TD n°1 d'algorithmique : Programmer un jeu sur la calculatrice

CORRECTION

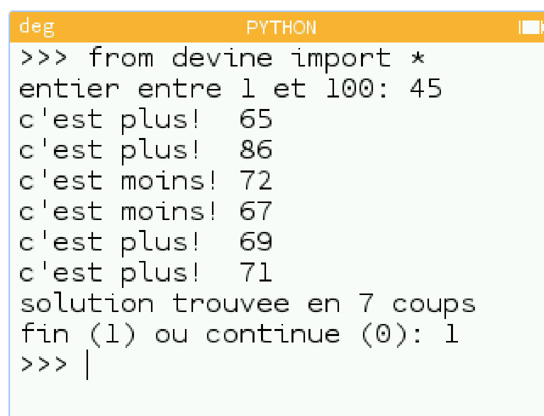
1. Devine un nombre

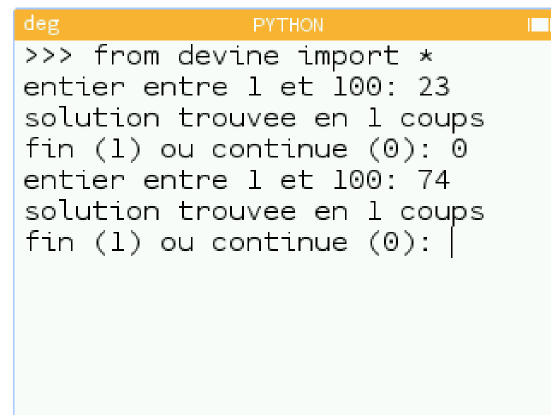
Le plus simple des jeux : la calculatrice choisit un nombre entre deux bornes, par exemple entre 1 et 100, et le joueur doit deviner ce nombre. Pour cela, il propose un nombre et la calculatrice répond « plus » ou « moins ». Elle compte le nombre d'essais et, quand le joueur a trouvé la bonne réponse, elle affiche le nombre d'essais.

⊶ Pour programmer ce jeu, il suffit d'alterner les propositions du joueur et les affichages des réponses de la calculatrice « plus » ou « moins ». Le test d'arrêt de la boucle est évident puisque le jeu s'arrête quand le joueur a trouvé le nombre.

Voici une proposition de programme qui réalise ce projet. Un problème inattendu vient de la fonction *randint* (je pense que les autres fonctions du module *random* doivent avoir le même problème) : les nombres générés entre les deux formes suivent une suite pseudo-aléatoire qui commence **toujours** par le même nombre. Quand on prend *a*=1 et *b*=100 alors la valeur proposée par *randint*(*a*,*b*) est toujours 7, c'est ennuyeux.

```
from random import *
a,b,f=1,100,0
seed(123456)#pour modifier la suite aleatoire
while f==0:
  m=int(input("entier entre {} et {}: ".format(a,b)))
  n,c=randint(a,b),1
  while m!=n:
    c+=1
    t="bravo!"
    if m>n:t="c'est moins! "
    if m<n:t="c'est plus!  "
    m=int(input(t))
  print("solution trouvee en {} coups".format(c))
  f=int(input("fin (1) ou continue (0): "))
```





Pour contourner un peu ce problème j'ai mis l'instruction *seed*(123456) qui va faire démarrer la suite aléatoire à un nombre dépendant de 123456 (pourquoi pas), mais avec cette graine le nombre proposé sera toujours 23. Si la graine est 2025, le nombre proposé sera toujours 71 (mon illustration). Normalement les générateurs de nombres aléatoires n'ont pas ce défaut car ils choisissent une graine en fonction de la date d'horloge. Pour l'instant, Numworks n'a pas encore de date d'horloge, donc il nous faut faire avec ce défaut. Vous pouvez changer la graine à chaque exécution du programme pour que le jeu soit moins prévisible. Vous pouvez aussi apprendre la suite des nombres pseudo-aléatoires que va proposer la calculatrice pour une graine donnée et montrer à votre camarade comment vous êtes fort : avec la graine 123456 on a toujours 23 et puis ensuite on a toujours 74, etc.

2. Entraînement aux tables

La calculatrice choisit deux nombres entre deux bornes, par exemple 2 et 10, et demande le produit de ces deux nombres. Si le résultat entré par le joueur est correct, elle propose un autre produit. Sinon, elle repose la question. Dans tous les cas, le nombre de bonnes réponses est enregistré ainsi que le nombre de question. On peut afficher le pourcentage de bonnes réponses avant chaque nouvelle question.

⊶ Pour programmer ce jeu, il suffit de prévoir l'interaction entre la calculatrice (qui pose les questions et affiche le score du joueur) et le joueur. La calculatrice formule sa question en affichant le produit des deux nombres choisis et attend la réponse. Cela peut se faire avec l'instruction *n*=*int*(*input*("{}x{}=".format(*a*,*b*))) qui va mettre votre réponse dans la mémoire *n* pour le traitement. Il faut prévoir un test d'arrêt pour éviter la boucle infinie : par exemple, tant que le nombre de questions est inférieur à 20 *ou* le tau de bonnes réponses inférieur à 90%.

Aucune difficulté dans ce projet non plus. Il faut créer une boucle pour générer un certain nombre de questions sur les tables. Le plus simple est de poser un nombre fixe de questions, par exemple 10. Dans ce cas, on fait une boucle *for*.

J'ai préféré écrire une boucle *while* et mettre une condition d'arrêt un peu élaborée : tant que l'on n'a pas répondu correctement à 20 questions au moins ou que le taux de bonnes réponses est inférieur à 90%. Répondre correctement à 20 questions ne suffira pas pour sortir de la boucle, il faut en plus que le taux de bonnes réponses dépasse 90%. Avec 2 mauvaises réponses sur 20, le taux est à 90%, on continue. Si on arrive à 20 bonnes réponses sur 22 alors le taux est égal à 90,9%, ce n'est pas assez car il n'y a que 20 bonnes réponses (il en faut plus). Il faut donc arriver à 21 bonnes réponses sur 23 pour obtenir le « bravo ! » final. Bien sûr, cela devient plus compliqué de finir si on a fait 5 mauvaises réponses car alors, pour atteindre le taux de 90%, il faut au moins 50 réponses...

	mauvaises réponses													
questions	0		1		2		3		4		5		6	
20	20	1	19	0,950	18,000	0,900	17,000	0,850	16,000	0,800	15,000	0,750	14,000	0,700
21	21	1	20	0,952	19,000	0,905	18,000	0,857	17,000	0,810	16,000	0,762	15,000	0,714
22	22	1	21	0,955	20,000	0,909	19,000	0,864	18,000	0,818	17,000	0,773	16,000	0,727
23	23	1	22	0,957	21,000	0,913	20,000	0,870	19,000	0,826	18,000	0,783	17,000	0,739
24	24	1	23	0,958	22,000	0,917	21,000	0,875	20,000	0,833	19,000	0,792	18,000	0,750
25	25	1	24	0,960	23,000	0,920	22,000	0,880	21,000	0,840	20,000	0,800	19,000	0,760
26	26	1	25	0,962	24,000	0,923	23,000	0,885	22,000	0,846	21,000	0,808	20,000	0,769
27	27	1	26	0,963	25,000	0,926	24,000	0,889	23,000	0,852	22,000	0,815	21,000	0,778
28	28	1	27	0,964	26,000	0,929	25,000	0,893	24,000	0,857	23,000	0,821	22,000	0,786
29	29	1	28	0,966	27,000	0,931	26,000	0,897	25,000	0,862	24,000	0,828	23,000	0,793
30	30	1	29	0,967	28,000	0,933	27,000	0,900	26,000	0,867	25,000	0,833	24,000	0,800
31	31	1	30	0,968	29,000	0,935	28,000	0,903	27,000	0,871	26,000	0,839	25,000	0,806
32	32	1	31	0,969	30,000	0,938	29,000	0,906	28,000	0,875	27,000	0,844	26,000	0,813
33	33	1	32	0,970	31,000	0,939	30,000	0,909	29,000	0,879	28,000	0,848	27,000	0,818
34	34	1	33	0,971	32,000	0,941	31,000	0,912	30,000	0,882	29,000	0,853	28,000	0,824
35	35	1	34	0,971	33,000	0,943	32,000	0,914	31,000	0,886	30,000	0,857	29,000	0,829
36	36	1	35	0,972	34,000	0,944	33,000	0,917	32,000	0,889	31,000	0,861	30,000	0,833
37	37	1	36	0,973	35,000	0,946	34,000	0,919	33,000	0,892	32,000	0,865	31,000	0,838
38	38	1	37	0,974	36,000	0,947	35,000	0,921	34,000	0,895	33,000	0,868	32,000	0,842
39	39	1	38	0,974	37,000	0,949	36,000	0,923	35,000	0,897	34,000	0,872	33,000	0,846
40	40	1	39	0,975	38,000	0,950	37,000	0,925	36,000	0,900	35,000	0,875	34,000	0,850
41	41	1	40	0,976	39,000	0,951	38,000	0,927	37,000	0,902	36,000	0,878	35,000	0,854
42	42	1	41	0,976	40,000	0,952	39,000	0,929	38,000	0,905	37,000	0,881	36,000	0,857
43	43	1	42	0,977	41,000	0,953	40,000	0,930	39,000	0,907	38,000	0,884	37,000	0,860
44	44	1	43	0,977	42,000	0,955	41,000	0,932	40,000	0,909	39,000	0,886	38,000	0,864
45	45	1	44	0,978	43,000	0,956	42,000	0,933	41,000	0,911	40,000	0,889	39,000	0,867
46	46	1	45	0,978	44,000	0,957	43,000	0,935	42,000	0,913	41,000	0,891	40,000	0,870
47	47	1	46	0,979	45,000	0,957	44,000	0,936	43,000	0,915	42,000	0,894	41,000	0,872
48	48	1	47	0,979	46,000	0,958	45,000	0,938	44,000	0,917	43,000	0,896	42,000	0,875
49	49	1	48	0,980	47,000	0,959	46,000	0,939	45,000	0,918	44,000	0,898	43,000	0,878
50	50	1	49	0,980	48,000	0,960	47,000	0,940	46,000	0,920	45,000	0,900	44,000	0,880
51	51	1	50	0,980	49,000	0,961	48,000	0,941	47,000	0,922	46,000	0,902	45,000	0,882

```python
from random import *
def tables(s=12345,m=2,M=10,t=0.9):
  seed(s)
  br,mr,tx=0,0,0
  while tx<=t or br<=20:
    a=randint(m,M)
    b=randint(m,M)
    c=a*b
    n=int(input("{}x{}=".format(a,b)))
    if n==c:
      br+=1
      print("Oui!")
    else:
      mr+=1
      print("NON ({}x{}={})".format(a,b,c))
    tx=br/(br+mr)
  print("{} bonnes reponses sur {}".format(br,br+mr))
```

```
deg                PYTHON
>>> from tables import *
>>> tables(789456)
8x8=64
Oui!
3x5=15
Oui!
4x7=28
Oui!
5x4=21
NON (5x4=20)
8x4=
 ....
5x10=0
NON (5x10=50)
8x8=64
Oui!
7x7=49
Oui!
21 bonnes reponses sur 23
>>>
```

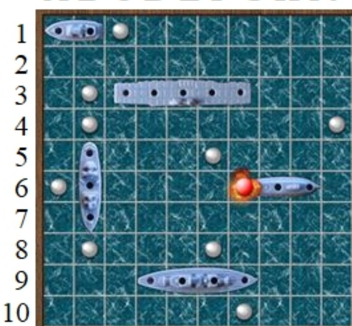
Voici donc ma proposition de programme : si je fais 2 erreurs il me pose 23 questions.
En donnant cette solution, je n'ai pas tenu compte d'une contrainte de l'exercice : reposer la même question tant que l'utilisateur fait une erreur. J'avais privilégié un aspect de l'apprentissage qui consiste à donner la correction (en espérant que celle-ci serve à l'apprenant).
Je vais maintenant implémenter la contrainte de l'énoncé, en évitant de comptabiliser les répétitions comme des nouvelles questions, en ne donnant pas la réponse. Je vais aussi éviter de comptabiliser la bonne réponse donnée dans les répétitions comme une bonne réponse (à la base, il s'agit d'une réponse erronée que l'apprenant à fini par corriger lui-même). On pouvait prendre d'autres options par rapport à cette contrainte. La fonction *tables2*() ci-dessous donne le programme final. Remarquez au passage l'utilisation des arguments par défaut qui sont fournis à la fonction dans les parenthèses : s=12345,m=2,M=10,t=0.9. Dans cette parenthèse, on donne la graine du générateur aléatoire (s) qu'il faut changer si on ne veut pas avoir toujours la même série de questions. Comme il s'agit d'une variable disposant d'une valeur par défaut (12345), on peut lancer le programme avec l'instruction *tables2*(). Cela suffit ; dans ce cas les variables s, m, M et t prendront les valeurs par défaut. Généralement, on lancera ce programme en donnant une nouvelle graine, par exemple on écrira *tables2*(1789). Si on veut interroger l'apprenant avec des questions plus difficiles, on peut écrire *tables2*(2018,3,12). On pourra ainsi avoir des questions sur les tables de 11 et 12... Si on veut seulement changer le taux limite, il faut donner les valeurs des autres variables car elles sont situées avant (on peut aussi modifier l'ordre des variables dans les parenthèses). Par exemple, pour on tapera *tables2*(2018,3,12,0.85) pour mettre le taux à 85% et, accessoirement, interroger sur des multiplicateurs compris entre 3 et 12.

```
from random import *
def tables2(s=12345,m=2,M=10,t=0.9):
  seed(s)
  br,mr,tx=0,0,0
  while tx<=t or br<=20:
    a=randint(m,M)
    b=randint(m,M)
    c=a*b
    n=int(input("{}x{}=".format(a,b)))
    if n==c:
      br+=1
      print("Oui!")
    else:
      mr+=1
      print("NON! essaie de nouveau...")
      while n!=c:
        n=int(input("{}x{}=".format(a,b)))
      print("Bravo! on continue...")
    tx=br/(br+mr)
  print("{} bonnes reponses sur {}".format(br,br+mr))
```

```
deg                PYTHON
>>> from tables import *
>>> tables2(456)
6x4=24
Oui!
4x6=25
NON! essaie de nouveau...
4x6=24
Bravo! on continue...
8x6=48
Oui!
2x5=
 ...
9x4=36
Oui!
6x10=60
Oui!
28 bonnes reponses sur 31
>>>
```

3. Bataille navale

La calculatrice choisit des emplacements pour 5 bateaux dont les longueurs sont connues par le joueur (2, 3, 3, 4 et 5) et le joueur propose successivement le lieu à bombarder repéré par sa colonne (entre A et J) et sa ligne (entre 1 et 10). La calculatrice répond en affichant un point rouge si un bateau est touché, un point blanc si le bombardement a raté les cibles. Lorsqu'un bateau est coulé (tous ses points sont touchés), la calculatrice doit le signaler, par exemple en entourant les points touchés par un rectangle. Le but du jeu est de couler tous les bateaux ennemis.



⊶Pour programmer ce jeu, il faut prévoir un affichage graphique (module kandinsky) ; les saisies du joueur sont des séquences de 2 caractères comme B7 ou H2 et les réponses de la calculatrice sont données par l'affichage qui est mis à jour. Au départ, il faut dessiner une grille 10×10 avec les indications des lettres et chiffres. Je rappelle que l'écran de la Numworks est composé de 320 pixels sur sa longueur et de 222 pixels sur sa hauteur. On peut fixer la couleur de chacun des pixels en exécutant l'instruction set_pixel(x,y,color), où color est une couleur du système RGB. Par exemple, c1=color(255,255,255) fixe la couleur c1 à blanc et c2=color(255,0,0) fixe la couleur c2 à rouge.

Comme dirait Napoléon, ça se corse, car ici, on va utiliser le module kandinsky pour obtenir l'interface graphique. Dans un premier temps cependant on peut se focaliser sur les autres difficultés du programme : le choix de l'emplacement des bateaux et les interactions avec l'utilisateur. Les échanges avec le programme se feront uniquement dans la console (comme pour les programmes précédent). Il est toujours possible de vérifier sur le papier que les valeurs sont correctes. Dans un deuxième temps, lorsque tout le reste est au point, on cherchera à afficher la grille, à marquer les coups du joueur et à donner d'autres informations (« touché », « raté » ou « coulé », le nombre de coups, la case visée). La difficulté de passer en mode graphique vient du fait que l'on n'a plus accès aux informations de la console (données par *print* ou *input*). Même les messages d'erreur ne sont pas visibles et c'est ennuyeux lorsqu'on est dans le workshop de Numworks (lorsqu'on programme sur son ordinateur via internet) car on reste alors bloqué en mode graphique sans possibilité d'accéder au message qui est sur la console (cela n'arrive pas sur la calculatrice qui bascule plus facilement sur la console avec la flèche « retour »).
Dans le programme qui suit, la première colonne concerne le choix de l'emplacement des bateaux, la seconde les échanges avec le joueur. J'ai ajouté des traits verticaux pour que l'on comprenne plus facilement les indentations. J'ai donc utilisé 5 listes différentes (B, E, T, X et plus loin K) : B contient le nombre et la longueur des bateaux (cela pourrait être changé), est la liste principale qui contiendra les emplacements des bateaux, avec une liste pour chaque bateau (regardez la sortie du programme, encadrée en bleu : elle affiche cette liste E choisie. On voit que le 1er bateau correspond aux emplacements I3 et I4). Dans la liste T, j'ai mis les emplacements occupés par les bateaux (sans regrouper les emplacements de chaque bateau dans des listes), dans la liste X, je vais écrire si le bateau de même rang de la liste T a été touché (1) ou non (0). De cette façon, je n'ai qu'à tester la somme des valeurs de la liste X pour savoir si le jeu est fini. On peut sans doute simplifier ce programme et se passer de certaines de ces listes.

Pour les emplacements, il faut décider aléatoirement si le bateau est vertical (r=1) ou horizontal (r=0) ; ils doivent aussi bien sûr éviter que les bateaux créés ne se chevauchent (if K[j] in T)... Tant que la variable *good* reste à *False*, le bateau qui est en train d'être implanté ne vérifie pas ces contraintes.

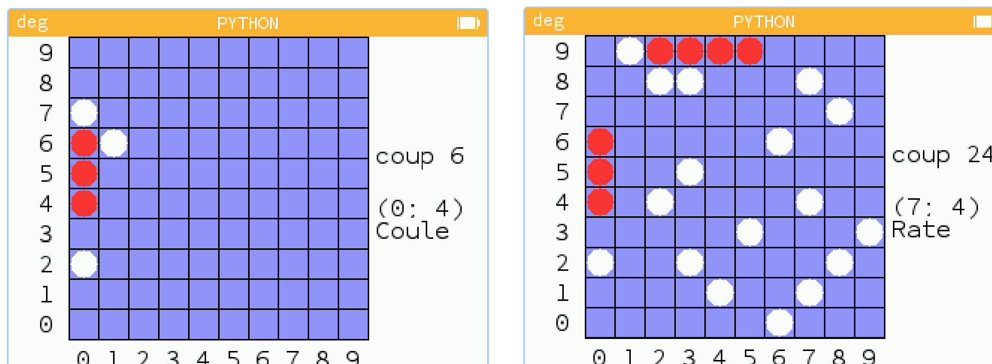
```
from random import *
from math import *
B=[2,3,3,4,5]
E=len(B)*[[]]
T=[]
X=sum(B)*[0]
#choix des emplacements
for i in range(len(B)):
  r=randint(0,1)#0:horiz.,1:vert.
  good=False
  while good==False:
    K=B[i]*[0]
    if r==0:
      c=randint(0,10-B[i])
      l=randint(0,9)
    else:
      c=randint(0,9)
      l=randint(0,10-B[i])
    good=True
    for j in range(B[i]):
      if r==0:K[j]=str(c+j)+str(l)
      else:K[j]=str(c)+str(l+j)
      if K[j] in T:good=False
  E[i]=K
  for j in range(B[i]):T.append(K[j])
print(E)
```

```
def entrer():
  c=input()
  if len(c)==2:return c
  else:entrer()
```

```
cp=0
while sum(X)!=len(X):
  coord=entrer()
  cp+=1
  col=int(coord[0])#colonne(0-9)
  lig=int(coord[1])#ligne  (0-9)
  txt="Rate"
  if coord in T:
    rg=T.index(coord)
    X[rg]=1
    txt="Touche"
    for i in range(len(E)):
      if coord in E[i]:
        tch=0
        for j in range(len(E[i])):
          rg=T.index(E[i][j])
          if X[rg]==1:tch+=1
        if tch==len(E[i]):
          txt="Coule"
  print(txt)
print("Mission accomplie en {} coups".format(cp))
```

```
deg                PYTHON
>>> from bataille_navale impor
[['83', '84'], ['21', '22', '23'], ['04', '05', '06'],
 ['29', '39', '49', '59'], ['37', '47', '57', '67', '77']]
83
Touche
84
Coule
85
Rate
```

La partie concernant le jeu est réduite : il suffit de traduire les deux chiffres entrés en un rang de colonne (col) et un rang de ligne (lig) ; ensuite, on examine si la case est occupée (if coord in T) et, si oui, on met à jour la liste X (X[rg]=1) et on examine ensuite toutes les cases correspondant au bateau touché pour savoir si le bateau est coulé ou non.



Ces traitements sont un peu compliqués ? Oui, je le reconnais mais on peut sans doute simplifier. Passons maintenant au traitement graphique. Il faut tracer la grille, inscrire les numéros des lignes et des colonnes. Il faut une fonction pour tracer des cercles rouges (touché ou coulé) ou blancs (raté) et une pour inscrire les messages (le nombre de coups, le résultat du tir et sa localisation). Le programme complet est alors deux fois plus long, mais il est dix fois plus performant. Pour pouvoir jouer sur le workshop avec le module kandinsky, j'ai fini par comprendre qu'il fallait englober le programme dans une fonction (d'où le *def bn*() dans ce qui suit). Je peux donc faire des copies d'écran pendant une phase du jeu pour montrer l'affichage et je peux aussi tester le programme directement dans le workshop sans être obligé de l'envoyer à chaque retouche sur la calculatrice. La dernière chose à ajouter est l'instruction *seed*(*s*) qui change la graine selon la valeur de *s*.

```python
from random import *
from math import *
from kandinsky import *
noir=color(0,0,0)
bleu=color(150,150,255)
blanc=color(255,255,255)
rouge=color(255,55,55)
def segmentH(x1,x2,y):
  for i in range(x2-x1):
    set_pixel(x1+i,y,noir)

def segmentV(x,y1,y2):
  for i in range(y2-y1):
    set_pixel(x,y1+i,noir)

def carre(x,y,c,l):
  for i in range(l):
    for j in range(l):
      set_pixel(x+i,y+j,c)

def plateau():
  for i in range(10):
    draw_string(str(i),45+i*20,205)
    draw_string(str(9-i),20,2+i*20)
  carre(40,0,bleu,200)
  for i in range(11):
    segmentH(40,240,0+i*20)
    segmentV(40+i*20,0,200)
  set_pixel(240,200,noir)#dernier pt

def cercle(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def message(t,c,m):
  carre(245,70,blanc,70)
  draw_string("coup "+c,245,70)
  draw_string("("+t[0]+"; "+t[1]+")",245,105)
  draw_string(m,245,120)

def entrer():
  c=input()
  if len(c)==2:return c
  else:entrer()
```

```python
def bn(s):
  seed(s)
  B=[2,3,3,4,5]
  E=len(B)*[[]]
  T=[]
  X=sum(B)*[0]
  plateau()
  #choix des emplacements
  for i in range(len(B)):
    r=randint(0,1)#0:horiz.,1:vert.
    good=False
    while good==False:
      K=B[i]*[0]
      if r==0:
        c=randint(0,10-B[i])
        l=randint(0,9)
      else:
        c=randint(0,9)
        l=randint(0,10-B[i])
      good=True
      for j in range(B[i]):
        if r==0:K[j]=str(c+j)+str(l)
        else:K[j]=str(c)+str(l+j)
        if K[j] in T:good=False
    E[i]=K
    for j in range(B[i]):T.append(K[j])
  print(E)
  #jeu
  cp=0
  while sum(X)!=len(X):
    coord=entrer()
    cp+=1
    col=int(coord[0])#colonne(0-9)
    lig=int(coord[1])#ligne  (0-9)
    txt="Rate"
    if coord in T:
      coul=rouge
      rg=T.index(coord)
      X[rg]=1
      txt="Touche"
      for i in range(len(E)):
        if coord in E[i]:
          tch=0
          for j in range(len(E[i])):
            rg=T.index(E[i][j])
            if X[rg]==1:tch+=1
          if tch==len(E[i]):
            txt="Coule"
    else: coul=blanc
    message(coord,str(cp),txt)
    cercle(50+20*col,190-20*lig,9,coul,9)
    print(txt)
  draw_string("Bravo!",245,180)
  print("Mission accomplie en {} coups".format(cp))
```

4. <u>Master mind</u>

La calculatrice choisit une combinaison de couleurs pour 4 emplacements, les couleurs étant choisies parmi 6 couleurs possibles : jaune, bleu, rouge, vert, blanc et noir. Des variantes plus difficiles existent (possibilité d'ajouter des couleurs, d'utiliser des espaces vides comme couleur, ajout d'un 5ème emplacement, etc .)

À chaque tour, le joueur propose une combinaison de pions de couleur et la calculatrice lui indique :

- le nombre de pions de la *bonne couleur bien placés* en utilisant le

même nombre de *fiches noires*

- le nombre de pions de la *bonne couleur*, mais *mal placés*, avec les *fiches blanches*

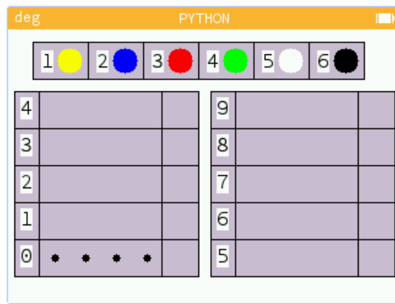
Les tours successifs (dix coups maximum) restent affichés de manière à pouvoir servir de base à la réflexion du joueur, le but étant de trouver la combinaison en un minimum de coups.

🗝 Pour programmer ce jeu, il faut donc prévoir un affichage graphique (module kandinsky) ; les saisies du joueur doivent être codifiées, par exemple avec des chiffres : jaune=1, bleu=2, rouge=3, vert=4, blanc=5 et noir=6. Le premier coup du joueur de l'illustration est donc 3245. La calculatrice doit afficher des disques colorés aux bons emplacements ainsi que les fiches (noires ou blanches) sur le côté. Prévoir donc une fonction *disque*(c,x,y) qui affiche un disque de la couleur c centré en un point de coordonnées (x;y).

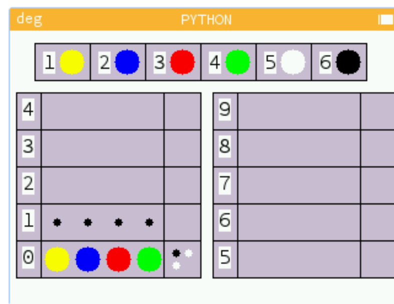
Une fois que l'on a compris le principe pour entrer les propositions (un input qui contrôle quelques informations comme la longueur de la chaîne et la valeur numérique des caractères entrés, ici entre 1 et 6) et que l'on dispose de méthodes pour afficher un cercle et pour tracer des segments, il ne reste plus qu'à mettre cela en forme. La disposition doit être étudiée pour qu'on puisse entrer 10 propositions, les laisser afficher ainsi que les corrections (les fiches noires et blanches). J'ai choisi de disposer cela sur deux carrés pour occuper toute la place tout en conservant la « verticalité » des entrées. J'ai ajouté aussi une légende assez claire qui permet de traduire les chiffres en codes couleurs pour éviter la difficulté supplémentaire au joueur de se rappeler ces codes.

```
from random import *
from kandinsky import *
from math import *

def segH(x1,x2,y):
  for i in range(x2-x1):
    set_pixel(x1+i,y,color(0,0,0))

def segV(x,y1,y2):
  for i in range(y2-y1):
    set_pixel(x,y1+i,color(0,0,0))

def carre(x,y,c,l):
  for i in range(l):
    for j in range(l):
      set_pixel(x+i,y+j,c)

def cercle(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def couleur(c):
  cl=[color(255,255,0),color(0,0,255),color(255,0,0),color(0,255,0),
  color(255,255,255),color(0,0,0)]
  return cl[c]

def plateau():
  carre(5,50,color(200,190,210),150)
  carre(165,50,color(200,190,210),150)
  for i in range(5):
    draw_string(str(i),10,175-i*30)
    draw_string(str(i+5),170,175-i*30)
  for i in range(6):
    segH(5,155,50+i*30)
    segH(165,315,50+i*30)
  for i in range(2):
    segV(5+i*160,50,200)
    segV(25+i*160,50,200)
    segV(125+i*160,50,200)
    segV(155+i*160,50,200)
    set_pixel(155+i*160,200,color(0,0,0))#dernier pt
  for i in range(270):
    for j in range(31):
      set_pixel(20+i,10+j,color(200,190,210))
  for i in range(6):
    segH(20+45*i,65+45*i,10)
    segH(20+45*i,65+45*i,40)
    segV(20+45*i,10,40)
    segV(65+45*i,10,40)
    set_pixel(290,40,color(0,0,0))
    draw_string(str(i+1),27+45*i,17)
    cercle(50+45*i,25,10,couleur(i),10)
    prepare(0)

def prepare(c):
  for i in range(4):
    cercle(38+25*i+160*(c//5),185-30*(c%5),3,color(0,0,0),3)

def dessine(ch,c):
  for i in range(4):
    cercle(38+25*i+160*(c//5),185-30*(c%5),10,couleur(int(ch[i])-1),10)

def evalue(r,c):
  b=0
  for i in range(4):
    if r[i]>0:
      cercle(135+10*(b%2)+160*((c-1)//5),180-30*((c-1)%5)+10*(b//2),3,couleur(3+r[i]),3)
      b+=1

def entrer():
  c=''
  while (len(c)==4 and 0<int(c[0])<7
   and 0<int(c[1])<7 and 0<int(c[2])<7
   and 0<int(c[3])<7)==False: c=input()
  return c

  def mm(s):
    seed(s)
    plateau()
    combi,R="",4*[0]
    for i in range(4):
      combi+=str(randint(0,5))
    stock=combi
    cp=0
    while sum(R)!=8 and cp<10:
      R=4*[0]#resultat(0:mc,1:bcmp,2bcbp)
      choix=entrer()
      dessine(choix,cp)
      print(cp,choix,end=" - ")
      cp+=1
      bc=0
      #test bcbp
      for i in range(4):
        if choix[i]==combi[i]:
          R[bc]+=2
          bc+=1
          combi=combi[0:i]+"6"+combi[i+1:4]
          choix=choix[0:i]+"6"+choix[i+1:4]
      #test bcmp
      for i in range(4):
        for j in range(4):
          if choix[i]==combi[j] and choix[i]!="6":
            R[bc]+=1
            bc+=1
            combi=combi[0:j]+"6"+combi[j+1:4]
            choix=choix[0:i]+"6"+choix[i+1:4]
      evalue(R,cp)
      if cp<10 and sum(R)!=8:
        prepare(cp)
      for i in range(4):
        print(R[i],end="")
      print("")
      combi=stock
    if cp<10:
      draw_string("Bravo!",140,205)
      print("Solution trouvee en {} coups".format(cp))
    else:
      draw_string("Echec!",140,205)
      print("Il fallait trouver {}".format(combi))
```

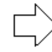
Le jeu ainsi créé est fonctionnel :

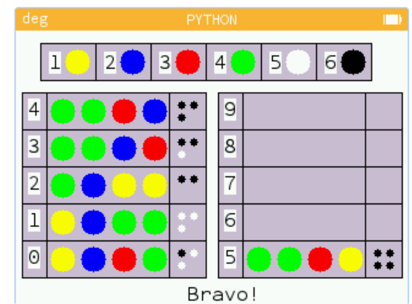


après avoir entré mm(12)



après avoir entré 1 2 3 4 EXE





et après un peu de chance et beaucoup de cogitation

le seul problème, me semble t-il, c'est la difficulté persistante à passer du code couleur aux chiffres. Il faudrait concevoir l'affichage autrement, en rappelant les chiffres entrés, par exemple dans la colonne de gauche. Mais un caractère affiché occupe toujours un rectangle de 10×20, il faudrait disposer de 40 pixels horizontaux (actuellement la colonne de gauche occupe 20 pixels horizontaux).