple le dégradé de vert $coul(0,\frac{(prof-n)\times 255}{prof},0)$. En procédant ainsi, on obtient l'image de l'ensemble de Julia associé au point I. Programmer l'affichage d'une telle image pour x et y compris entre -2 et 2, avec la possibilité de modifier les coordonnées de I et la possibilité de zoomer sur une partie de cette image (par exemple en cliquant dans le cadre deux extrémités de la nouvelle fenêtre en diagonale).

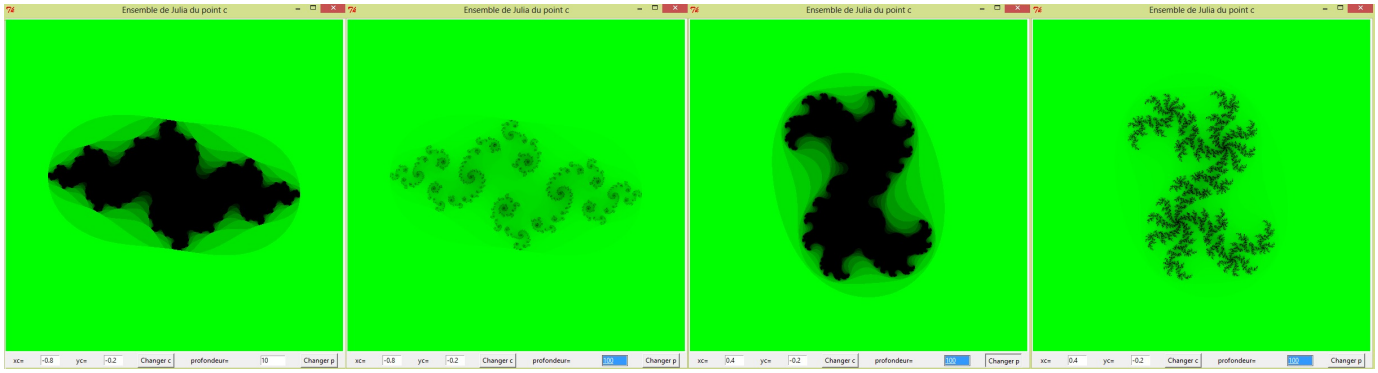


Disons le tout de suite, nous avons délibérément choisi de ne pas parler de nombres complexes pour rester abordable au niveau de la classe de 1ère, mais l'opération sur les coordonnées définie par l'énoncé revient à appliquer au complexe $z=x+iy$, la transformation $z'=z^2+z_I$ avec $z_I=x_I+iy_I$. L'utilisation des complexes n'apporte pas grand chose de plus à cette situation, tant que l'on se satisfait d'une exploration du phénomène, sans rechercher les explications qui sont, elles, liées aux propriétés de ce type de suites complexes qui ont été étudiées par le mathématicien Gaston Julia au début du XXème siècle.
Avant de mettre en place l'interactivité recherchée, on doit déjà construire cette image de l'ensemble de

Julia associé à un point. Il nous faut un *Canvas* (cadre) pour y dessiner des points, c'est-à-dire des rectangles de 1 pixel de côté (*cadre.create_rectangle*(*x*,*y*,*x*+1,*y*+1,*width*=0,*fill*=*coul*)). Tout se joue sur la couleur à donner au pixel. Cette couleur peut être choisie dans un dégradé dès lors que l'on a déterminé la valeur de l'entier *n* (entre 1 et un maximum appelé *profondeur*) pour lequel l'image P_n du point sort du disque de rayon 2. La fonction *calcule*() effectue cette détermination. Il faut, au préalable, avoir déterminé les coordonnées réelles (x_P;y_P) du point *P* auquel elle s'applique. Comme on balaye la fenêtre pixel par pixel, on applique des formules de changement de repère aux coordonnées (*x*;*y*) de *P* dans la fenêtre : $x_P=x\times(xMax-xMin)/taille+xMin$, où *xMax* et *xMin* sont les maximum et minimum de l'abscisse réelle *x* (au départ, on a dit que c'était 2 et –2) et où *taille* est la dimension de la fenêtre (on a pris 600 pixels, mais c'est beaucoup, les temps de calcul seront longs, 400 aurait pu faire l'affaire).

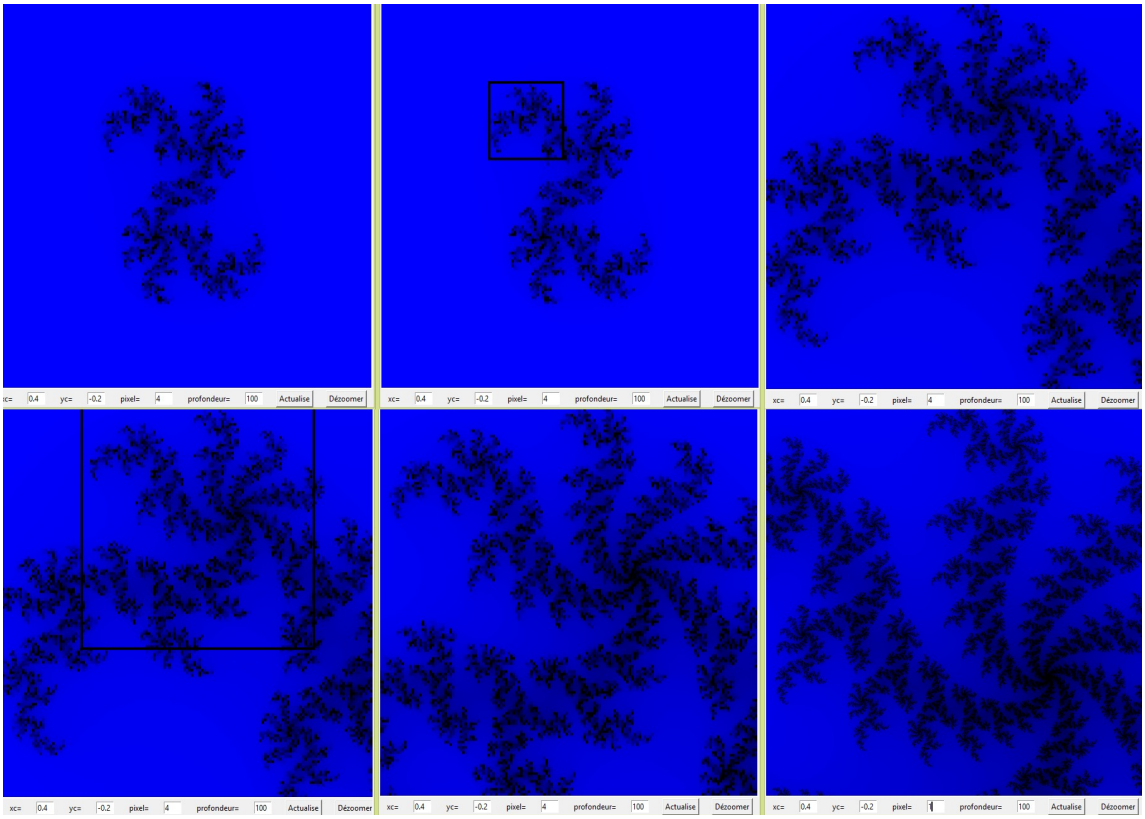
```
from tkinter import *
class Point :
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def suivant(self):
        return Point(self.x**2-self.y**2+iPoint.x,self.x*self.y*2+iPoint.y)
    def distance(self):
        return self.x**2+self.y**2

def modifie():
    xI=float(entree1.get())
    yI=float(entree2.get())
    iPoint=Point(xI,yI)
    trace()
def profondeur():
    global prof
    index=coeffProf.curselection()
    prof=int(coeffProf.get(index))
    trace()
def calcule(p):
    p1=p
    for i in range(prof):
        p1=p1.suivant()
        if p1.distance()>=2: return i
    return -1
def trace():
    for x in range(taille):
        for y in range(taille):
            p=Point(x*(xMax-xMin)/taille+xMin,y*(yMax-yMin)/taille+yMin)
            n=calcule(p)
            if n==-1: coul='black'
            else:
                coul,vert='#00',(prof-n)*255//prof
                if vert>15 :coul+=hex(vert)[2:]
                else:coul+='0'+hex(vert)[2:]
                coul+='00'
            cadre.create_rectangle(x,y,x+1,y+1,width=0,fill=coul)

fen=Tk()
fen.title('Ensemble de Julia du point c')
xI,yI=-0.8,-0.2
taille,prof=600,10
iPoint=Point(xI,yI)
xMin,xMax,yMin,yMax=-2,2,-2,2
cadre=Canvas(fen,width=600,height=600,bg='white')
cadre.grid(row=1,column=1,columnspan=9)
Label(fen,text="xc=").grid(row=2,column=1)
entree1=Entry(fen,width=5)
entree1.insert(END,"-0.8")
entree1.grid(row=2,column=2)
Label(fen,text="yc=").grid(row=2,column=3)
entree2=Entry(fen,width=5)
entree2.insert(END,"-0.2")
entree2.grid(row=2,column=4)
Button(fen,text='Changer c',command=modifie).grid(row=2,column=5)
Label(fen,text="profondeur=").grid(row=2,column=6)
coeffProf=Listbox(fen,height=1,width=7)
for i in [10,20,30,50,75,100,150,200,300,500,1000]:
    coeffProf.insert(END,str(i))
coeffProf.grid(row=2,column=8)
Button(fen,text='Changer p',command=profondeur).grid(row=2,column=9)
trace()
fen.mainloop()
```

Nous avons défini une classe *Point* qui ne contient que deux attributs (*x* et *y*) et deux méthodes (*suivant*() qui renvoie un nouvel objet de la classe *Point* répondant à la définition du point suivant donnée dans l'énoncé et *distance*() qui renvoie la distance du point à l'origine). Pour introduire un peu d'interactivité dans ce programme initial, nous avons disposé deux cadres de texte qui permettent d'entrer de nouvelles valeurs pour les coordonnées du point de référence *I* (nous l'avons renommé *C* pour une commodité d'affichage, le *I* ne se lisant pas bien), un bouton pour relancer le dessin après la modification de *C* (celui-ci déclenche la fonction *modifie*() qui change la variable globale iPoint (le point de référence)) et une *listBox*() pour choisir la profondeur d'investigation dans une liste préétablie (cette dernière option nécessitant l'usage des flèches du clavier pour atteindre les différentes valeurs ne semble pas particulièrement adaptée, nous ne la conserverons pas dans les versions ultérieures).



La disposition des « widgets » dans la fenêtre est assurée ici par la méthode *grid*() qui permet de les disposer selon une grille : ligne 1 nous plaçons le cadre graphique (de la classe *Canvas*) et dans la ligne 2, tous les widgets permettant l'interaction : les cadres de texte (*Entry*) accompagnés de leur étiquette (*Label*), les boutons (*Button*) et la liste déroulante (*Listbox*). Sur cette ligne (*row*=2), les emplacements sont fixés par le numéro de colonne (*column*=i). Lorsqu'un widget s'étale sur plusieurs colonnes, on indique leur nombre (*columnspan*=9 pour le cadre ici). La largeur des cadres de texte peut se paramétrer (*width*=5) ainsi que bien d'autres caractéristiques dont nous ne faisons pas usage, et ces zones peuvent contenir un texte

initial (ici, on y a indiqué la valeur par défaut).
Le résultat est déjà assez satisfaisant mais la palette de couleur utilisée se révèle trop pauvre, juste suffisante pour donner un aperçu de l'ensemble de Julia considéré. La nécessité de disposer d'un zoom se fait aussi sentir, car on aimerait pouvoir agrandir certaines parties de la figure pour l'observer avec plus de détails. Notre deuxième version du programme va ajouter ces éléments pour une meilleure performance et un plus grand confort d'utilisation (on pourrait aussi bien modifier les paramètres d'affichages directement dans le programme mais cela n'est pas très commode). Nous ne travaillerons pas directement l'optimisation de nos calculs (il faudrait en particulier utiliser les nombres complexes), mais pour accélérer l'affichage de la figure nous disposons déjà du paramètre profondeur qui, en limitant les investigations donne une image plus ou moins affinée et, par voie de conséquence, plus ou moins lente à s'afficher. Notre 2ème version ajoute une autre disposition pour accélérer l'affichage (tout en diminuant sa qualité) : la pixellisation peut être augmentée (au lieu de carrés de 1 pixel de côté, on peut dessiner avec des carrés de 2, 3, 4, etc. pixels de côtés. Le paramètre *pixel* a donc pour fonction de réduire le nombre de points. Il va être ajouté aux dispositifs d'interactivité dans la ligne 2 du cadre pour être réglable par l'utilisateur. La profondeur va être réglée par un cadre de texte (suppression de la *Listbox* qui n'apporte rien). Le zoomage va se réaliser en deux temps, comme il est dit dans le texte : on clique tout d'abord sur deux points de l'image (donc le *Canvas* doit être rendu sensible avec la méthode *bind*), et un cadre carré noir indique le cadre sélectionné pour être agrandi (cette opération doit pouvoir être recommencée jusqu'à satisfaction), on lance le recalcul du dessin en validant la sélection (un bouton *actualise* lance, d'un coup, toutes les opérations de modification). L'option de pixellisation est réglée par défaut sur 4 pour accélérer le choix du dessin que l'on veut obtenir.



Cette deuxième version ne règle pas le défaut de la palette de couleur. On aimerait aussi avoir des indications sur les valeurs des coordonnées réelles. Ces indications seront données par une option d'affichage des axes gradués dans une version ultérieure. Mais, pour l'essentiel, cette version n°2 nous donne satisfaction, car on peut observer à loisir (avec des temps d'attente importants lorsque la pixelisation est minimale et la profondeur maximale) ces magnifiques ensembles fractals. Le fonctionnement du zoom nécessite d'utiliser les évènements de souris *<ButtonPress-1>* et *<ButtonRelease-1>* qui lance chacun une fonction (*coin1*() quand on clique sur le bouton gauche et *coin2*() quand on relâche ce bouton). Les variables *x1, y1, x2* et *y2* enregistrent les coordonnées des clics (dans le système de repérage lié à la fenêtre) et l'on s'arrange pour que *x1* et *y1* soient les coordonnées du point supérieur gauche (on utilise pour cela les fonction *min* et *max* de Python) car l'utilisateur peut décrire la diagonale dans 4 sens différents. On recalcule aussi, dans la fonction *coin2*(), la valeur de *x2* et *y2* pour que la fenêtre soit carrée (et non pas rectangulaire). La mise à jour des nouvelles valeurs de *xMin, xMax, yMin* et *yMax* se fait dans

la fonction *modifie*() qui est lancée lorsque l'on souhaite modifier quelque chose dans l'affichage.

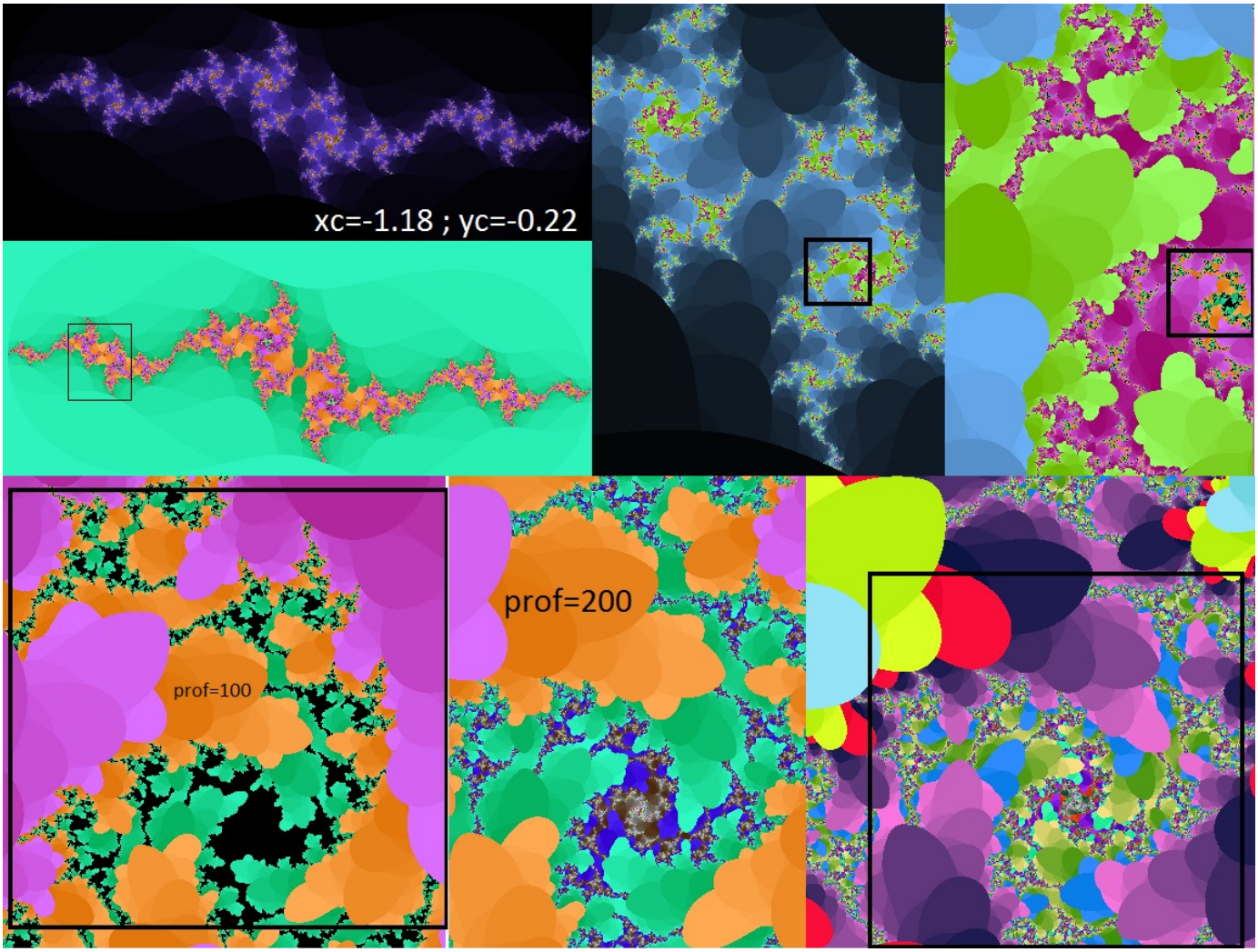
```
def actualise():
    global prof,pixel,iPoint,xMin,xMax,yMin,yMax,treelle
    xI=float(entree1.get())
    yI=float(entree2.get())
    iPoint=Point(xI,yI)
    pixel=int(entree3.get())
    prof=int(entree4.get())
    if x2>0:
        xMin,yMin=x1*treelle/taille+xMin,y1*treelle/taille+yMin
        c=min(max(x1,x2)-min(x1,x2),max(y1,y2)-min(y1,y2))
        treelle=c*treelle/taille
        xMax,yMax=xMin+treelle,yMin+treelle
    trace()
def coin1(event):
    global x1,y1
    x1,y1=event.x,event.y
def coin2(event):
    global x1,y1,x2,y2
    x2,y2=event.x,event.y
    c=min(max(x1,x2)-min(x1,x2),max(y1,y2)-min(y1,y2))
    x1,y1=min(x1,x2),min(y1,y2)
    x2,y2=x1+c,y1+c
    retrace()
    cadre.create_rectangle(x1,y1,x2,y2,width=4)
def dezoom():
    global xMin,xMax,yMin,yMax,treelle
    xMin,xMax,yMin,yMax,treelle=-2,2,-2,2,4
    cadre.delete(ALL)
    trace()
def calcule(p):
    p1=p
    for i in range(prof):
        p1=p1.suivant()
        if p1.distance()>=2: return i
    return -1
def retrace():
    for pix in image:
        cadre.create_rectangle(\
            pix[0]*pixel,pix[1]*pixel,\
            (pix[0]+1)*pixel,(pix[1]+1)*pixel,\
            width=0,fill=pix[2])
```

```
def trace():
    global image
    image=[]
    for x in range(taille//pixel):
        for y in range(taille//pixel):
            p=Point(x*pixel*treelle/taille+xMin,y*pixel*treelle/taille+yMin)
            n=iteration(p)
            if n==-1: coul='black'
            else:
                coul,bleu='#0000',(prof-n)*255//prof
                if bleu>15 :coul+=hex(bleu)[2:]
                else:coul+='0'+hex(bleu)[2:]
            image.append([x,y,coul])
            cadre.create_rectangle(x*pixel,y*pixel,(x+1)*pixel,(y+1)*pixel,width=0,fill=coul)

fen=Tk()
fen.title('Ensemble de Julia du point c')
taille,treelle,prof,xMin,xMax,yMin,yMax=600,4,100,-2,2,-2,2
xI,yI,pixel,x1,x2,y1,y2,image=0.4,-0.2,4,0,0,0,0,[]
iPoint=Point(xI,yI)
cadre=Canvas(fen,width=600,height=600,bg='white')
cadre.grid(row=1,column=1,columnspan=10)
cadre.bind('<ButtonPress-1>',coin1)
cadre.bind('<ButtonRelease-1>',coin2)
Label(fen,text="xc=").grid(row=2,column=1)
entree1=Entry(fen,width=4)
entree1.insert(END,"0.4")
entree1.grid(row=2,column=2)
Label(fen,text="yc=").grid(row=2,column=3)
entree2=Entry(fen,width=4)
entree2.insert(END,"-0.2")
entree2.grid(row=2,column=4)
Label(fen,text="pixel=").grid(row=2,column=5)
entree3=Entry(fen,width=4)
entree3.insert(END,"4")
entree3.grid(row=2,column=6)
Label(fen,text="profondeur=").grid(row=2,column=7)
entree4=Entry(fen,width=4)
entree4.insert(END,"100")
entree4.grid(row=2,column=8)
Button(fen,text='Actualise',command=actualise).grid(row=2,column=9)
Button(fen,text='Dézoomer',command=dezoom).grid(row=2,column=10)
trace()
fen.mainloop()
```

Dans cette version bleue, nous enregistrons les données de l'image (*x, y* et *n*) dans la liste *image* afin de pouvoir redessiner celle-ci sans devoir recalculer les valeurs de *n* (lorsqu'on ne fait que changer les paramètres d'affichages). Le bouton « Dézoomer » permet de revenir aux conditions initiales (on pourrait remettre également les valeurs par défaut de *pixel* et de *profondeur*).

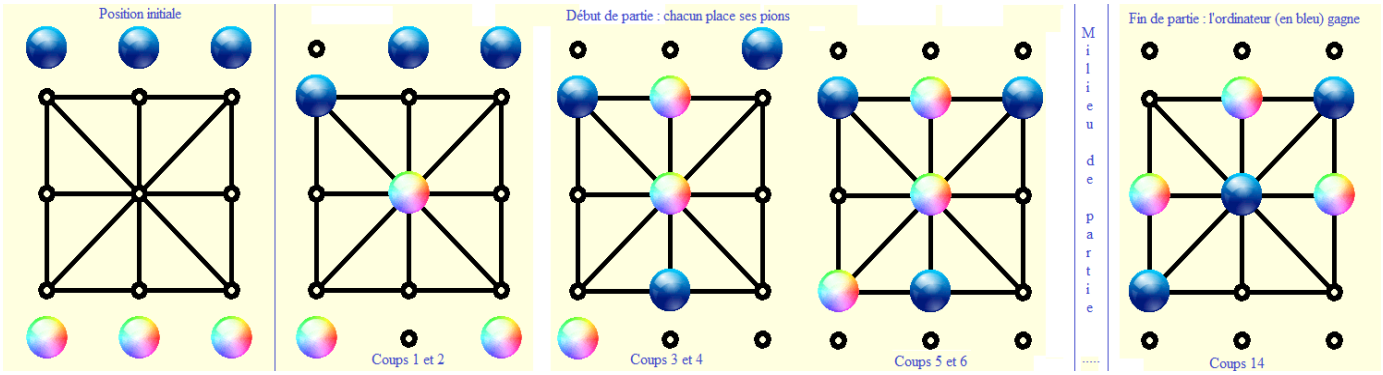
Notre objectif d'interactivité ne s'arrête pas avec cette version. Nous avons programmé une palette plus riche, en nous inspirant du modèle proposé par Laurent Signac dans son très sympathique livre « Divertissements mathématiques et informatiques » (paru en 2011 aux éditions HK, collection MiniMax) où cette situation est traitée dans le langage *Ruby*. Cet auteur propose ((4*n*)%256, 2*n*, (6*n*)%256) comme palette RVB (les composantes Rouge, Verte et Bleue sont des entiers entre 0 et 255 qui sont exprimées en hexadécimal (base 16) dans la syntaxe Python utilisée). Nous avons adopté quelque chose de similaire mais modifiable par des boutons, dans une 3ème ligne de notre fenêtre. Notre couleur associée à *n* est : ((*prof*−*n*)**col*[0])%256, (*prof*−*n*)**col*[1])%256, (*prof*−*n*)**col*[2])%256) où *col*=[3,5,7] par défaut. L'idée de multiplier le nombre *prof*−*n* (qui décroit de *prof* à 0 quand *n* croit de 0 à *prof*) par un coefficient réglable permet de modifier la palette de façon assez riche (et imprévisible, il faut bien le dire). Avec 10 valeurs au choix pour les paramètres *col*[i] (nous avons pris les 10 premiers nombres premiers mais ce choix n'apporte rien de spécial sinon la diversité), nous obtenons plus de 1000 palettes différentes pour apporter une très légère touche créative à cette application. Une case à cocher permet d'inverser les coloris en utilisant les valeurs de *n* au lieu de celles de *prof*−*n* dans le calcul des composantes de la couleur. Le fonctionnement des boutons ne prévoit pas l'envoi d'un paramètre (on exécute une fonction sans arguments) mais nous avons six boutons pour régler les valeurs des *col*[i] et nous ne voulons pas écrire six fonctions différentes. La solution à ce problème réside dans l'utilisation de fonctions *lambda*, une particularité de Python, qui s'écrit simplement : *lambda*:*palette*(0,0) pour lancer la fonction *palette*() avec les deux arguments 0 et 0. Pour construire le bouton « rouge- » qui, en lançant *palette*(0,0), diminue la valeur de *col*[0], l'instruction complète est : *Button*(*fen*, *text*='rouge-', *command*=*lambda*:*palette*(0,0), *bg*='red').*grid*(*row*=3, *column*=1).

Le résultat de ces améliorations est visible au début de cette partie et dans l'image qui suit. Nous pourrions encore enrichir l'interactivité de ce programme en proposant différentes options d'enregistrement ou d'impression de l'image, ou en combinant une passerelle entre ces ensembles de Julia et l'ensemble de Mandelbrot auxquels ils sont liés (nous avions fait cela il y a quelques années dans un programme *Java* qui se trouve sur mathadomicile). L'intérêt cependant de ces améliorations est amoindri par les temps d'attente pour recalculer ou même redessiner une de ces images. Il faudrait d'abord réduire ceux-ci pour espérer motiver de nouveaux développements.



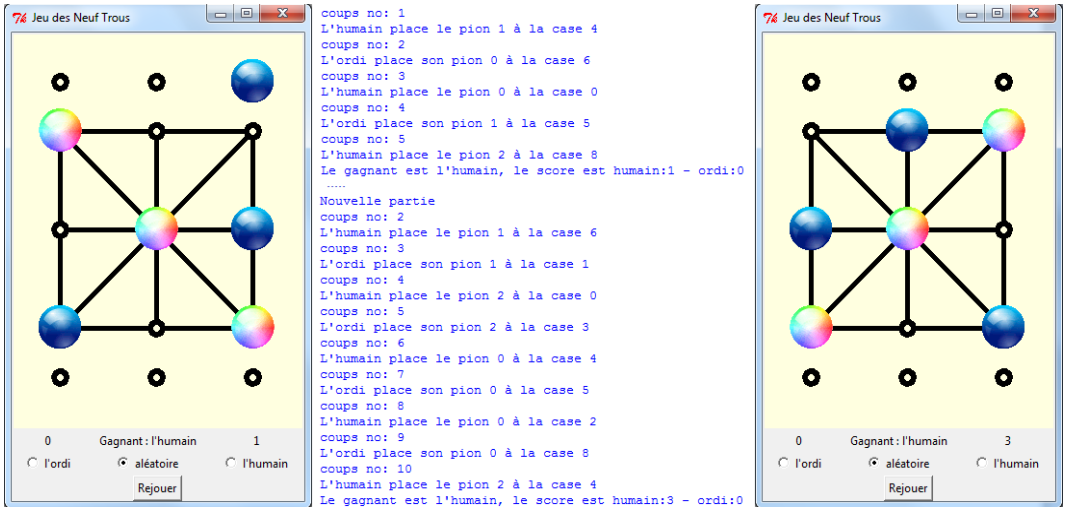
c) Donnons-nous un objectif un peu plus ludique que la simple exploration d'une image statique, fusse t-elle magnifique, et un peu plus ambitieux que la programmation du sudoku où les contraintes ne laissent aucune possibilité de variation. Un jeu simple à deux joueurs dont l'un des joueurs est l'ordinateur et qui nécessite une forme d'intelligence de sa part : le jeu des neuf trous appelé aussi *tapatan*. Le plateau de jeu est une grille carrée de 3 sur 3 et chaque joueur dispose de trois pions qu'il doit aligner sur le plateau. Au début, chacun à tour de rôle pose un pion, n'importe où sur le plateau. Une fois les six pions posés, chacun peut déplacer un de ses pions d'une position, selon un des tracés figurés sur le plateau (le long des six lignes ou des deux diagonales), si le nouvel emplacement est libre. Programmer ce jeu avec un compteur des parties gagnées/perdues et un dispositif permettant de choisir si c'est le joueur qui commence (plus facile), si c'est l'ordinateur (plus difficile) ou bien si ce choix est laissé au hasard (équilibré). Dans un 1er temps, on peut se consacrer à la réalisation du jeu, sans chercher une réponse intelligente de la part de l'ordinateur (rendre ses choix aléatoires).

On dessiner le plateau dans un *Canvas* cadre une fois pour toute. Les pions sont créés à partir de deux images *photo1* et *photo2 (*au format gif) qui sont utilisées pour instancier des variables pion0[0], pion0[1] et pion0[2] pour les trois pions de l'ordinateur (désigné par 0 ou ordi) et pion1[*i*] pour ceux de l'humain (*i* variant de 0 à 2 pour l'humain désigné par 1). L'emplacement de ces pions est réglé par les coordonnées qui sont initialisées (pion0=[] puis pion0.*append*(cadre.*create_image*(*i**100+50,50,*image*=photo1) crée le pion0[0]), utilisées ultérieurement (avec [*xDeb*,*yDeb*]=cadre.*coords*(pion1[*i*]), les variables *xDeb* et *yDeb* contiennent les coordonnées de pion1[*i*]) ou modifiées (cadre.*coords*(pion1[*i*],*x1*,*y1*) affecte les coordonnées *x1* et *y1* au pion pion1[*i*]). Le mouvement des pions est lié aux mouvements de la souris : le clic déclenche la fonction *clic*() qui détecte que le pion *choixPion* a été sélectionné (on ne s'intéresse qu'aux pions de l'humain car l'ordinateur bouge lui-même ses pions) en acceptant un certain domaine de variation des coordonnées du clic (*x.event* et *y.event*) autour de coordonnées du pion ; le déplacement de la souris, bouton de gauche enfoncé, lance la fonction *drag*() qui redessine le pion lors de son déplacement ; le relâchement de la souris lance la fonction *lache*() qui examine le nouvel emplacement, fixe le pion à cet emplacement s'il est vide et s'il correspond à un placement valide (fonction *avoisine*() de la classe *Jeu*).

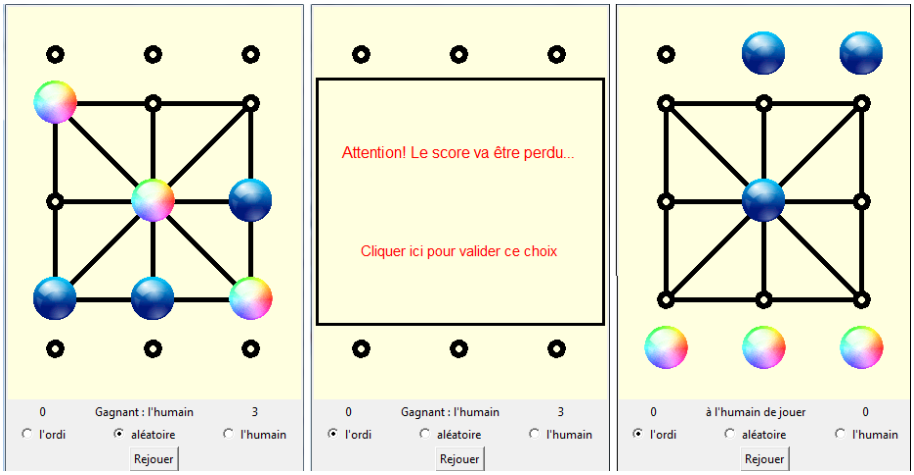


La classe *Jeu* que nous avons définie n'est pas forcément indispensable. Elle prend en arguments les emplacement des six pions et le joueur qui a la main (0 ou 1). Pour l'instant, elle dispose de la méthode *voisin*() qui se contente d'examiner les voisins valides, c'est-à-dire les placements valides de chacun des pions du joueur, et d'en renvoyer la liste. Cette classe est aussi utilisée pour sa méthode *finished*() qui détermine si une configuration de jeu est gagnante (dans ce cas, elle renvoie le joueur qui gagne, ou −1 si personne n'a encore gagné).

Lorsqu'un joueur a gagné, le score est modifié et un petit message est affiché à la place où, le reste du temps, on affiche que c'est à l'humain de joueur (lorsque l'ordinateur doit jouer, l'affichage le signale, mais ici, la réflexion de l'ordinateur est trop rapide pour que cela se voie). Le bouton « Rejouer » permet de rejouer une partie, ce qui repositionne les pions à l'extérieur du plateau et réinitialise les variables qui doivent l'être. Le mécanisme d'une telle interactivité, aussi modeste soit-elle, est assez délicat à régler. En plus des inévitables erreurs flagrantes de programmation, il y a souvent des petites erreurs bien cachées qui obligent à s'interroger sur les variables : à quels moments sont-elles initialisées, à quels moments changent-elles de valeurs ? Combien valent alors les autres ? La portée d'une variable est-elle celle que l'on attend (la portée d'une variable est souvent limitée à la fonction, ou à la boucle dans laquelle elle apparaît, à moins que la variable soit déclarée *global*, dans ce cas elle vaut la même chose pour toutes les parties qui reconnaissent cette variable globale...) ? Quelques instructions *print*() judicieusement placées aident souvent dans cette démarche de débogage.



Pour le changement du mode d'utilisation (les trois boutons « radio »), nous devons avertir l'utilisateur que cela va remettre les scores à 0-0 (car on n'utilise plus les mêmes règles). Afin de réaliser cela simplement, nous affichons juste un bandeau provisoire qui s'efface lorsqu'on clique dedans. On peut prévoir aussi qu'en cas d'erreur de manipulation, on souhaite éviter la remise à zéro des compteurs et remettre le réglage précédent. Pour l'instant, ces subtilités ne sont pas vraiment justifiées mais elles pourraient le devenir avec une application opérationnelle. Au stade de développement où l'on est, on pourrait aussi apporter quelques améliorations : l'animation étant encore assez pauvre, il serait judicieux de



simuler une certaine lenteur sur une trajectoire pour les mouvements du pion de l'ordinateur ; on peut aussi prévoir de jouer un son lors du déplacement d'un pion et un autre lors de son placement. On peut aussi envisager d'enregistrer le score sur un serveur, mais cela dépasse légèrement les objectifs de ce document. Consacrons-nous plutôt à la partie « intelligence artificielle » qui va faire de notre ordinateur un adversaire plus sérieux.

```
from tkinter import *
from random import randint
class Jeu :
    def __init__(self,mp=[-1,-1,-1],op=[-1,-1,-1],j=0):
        self.mesPlaces=[]
        self.ordiPlaces=[]
        self.joueur=j
        for i in range(3):
            self.mesPlaces.append(mp[i])
            self.ordiPlaces.append(op[i])
    def voisin(self):#retourne une liste contenant [pion à bouger,place libre à occuper]
        voisin=[]
        for i in range(3):
            for j in range(9):
                if j not in self.mesPlaces and j not in self.ordiPlaces:
                    if self.joueur==1 and self.isVoisine(self.mesPlaces[i],j):voisin.append([i,j])
                    elif self.joueur==0 and self.isVoisine(self.ordiPlaces[i],j):voisin.append([i,j])
        return voisin
    def isVoisine(self,c1,c2):
        if c1==4 or c2==4: return True
        if c1>c2:c1,c2=c2,c1
        if (c1==0 and (c2==1 or c2==3))or(c1==1 and c2==2)or(c1==2 and c2==5)or\
           (c1==3 and c2==6)or(c1==5 and c2==8)or(c1==6 and c2==7)or(c1==7 and c2==8): return True
        return False
    def finished(self):#retourne 1 si l'humain gagne, 0 si c'est l'ordi, -1 dans les autres cas
        winner=-1
        gagne=[[0,3,6],[1,4,7],[2,5,8],[0,1,2],[3,4,5],[6,7,8],[0,4,8],[2,4,6]]
        if sorted(self.mesPlaces) in gagne: winner=1
        if sorted(self.ordiPlaces) in gagne: winner=0
        return winner
```

```
def ordinateur():
    global coup,possible,tour,debut,ordiPlaces
    choixPion,choixPlace,anciennePlace=-1,-1,-1
    if debut:
        choixPion=coup//2
        choixPlace=randint(0,8)#jeu provisoirement aléatoire
        while not possible[choixPlace]:choixPlace=randint(0,8)
    else:
        listeChoixPion=Jeu(mesPlaces,ordiPlaces).voisin()
        randy=randint(0,len(listeChoixPion)-1)
        choixPion=listeChoixPion[randy][0]
        choixPlace=listeChoixPion[randy][1]#jeu provisoirement aléatoire
        anciennePlace=ordiPlaces[choixPion]
    coup+=1
    print('coups no:',str(coup))
    print("L'ordi place son pion {} à la case {}".format(choixPion,choixPlace))
    possible[choixPlace]=False #on occupe une place qui était libre
    if coup==6: debut=False
    x1,y1=choixPlace%3*100+50,choixPlace//3*100+100
    cadre.coords(pion0[choixPion],x1,y1)
    ordiPlaces[choixPion]=choixPlace
    if notGagne(0):
        tour="l'humain"
        message.configure(text="à {} de jouer".format(tour))
    if not debut:
        possible[anciennePlace]=True #on libère une place
def clic(event):
    global detectionPion,choixPion,xDeb,yDeb,gagnant,avertissement
    x1,y1,detectionPion=event.x,event.y,False
    if avertissement and 0<x1<300 and 70<y1<330: eraseMessage()
    if tour=="l'humain" and gagnant==-1:
        for i in range(3):
            [xDeb,yDeb]=cadre.coords(pion1[i])
            if (x1-xDeb)**2+(y1-yDeb)**2<400:#estimation du rayon du pion à 20
                choixPion=i
                if debut and resteAjouer.count(choixPion)==0 :
                    choixPion=-1
                    break
                detectionPion=True
                break
```

```
def changeDebut():
    global coup,score,avertissement,previousChoix
    if coup!=0 and score!=[0,0]:
        avertissement=True
        fenMessage('Attention! Le score va être perdu...')
def fenMessage(s):#bandeau d'avertissement
    global band,mess1,mess2
    band=cadre.create_rectangle(5,75,298,325,width=3,fill='light yellow')
    mess1=cadre.create_text(150,150,text=s,font="Arial 12",fill='red')
    mess2=cadre.create_text(150,250,text='Cliquer ici pour valider ce choix'\
                            ,font="Arial 11",fill='red')
def eraseMessage():
    global coup,score,tour,band,mess1,mess2,avertissement
    score=[0,0]
    monScore.configure(text="0")
    ordiScore.configure(text="0")
    cadre.delete(band)
    cadre.delete(mess1)
    cadre.delete(mess2)
    avertissement=False
    rejouer()
def rejouer():
    global coup,resteAjouer,possible,debut,detectionPion,choixPion,mesPlaces\
           ,ordiPlaces,tour,gagnant,avertissement,band,mess1,mess2,previousChoix
    for i in range(3):
        cadre.coords(pion0[i],i*100+50,50)
        cadre.coords(pion1[i],i*100+50,350)
    coup,resteAjouer,possible=0,[0,1,2],[]
    debut,detectionPion,choixPion=True,False,-1
    mesPlaces,ordiPlaces,gagnant=[-1,-1,-1],[-1,-1,-1],-1
    if avertissement:
        cadre.delete(band)
        cadre.delete(mess1)
        cadre.delete(mess2)
        choixDebut.set(previousChoix)
        avertissement=False
    previousChoix=choixDebut.get()
    for i in range(9): possible.append(True)
    if choixDebut.get()==0:tour=joueurDebut[0]
    elif choixDebut.get()==1:tour=joueurDebut[randint(0,1)*2]
    else:tour=joueurDebut[2]
    message.configure(text="à {} de jouer".format(tour))
    if tour=="l'ordi": ordinateur()
    print('\nNouvelle partie')
def notGagne(j):
    global coup,gagnant
    if coup<5:return True
    gagnant=Jeu(mesPlaces,ordiPlaces,j).finished()
    if gagnant>=0:
        message.configure(text="Gagnant : {}".format(joueurDebut[gagnant*2]))
        score[gagnant]+=1
        monScore.configure(text=str(score[1]))
        ordiScore.configure(text=str(score[0]))
        print("Le gagnant est {}, le score est humain:{} - ordi:{}".\
              format(joueurDebut[gagnant*2],score[1],score[0]))
        return False
    return True
def tracePlateau():
    cadre.create_rectangle(50,100,250,300,width=5)
    cadre.create_line(50,100,250,300,width=5,fill='black')
    cadre.create_line(50,300,250,100,width=5,fill='black')
    cadre.create_line(50,200,250,200,width=5,fill='black')
    cadre.create_line(150,100,150,300,width=5,fill='black')
    for i in range(15):
        xt,yt=i%3*100+50,i//3*100
        if i<3:yt+=50
        if i>11:yt-=50
        cadre.create_oval(xt-7,yt-7,xt+7,yt+7,width=5,fill='black')
        cadre.create_oval(xt-3,yt-3,xt+3,yt+3,width=0,fill='light yellow')
def drag(event):
    x1,y1=event.x,event.y
    if detectionPion:
        if x1<0:x1=0
        if x1>300:x1=300
        if y1<50:y1=50
        if y1>350:x1=350
        cadre.coords(pion1[choixPion],x1,y1)
```

```
def lache(event):
    global detectionPion,possible,choixPion,coup,tour,debut,mesPlaces,resteAjouer,xDeb,yDeb
    if detectionPion and gagnant==-1:
        x1,y1,x2,y2,choixPlace=event.x,event.y,0,0,-1
        for i in range(9):
            x2,y2=i%3*100+50,i//3*100+100
            if (x2-x1)**2+(y2-y1)**2<400:
                choixPlace=i
                break
        if not debut:listeChoixPion=Jeu(mesPlaces,ordiPlaces,1).voisin()
        if possible[choixPlace] and choixPlace>=0 and \
           (debut or (not debut and [choixPion,choixPlace] in listeChoixPion)):
            anciennePlace=mesPlaces[choixPion]
            cadre.coords(pion1[choixPion],x2,y2)
            mesPlaces[choixPion]=choixPlace
            coup+=1
            print('coups no:',str(coup))
            print("L'humain place le pion {} à la case {}".format(choixPion,choixPlace))
            if notGagne(1):
                tour="l'ordi"
                message.configure(text="à {} de jouer".format(tour))
                resteAjouer[choixPion]=-1
                possible[choixPlace]=False #on occupe une place qui était libre
                if not debut:
                    possible[anciennePlace]=True #on libère une place
                    print("possible:",possible)
                if coup==6: debut=False
                choixPion=-1
                ordinateur()
        else:cadre.coords(pion1[choixPion],xDeb,yDeb)
        detectionPion=False
```

```
fen=Tk()
fen.title('Jeu des Neuf Trous')
joueurDebut=["l'ordi","aléatoire","l'humain"]
tour=joueurDebut[randint(0,1)*2]
cadre=Canvas(fen,width=300,height=400,bg='light yellow')
cadre.grid(row=1,column=1,columnspan=3)
cadre.bind("<Button-1>",clic)
cadre.bind("<B1-Motion>",drag)
cadre.bind("<ButtonRelease-1>",lache)
ordiScore=Label(fen,text="0")
ordiScore.grid(row=2,column=1)
message=Label(fen,text="à {} de jouer".format(tour))
message.grid(row=2,column=2)
monScore=Label(fen,text="0")
monScore.grid(row=2,column=3)
choixDebut=IntVar()
choixDebut.set(1)
for i in range(3): # Création des trois 'boutons radio' :
    bouton=Radiobutton(fen,text=joueurDebut[i],variable=choixDebut,
                       value=i,command=changeDebut)
    bouton.grid(row=3,column=i+1)
bouton=Button(fen,text="Rejouer",command=rejouer)
bouton.grid(row=4,column=1,columnspan=3)
photo1=PhotoImage(file='buttonBleu.gif')
photo2=PhotoImage(file='buttonArcEnCiel.gif')
#initialisations des variables globales
debut,detectionPion,avertissement=True,False,False
possible,coup,score,resteAjouer=[],0,[0,0],[0,1,2]
mesPlaces,ordiPlaces,choixPion=[-1,-1,-1],[-1,-1,-1],-1
xDeb,yDeb,gagnant,previousChoix=0,0,-1,1
band,mess1,mess2=None,None,None
for i in range(9): possible.append(True)
pion0,pion1=[],[]
tracePlateau()
for i in range(3):# Création des six pions :
    pion0.append(cadre.create_image(i*100+50,50,image=photo1))#ordi
    pion1.append(cadre.create_image(i*100+50,350,image=photo2))#humain
if tour=="l'ordi": ordinateur()
fen.mainloop()
```

La question qui se pose est de nature algorithmique. Que doit faire l'ordinateur pour bien jouer : il a, à sa disposition, une liste de possibilités valides (*listeChoixPion*) parmi lesquelles il doit choisir la plus prometteuse. On pourrait se contenter d'éliminer les possibilités qui permettraient à l'humain de gagner au coup suivant. Ce serait déjà un peu mieux que de faire un choix totalement aléatoire. La solution idéale serait d'examiner les prolongements de chacun des coups possibles jusqu'à l'issue la pire (perte de l'ordinateur), la meilleure (gain de l'ordinateur), ou jusqu'à la conviction que la partie est nulle (un cycle

sans vainqueur). Cette solution risque d'être coûteuse en temps d'exécution aussi nous allons examiner l'option alternative qui consiste à offrir un choix de plusieurs niveaux pour la réflexion de l'ordinateur : au niveau 0, il répond aléatoirement. Au niveau 1, il examine les prolongements jusqu'au coup suivant. Au niveau 2, il les examine jusqu'à deux coups (après le coup de l'ordinateur, l'humain joue, puis l'ordinateur, puis encore l'humain). Un humain doit, avec un peu d'attention et d'entraînement, jouer à ce niveau 2. Pour être un peu plus difficile à battre, il faudrait pouvoir faire jouer l'ordinateur jusqu'au niveau 3.
L'algorithme du *minimax* est parfaitement adapté à cette situation. Parmi toutes les possibilités de jeu, l'ordinateur choisit celle qui maximise son gain. Lorsqu'il examine les prolongements de son coup, parmi toutes les possibilités de jeu de l'humain, il choisit celle qui minimise son gain. Arrivé à la profondeur maximale de sa réflexion ou bien à une situation où l'un des deux joueurs a gagné, il note 0 les configurations indécises, +10 celles où il gagne et −10 celles où il perd. Cet algorithme remplace le choix aléatoire fait dans la fonction *ordinateur*() et quelques fonctions nouvelles s'ajoutent à notre programme : *chercheMaxi*() et *chercheMini*() sont les deux fonctions qui s'appellent alternativement, jusqu'à atteindre la profondeur désirée ; *evaluer*() note les différentes configurations aux extrémités (feuilles) de l'arbre de la réflexion de l'ordinateur. Nous invitons le lecteur curieux d'en apprendre davantage sur l'algorithme du *minimax* d'effectuer quelques recherches à ce sujet sur internet.

```
class Jeu :
    ....
    def setPion(self,pion,case,joueur):
        if joueur==0:self.ordiPlaces[pion]=case
        else: self.mesPlaces[pion]=case
    def getPion(self,pion,joueur):
        if joueur==0: return self.ordiPlaces[pion]
        else: return self.mesPlaces[pion]
```

```
    ....
def evaluer(j):
    gagnant=j.finished()
    if gagnant==0:return [0,0,10]
    elif gagnant==1: return [0,0,-10]
    else:return [0,0,0]
def chercheMaxi(mp,op,prof,cp): # L'ordi joue
    max_pion,max_case,maximum=-1,-1,-100
    mesPlaces,ordiPlaces=[-1,-1,-1],[-1,-1,-1]
    for i in range(3):
        mesPlaces[i],ordiPlaces[i]=mp[i],op[i]
    jeu=Jeu(mesPlaces,ordiPlaces)
    if prof==0 or jeu.finished()>=0: return evaluer(jeu)
    if cp//2<3:
        pion=cp//2#l'ordi joue ses pions dans l'ordre
        for i in range(9):
            if i not in ordiPlaces and i not in mesPlaces:#case vide
                jeu.setPion(pion,i,0)
                score=chercheMini(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp+1)[2]
                if score>maximum:
                    maximum=score
                    max_case=i
                    max_pion=pion
                jeu.setPion(pion,-1,0)
    else:
        listeChoixPion=jeu.voisin()
        for i in range(len(listeChoixPion)):
            pion=listeChoixPion[i][0]
            case=listeChoixPion[i][1]
            ancienneCase=jeu.getPion(pion,0)
            jeu.setPion(pion,case,0)
            score=chercheMini(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp)[2]
            if score>maximum:
                maximum=score
                max_case=case
                max_pion=pion
            jeu.setPion(pion,ancienneCase,0)
    return [max_pion,max_case,maximum]
```

```
def ordinateur():
    global coup,possible,tour,debut,ordiPlaces
    choixPion,choixPlace,anciennePlace=-1,-1,-1
    [choixPion,choixPlace,c]=chercheMaxi(mesPlaces,ordiPlaces,6,coup)
    if not debut:anciennePlace=ordiPlaces[choixPion]
    coup+=1
    print('coups no:',str(coup))
    print("L'ordi place son pion {} à la case {}".format(choixPion,choixPlace))
    possible[choixPlace]=False #on occupe une place qui était libre
    if coup==6: debut=False
    x1,y1=choixPlace%3*100+50,choixPlace//3*100+100
    cadre.coords(pion0[choixPion],x1,y1)
    ordiPlaces[choixPion]=choixPlace
    if notGagne(0):
        tour="l'humain"
        message.configure(text="à {} de jouer".format(tour))
    if not debut:
        possible[anciennePlace]=True #on libère une place
def chercheMini(mp,op,prof,cp): # coup de l'humain
    min_pion,min_case,minimum=-1,-1,100
    mesPlaces,ordiPlaces=[-1,-1,-1],[-1,-1,-1]
    for i in range(3):
        mesPlaces[i],ordiPlaces[i]=mp[i],op[i]
    jeu=Jeu(mesPlaces,ordiPlaces,1)
    if prof==0 or jeu.finished()>=0: return evaluer(jeu)
    if cp//2<3:
        for i in range(3):
            if mesPlaces[i]==-1:
                pion=i
                break
        for i in range(9):
            if i not in ordiPlaces and i not in mesPlaces:#case vide
                jeu.setPion(pion,i,1)
                score=chercheMaxi(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp+1)[2]
                if score<minimum:
                    minimum=score
                    min_case=i
                    min_pion=pion
                jeu.setPion(pion,-1,1)
    else:
        listeChoixPion=jeu.voisin()
        for i in range(len(listeChoixPion)):
            pion=listeChoixPion[i][0]
            case=listeChoixPion[i][1]
            ancienneCase=jeu.getPion(pion,1)
            jeu.setPion(pion,case,1)
            score=chercheMaxi(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp)[2]
            if score<minimum:
                minimum=score
                min_case=case
                min_pion=pion
            jeu.setPion(pion,ancienneCase,1)
    return [min_pion,min_case,minimum]
```

Voilà notre adversaire bien doté maintenant et difficile à battre, ce qui était notre objectif. Au niveau 3, l'ordinateur examine jusqu'à 6 coups successifs (3 aller-retours ordinateur-humain), ce qui représente une durée sensible mais raisonnable. Nous laissons au lecteur le soin d'améliorer l'animation. Nous n'avons pas rendu la profondeur réglable par l'utilisateur (cela ne semble pas nécessaire ici), le lecteur pourra apporter ce petit changement et d'autres encore auxquels nous ne pensons pas. Par contre, ce jeu des neuf trous étant un programme censé intéresser des utilisateurs non-initiées à Python, il nous est apparu indispensable de transformer ce *tapatan2.py* en un *tapatan2.exe* − un fichier portable (on dit aussi « standalone ») qui peut être exécuté sans avoir Python sur son ordinateur. L'opération n'est pas simple à exécuter, c'est une des faiblesses reconnues de Python, mais avec beaucoup de patience, on finit par comprendre comment réaliser cela : un module externe (*py2exe*) est nécessaire, il faut le télécharger, l'installer, créer un fichier *setup.py* sur le modèle de notre illustration, et lancer la commande *python setup.py py2exe* depuis une « invite de commande » *Windows* (nous n'avons pas essayé sur d'autres systèmes d'exploitation). C'est tout ! On obtient alors un ensemble de fichiers que l'on peut compresser avec un utilitaire en un fichier *zip*. Notre archive *tapatan.zip* est complètement autonome (en principe) : on peut la copier sur une clef ou la mettre sur un site, l'utilisateur final la décompresse et lance le fichier *tapatan2.exe* qui y est présente, sans même

savoir que c'est un programme Python.

```
from distutils.core import setup
import py2exe
Mydata_files = [('image', ['C:/Python34/image/buttonArcEnCiel.gif','C:/Python34/image/buttonBleu.gif'])]
setup(name="tapatan",version="1.0",description="Jeu des neuf trous ou Tapatan",author="PM",zipfile=None,
windows=['tapatan2.py'],data_files=Mydata_files,options={"py2exe":{"compressed":1,"bundle_files":2}})
```

d) Pour finir en beauté, envisageons une application graphique qui utilise un menu. Fixons un but à ce programme : réaliser des rosaces, des figures ayant *n* axes de symétrie concourants (*n* est variable, supérieur ou égal à 2). Les « pinceaux » ont une couleur et une taille réglables. On trace des lignes de plusieurs types (des segments, des arcs de cercles, des lignes libres, etc.) et on doit pouvoir gommer (supprimer des éléments). Dans un 2ème temps, on peut prévoir un enregistrement de l'image ainsi crée au format *png* ou une exportation de cette image dans le « presse-papier ».

L'installation du menu Python est réalisée dans une *Frame* (un *widget* pouvant contenir d'autres *widgets*) que l'on nomme mBar (mBar=*Frame*(fen), fen étant le conteneur *Tk*). On dispose ensuite des *Menubutton* dans cette *Frame* (par exemple, fich=*Menubutton*(mBar, *text*='Fichier') est un bouton portant l'étiquette « Fichier ») et à l'intérieur de ceux-ci, on met les différentes options par un dispositif un peu complexe :

me1=Menu(fich) on appelle me1 l'ensemble des options qui apparaissent dans fich

me1.*add_command*(*label*='Nouveau',*command*=effacer) on ajoute les différentes options avec la méthode

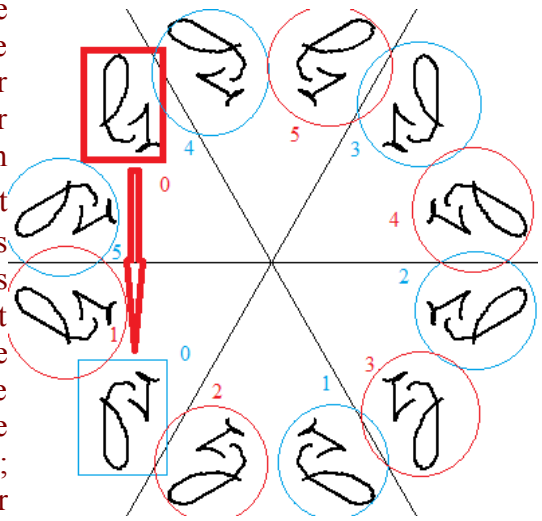
…. *add_command* qui prend en arguments un nom (label), la fonction qui sera appelée (*command*)

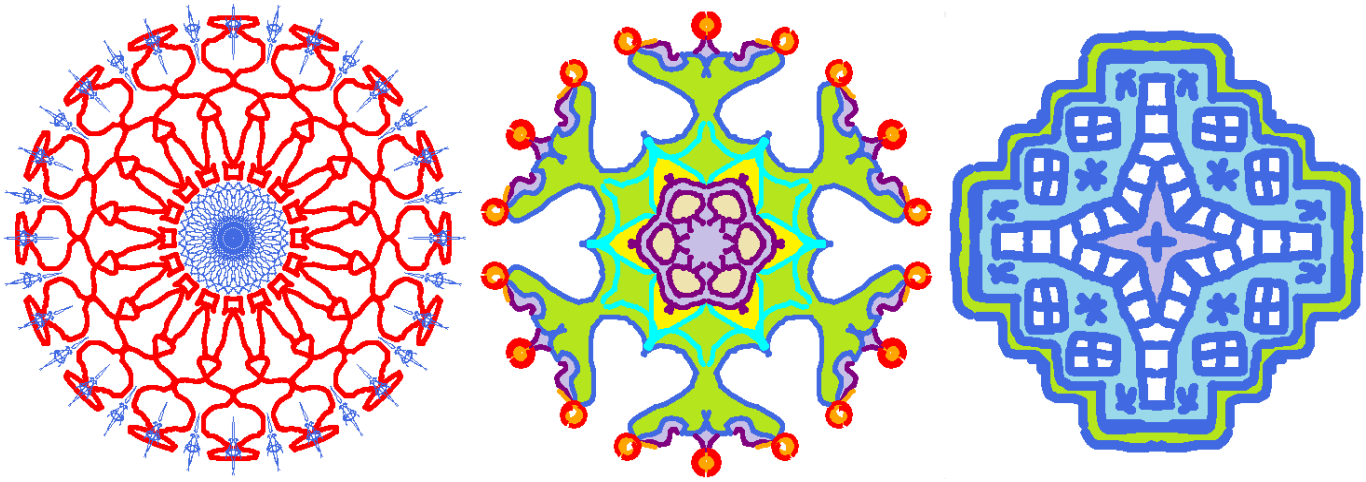
fich.*configure*(*menu*=me1) on enregistre notre menu en configurant le *Menubutton* fich avec ces options de *Menu* me1

La complexité augmente encore si l'on veut qu'une option de menu ouvre un volet contenant d'autres options de menu. Nous avons fait cela pour le *Menubutton* « Pinceau » qui contient les volets « forme », « taille » et « ordre » qui, chacun, ouvre des menus en cascade (la méthode Python est *add_cascade*). Dans le menu « forme » nous avons disposé un autre volet qui ouvre en cascade d'autres menus, l'ensemble pouvant être complexifié selon ce principe général. On peut ajouter autre chose que des menu de commande : des *checkbutton* (par la méthode *add_checkbutton*) qui sont liés à une commande mais surtout à une variable *IntVar* dont le contenu (obtenu grâce à la méthode *get*()) renseigne sur l'état de la case (1:cochée – 0:décochée) ; des *radiobutton* (par la méthode *add_radiobutton*) qui sont aussi liés à une commande et à une variable *IntVar*. Tous les *radiobutton* liés à une même variable forment un ensemble dans lequel on reconnaît l'option activée par la *value*, toujours obtenue par *get*(). On pourrait utiliser encore d'autres dispositifs (des *spinbox*, des cases d'entrée de texte, des curseurs, etc.) mais nous préférons tester le fonctionnement global du menu en n'utilisant que les dispositifs de base (nous aurions d'ailleurs pu nous limiter à la méthode *add_command*, la plus basique). Une 1ère version du programme avec un menu fonctionnant comme prévu est tout d'abord mise au point. Seule, la possibilité d'écrire dans le cadre est implémentée ici, les différentes options de pinceau seront mises au point progressivement, de même que la spécificité du dessin de la rosace (symétries+rotations).

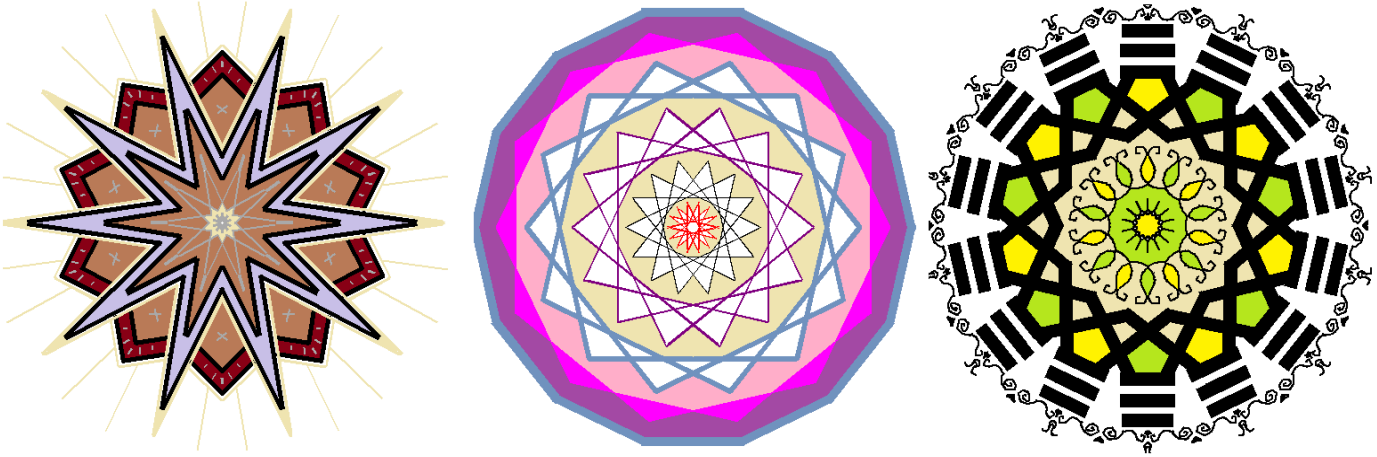
```
from tkinter import *
from math import *
def effacer():
    cadre.delete(ALL)
    if axes.get()==1:traceAxes()
def record():
    None
def clic(event):
    global coordL
    x1,y1=event.x,event.y
    coordL=[]
    if 0<x1<500 and 0<y1<500: coordL.append([x1,y1])
def drag(event):
    global coordL
    x1,y1=event.x,event.y
    coordL.append([x1,y1])
    cadre.create_line(coordL,width=taille,joinstyle=ROUND,fill=color)
def deleteGrille():
    global grilV,grilH
    for i in range(50):
        cadre.delete(grilV[i])
        cadre.delete(grilH[i])
def traceGrille():
    global grilV,grilH
    grilV,grilH=[],[]
    for i in range(50):
        grilV.append(cadre.create_line(i*10,0,i*10,500,width=1,fill='black'))
        grilH.append(cadre.create_line(0,i*10,500,i*10,width=1,fill='black'))
def traceAxes():
    global axe
    axe=[]
    for i in range(nbrAxes):
        axe.append(cadre.create_line(250+250*cos(i*pi/nbrAxes),250+250*sin(i*pi/nbrAxes),250+250*cos(pi+i*pi/nbrAxes),
                                     250+250*sin(pi+i*pi/nbrAxes),width=1,fill='black') )
def deleteAxes():
    global axe
    for i in range(nbrAxes):
        cadre.delete(axe[i])
def setCoul():
    global color
    color=col[couleur.get()]
def setPinc():
    global pinceau
    pinceau=pin[forme.get()]
def taillePinc():
    global taille
    taille=large.get()
def setOrdre():#changement du nombre d'axes
    global nbrAxes
    if axes.get()==1:deleteAxes()
    nbrAxes=nbr.get()
    if axes.get()==1:traceAxes()
def choixGrille():
    if grille.get()==1:traceGrille()
    else:deleteGrille()
def choixAxes():
    if axes.get()==1:traceAxes()
    else:deleteAxes()
fen=Tk()
fen.title('Rosaces V.1')
mBar=Frame(fen)
mBar.pack()
cadre=Canvas(fen,bg='white',height=500,width=500,borderwidth=2)
cadre.pack()
cadre.bind("<Button-1>",clic)
cadre.bind("<B1-Motion>",drag)
nbr,large,axes,grille,forme,couleur=IntVar(),IntVar(),IntVar(),IntVar(),IntVar(),IntVar()#variables tkinter
colF=['noir','bleu roi','rouge','vert clair','violet','cyan','marob','orange','jaune','vert foncé','gris']
col=['black','royal blue','red','light green','purple','cyan','maroon','orange','yellow','dark green','dark grey']
pinF=['ligne libre','segment','arc de cercle','triangle','carré','étoile']
pin=['lib','seg','arc','tri','car','etc']
nbr.set(2)
large.set(2)
couleur.set(0)
axes.set(1)
grille.set(0)
forme.set(0)
fich=Menubutton(mBar,text='Fichier',relief=RAISED)# Menu <Fichier> #
fich.pack(side=LEFT,padx='5')
me1=Menu(fich)
me1.add_command(label='Nouveau',underline=0,command=effacer)
me1.add_command(label='Enregistrer',underline=0,command=record)
me1.add_command(label='Quitter',underline=0,command=fen.quit)
fich.configure(menu=me1)
# Menu <Pinceau> #
pinc=Menubutton(mBar,text='Pinceau',relief=RAISED)# Menu <Pinceau> #
pinc.pack(side=LEFT,padx='5')
m0=Menu(pinc)
m1=Menu(m0)#couleur
for i in range(11):
    if i==0:m1.add_radiobutton(label=colF[i],foreground ='white',background=col[i],variable=couleur,value=i,command=setCoul)
    else:m1.add_radiobutton(label=colF[i],foreground ='black',background=col[i],variable=couleur,value=i,command=setCoul)
m0.add_cascade(label='couleur',underline=0,menu=m1)
pinc.configure(menu=m0)
m1=Menu(m0)#forme
m1.add_radiobutton(label='ligne libre',underline=1,variable=forme,value=0,command=setPinc)
m1.add_radiobutton(label='segment',underline=1,variable=forme,value=1,command=setPinc)
m1.add_radiobutton(label='arc de cercle',underline=1,variable=forme,value=2,command=setPinc)
m0.add_cascade(label='forme',underline=0,menu=m1)
m2=Menu(m1)
m2.add_radiobutton(label='Triangles',underline=1,variable=forme,value=3,command=setPinc)
m2.add_radiobutton(label='Carrés',underline=1,variable=forme,value=4,command=setPinc)
m2.add_radiobutton(label='Etoiles',underline=1,variable=forme,value=5,command=setPinc)
m1.add_cascade(label='polygones',underline=0,menu=m2)
pinc.configure(menu=m0)
m1=Menu(m0)#forme
m1.add_radiobutton(label='ligne libre',underline=1,variable=forme,value=0,command=setPinc)
m1.add_radiobutton(label='segment',underline=1,variable=forme,value=1,command=setPinc)
m1.add_radiobutton(label='arc de cercle',underline=1,variable=forme,value=2,command=setPinc)
m0.add_cascade(label='forme',underline=0,menu=m1)
m2=Menu(m1)
m2.add_radiobutton(label='Triangles',underline=1,variable=forme,value=3,command=setPinc)
m2.add_radiobutton(label='Carrés',underline=1,variable=forme,value=4,command=setPinc)
m2.add_radiobutton(label='Etoiles',underline=1,variable=forme,value=5,command=setPinc)
m1.add_cascade(label='polygones',underline=0,menu=m2)
pinc.configure(menu=m0)
m1=Menu(m0)#taille
for (i,lab) in [(1,'1 px'),(2,'2 px'),(5,'5 px'),(10,'10 px'),(15,'15 px'),(20,'20 px'),(30,'30 px')]:
    m1.add_radiobutton(label=lab,underline=1,variable=large,value=i,command=taillePinc)
m0.add_cascade(label='taille',underline=0,menu=m1)
pinc.configure(menu=m0)
optMenu=Menubutton(mBar,text='Options',relief=RAISED)# Menu <Options> #
optMenu.pack(side=LEFT,padx='3')
m0=Menu(optMenu)
m0.add_command(label='Activer :',foreground='blue')
m0.add_checkbutton(label='Axes de symétrie',variable=axes,command=choixAxes)
m0.add_checkbutton(label='Grille',variable=grille,command=choixGrille)
m0.add_separator()
m0.add_command(label='Symétrie :',foreground='blue')
m1=Menu(m0)#ordre
for i in range(2,25):
    m1.add_radiobutton(label='{} axes'.format(i),underline=1,variable=nbr,value=i,command=setOrdre)
m0.add_cascade(label='ordre',underline=0,menu=m1)
optMenu.configure(menu=m0)
color,pinceau,nbrAxes,taille,axe,grilV,grilH,coordL='black','lib',2,2,[],[],[],[]#variables globales
traceAxes()
fen.mainloop()
```

Pour dessiner une ligne « libre », comme avec un crayon, il faut utiliser la méthode *create_line*(). Cette méthode crée un segment si l'on donne quatre nombres comme premiers arguments (ce sera alors *create_line*(x1,y1,x2,y2)), mais si on donne une liste de coordonnées plus longue, la méthode crée une ligne apparente, constituée en réalité de micro-segments. C'est la méthode retenue, par défaut, pour dessiner dans le *Canvas* cadre. Notre objectif étant la création de rosaces, nous devons simultanément dessiner quatre fois le nombre d'axes cette ligne et ses images par symétrie et par rotation autour du centre. S'il y a trois axes, comme sur l'illustration, le motif (ℓ_1) est dessiné six fois par rotation d'un angle égal à $\frac{i\times\pi}{3}$, i variant de 0 à 5. La rotation d'un point est réalisée grâce à la fonction *rotatione*() qui travaille sur les coordonnées. Nous devons penser que les coordonnées données par le clic de souris sont dans un repère lié au cadre (l'origine est en haut à gauche) alors que nous voulons faire tourner notre point autour du centre de ce cadre. Un petit changement d'origine est donc nécessaire. Ensuite, nous utilisons une formule issue directement du cours de trigonométrie ($x'=x\cos(\alpha)+y\sin(\alpha)$; $y'=-x\sin(\alpha)+y\cos(\alpha)$). Pour les autres images, obtenues par symétrie des premières par rapport aux différents axes, nous prenons juste le symétrique (encadré en bleu) du motif original (encadré en rouge) et nous faisons subir à ce symétrique les différentes rotations. Ceci est rendu possible par le fait que l'un des axes est toujours horizontal ; les coordonnées du symétrique sont alors très simples à calculer (seul y est changé en $500-y$, si 500 est la hauteur du cadre).

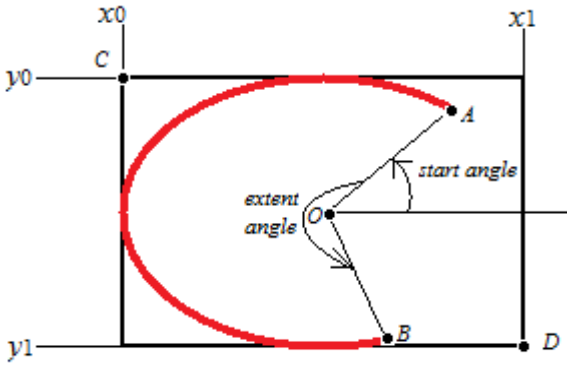




Les rosaces obtenues par ces lignes libres sont déjà assez satisfaisantes, surtout lorsqu'on les reprend dans un petit outil graphique comme le « *Paint* » de *Windows*, pour colorier l'intérieur des surfaces délimitées par nos lignes (voir notamment la rosace du milieu ci-dessus). Mais nous voulons ajouter maintenant les autres options de pinceau. La plus simple est l'option « segment » : au clic de souris, on enregistre les coordonnées du point initial pour les deux extrémités du segment dessiné, et pendant le glissé de la souris (*drag*), on modifie les coordonnées du 2ème point, ce qui a pour effet de modifier le segment dessiné au lieu d'en dessiner d'autres. Bien sûr, il faut simultanément dessiner les images de ce segment par les symétries et rotations de la rosace, mais pour cela on utilise le même principe que pour la ligne.

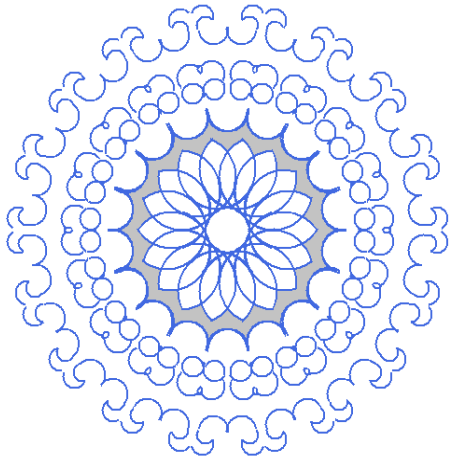
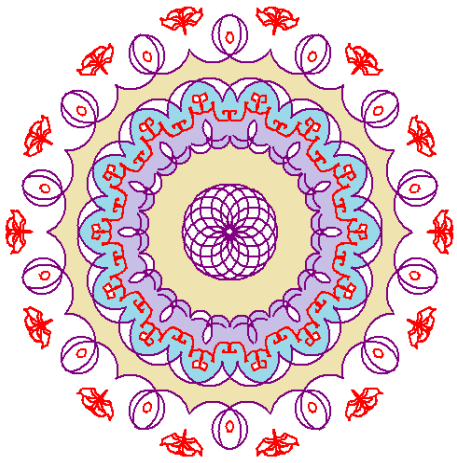
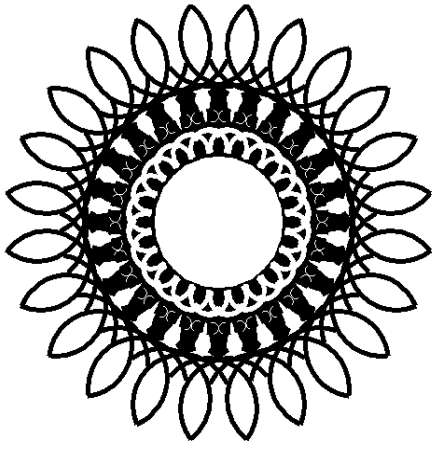


Pour dessiner les arcs de cercle, le travail mathématiques est un peu plus important, car la méthode *create_arc*() qui dessine des arcs de cercle nécessite qu'on précise un angle de départ (*start*) et un angle d'extension (*extent*). L'angle d'extension est toujours 180° si on veut dessiner des demi-cercles (c'est notre cas). Mais l'angle de départ, qui est l'angle entre le point situé le plus à droite sur le cercle et le point de départ choisi, va varier selon les valeurs des coordonnées du 2ème point et aussi, accessoirement, avec les rotations et symétries exercées sur ces points. Les coordonnées fournies à la méthode ne sont pas les coordonnées des extrémités *A* et *B* de l'arc, mais celles des deux coins de la diagonale du carré circonscrit au cercle (voir notre illustration qui montre le cas plus général d'un arc d'ellipse inscrite dans un rectangle). C'est peut-être là le point le plus difficile à régler, mathématiquement parlant. Nous laissons le lecteur cogiter notre solution qui réalise le dessin de ces arcs de cercle. Le résultat est assez saisissant et récompense largement ce petit effort cérébral.

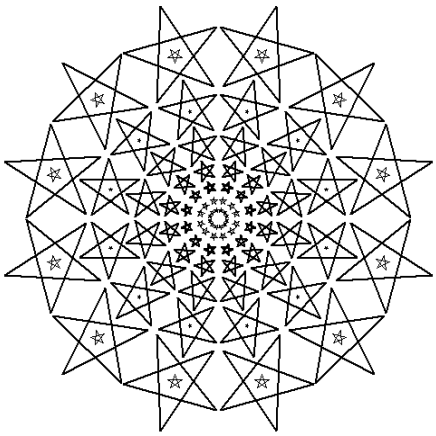
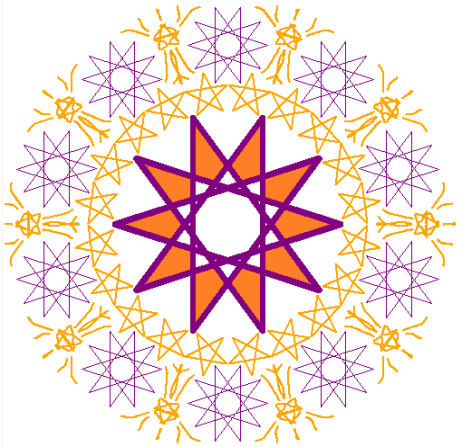
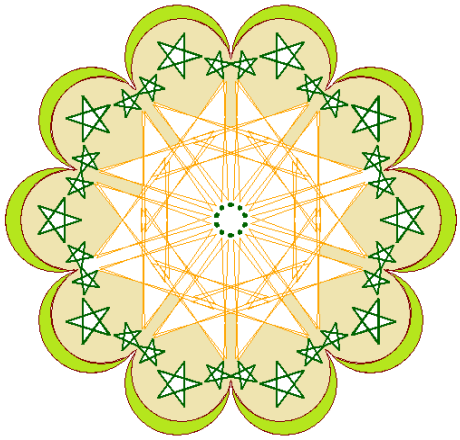



```
def clic(event):
    global coordL,ligne
    x1,y1=event.x,event.y
    if pinceau=='lib'
     .........
    elif pinceau=='arc' and 0<x1<500 and 0<y1<500:
        ligne=[]# contient les arcs (demi-cercles) définis par deux points
        coordL=[x1,y1,x1,y1]
        for i in range(4*nbrAxes):
            ligne.append(cadre.create arc(x1,y1,x1,y1,width=taille,start=0,extent=180,outline=color,style='arc',fill=''))
    .........
```

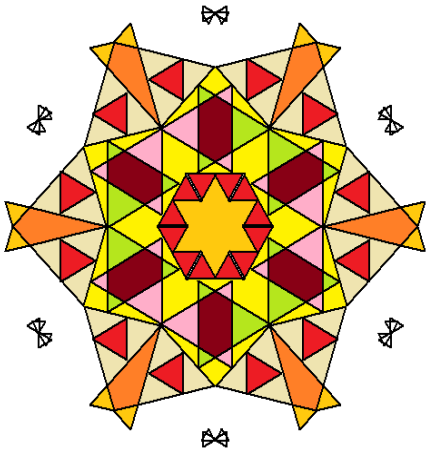
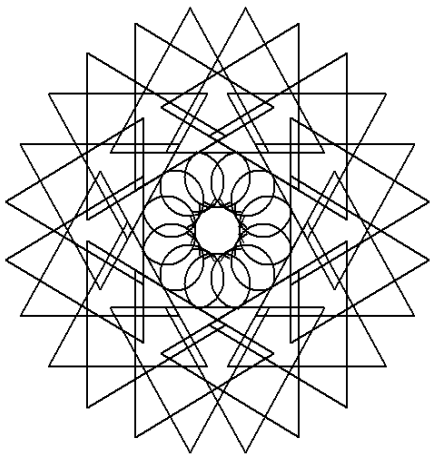
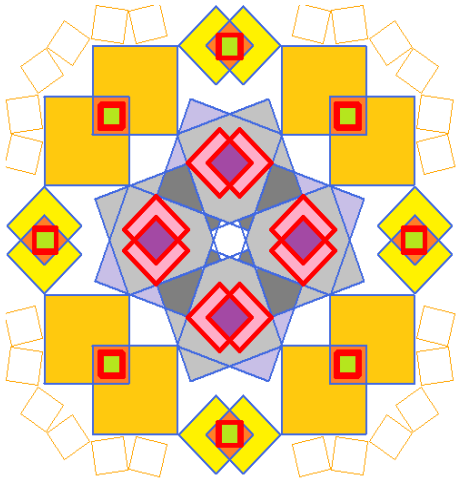
```
def drag(event):
    global coordL,ligne
    x1,y1=event.x,event.y
    if pinceau=='lib'
     .........
    elif pinceau=='arc' and 0<x1<500 and 0<y1<500:
        coordL[2],coordL[3]=x1,y1           # modification des coordonnées du 2e point
        x0,y0=(coordL[0]+coordL[2])/2,(coordL[1]+coordL[3])/2      # milieu
        R=sqrt((coordL[0]-coordL[2])**2+(coordL[1]-coordL[3])**2)/2# rayon
        x2,y2,x3,y3=x0-R,y0-R,x0+R,y0+R # coins opposés du carré contenant le cercle
        startAngle=asin((coordL[1]-coordL[3])/2/R)     # position angulaire de A si A est à droite
        if coordL[2]<coordL[0]:startAngle=pi-startAngle# position angulaire de A si A est à gauche
        for i in range(2*nbrAxes):
            cadre.itemconfigure(ligne[i],start=(startAngle+i*pi/nbrAxes)*180/pi)
            x0r,y0r=rotatione([x0,y0],i)           #seul le centre du cercle tourne
            cadre.coords(ligne[i],x0r-R,y0r-R,x0r+R,y0r+R)
        for i in range(2*nbrAxes):
            cadre.itemconfigure(ligne[2*nbrAxes+i],start=(pi-startAngle+i*pi/nbrAxes)*180/pi)
            x0r,y0r=rotatione(symetrise([x0,y0]),i)#le centre du cercle est symétrisé, puis tourne
            cadre.coords(ligne[2*nbrAxes+i],x0r-R,y0r-R,x0r+R,y0r+R)
     .........
```

Il ne nous reste plus qu'à implémenter le dessin des polygones que nous avons prévu dans notre « cahier des charges » : des triangles (équilatéraux), des carrés et des étoiles pentagonales (régulières). Cela ne devrait pas poser beaucoup de problèmes. Il s'agit juste de traduire ces figures à l'aide des coordonnées, ce qui ne nécessite pas vraiment d'explications à ce stade. Les étoiles pentagonales sont évidemment un peu moins simples à dessiner car on souhaite que les deux points du clic de souris (le 1er étant le lieu où l'on a cliqué et le 2d celui où l'on va lâcher le bouton de la souris) correspondent à deux sommets consécutifs de l'étoile (comme pour le triangle et le carré).

Les possibilités créatives de notre petit programme, prévues à notre cahier des charges, étant désormais atteintes, il ne nous reste plus qu'à régler quelques petits détails. La possibilité de gommer a été implémentée par la commande *effacer*() du menu fichier. Celle-ci se contente de supprimer tout le dessin excepté les axes que nous laissons affichés (si l'option axes est sélectionnée). Sans doute serait-il judicieux de prévoir, en plus de ce bouton « Nouveau » qui recommence tout, un autre bouton « Gommer Dernier » qui ne supprimerait que le dernier objet tracé. Comme nous avons utilisé une liste *ligne* pour tracer les différents objets (sauf les « lignes libres »), nous allons supprimer ces objets avec la nouvelle fonction *gommer*(). Nous devons donc modifier notre façon de dessiner les lignes libres pour qu'elles utilisent également la liste *ligne*. L'option de gommer le dernier objet ne peut être utilisée pour effacer tous les éléments du dessin les uns après les autres, il faudrait pour cela garder les noms de ces objets dans une autre liste. Nous avons, par contre, désactivé l'option « Gommer Dernier » quand il n'y a plus d'objet à gommer. Cette option se réactive quand on crée un objet.

La dernière fonction que nous avons à implémenter dans notre cahier des charges est le bouton « Enregistrer » qui doit générer un fichier image à partir du dessin actuel. Pour faire cela directement avec le programme, ce n'est pas très facile. Pour les besoins de ce petit fascicule, nous avons créé nos images en faisant des copies d'écran (cela met l'image dans le presse-papier, on peut alors la « coller » dans un logiciel tel que « *Paint* », la retravailler et l'enregistrer au format souhaité ou bien l'importer dans un traitement de texte). Si on veut vraiment réaliser l'enregistrement à partir du programme, la méthode la plus simple produit un fichier au format *ps* (*postscript*, format utilisé par les imprimantes) qui n'est pas un format d'image valide. La transformation du format *ps* en format *pdf* (facile à ouvrir et à imprimer) demande, au préalable, d'avoir téléchargé et installé *Ghostscript*, le programme qui va effectuer la conversion. Après installation de ce programme (on doit, sous Windows, déclarer le *path* dans les variables d'environnement pour indiquer le chemin vers les dossiers *lib* et *script* du programme), on peut implémenter notre fonction *record*() avec le code qui suit :

```
image=cadre.postscript(file="saved.ps",height=500,width=500,colormode="color")
process=Popen(["ps2pdf","saved.ps","iRosace.pdf"],shell=True)
process.wait()
remove("saved.ps")
```

Il faut tout de même importer des modules de Python mais ce sont des modules standard (déjà présents et

utilisables) : *from subprocess import Popen* et *from os import* *. La méthode *Popen* du module *subprocess* ouvre le script *ps2pdf* du programme *Ghostscript*, tandis que, du module *os*, on n'utilise que la fonction *remove* qui supprime le fichier intermédiaire *saved.ps*. À ce stade, il devrait rester le fichier *iRosace.pdf* dans le répertoire où se situe votre programme. Plus simple (à mon humble avis) que cet enregistrement en *pdf* (on voulait du *png* au départ) : enregistrer le fichier *ps* et l'ouvrir avec un logiciel comme *ImageMagick* (à télécharger et installer s'il ne se trouve pas déjà sur votre ordinateur). La conversion en un format image ne pose alors aucun problème.
Ajoutons la partie qui va permettre de créer directement le fichier à l'emplacement voulu avec un nom choisi au lieu d'un nom prédéfini ou, pire, d'un nom qui serait demandé à l'utilisateur par le biais de la console (le *shell*). Il faut ouvrir le répertoire courant de l'ordinateur et attendre que l'utilisateur saisisse un nom pour le fichier (l'extension *ps* étant ajoutée automatiquement). Cela se fait facilement grâce à la méthode *asksaveasfilename*() du sous-module *tkinter.filedialog* que l'on doit importer (*import tkinter.filedialog*). Tant que nous y sommes, pourquoi ne pas offrir la possibilité d'ouvrir un fichier image existant (en créant un sous-menu « Ouvrir » à côté du sous-menu « Enregistrer ») pour servir de fond à notre *Canvas* cadre ? Nous pourrions alors dessiner nos rosaces sur des photos, des dessins... La méthode cadre.*create_image*(xCentre,yCentre,*image*=photo) accepte des photos de différents types (*gif*, *png*) mais pas des images *ps* (ce sont pourtant les seules images crées par la méthode *postscript* de *tkinter* !) ni des images *jpeg*. La photo qui est donnée comme valeur du paramètre image est définie par l'instruction photo=*PhotoImage*(*file=tkinter.filedialog.askopenfilename*()).

Voilà notre programme terminé, ce qui clôture ce petit fascicule de questions de programmation.
PM-2015