TME 1 : intro python

Prise en main python

Python est un langage script : toute commande est passée à un interpréteur python qui la traduit en langage bas-niveau. Il existe deux manières prépondérantes pour exécuter du code python, soit par l'exécution de l'interpréteur sur un fichier contenant les commandes à exécuter (le programme, suffixé par convention par `*.py` : `python monprogramme.py`, soit par l'intermédiaire d'une "console" où l'on intéragit directement avec l'interpréteur. La console classique de python (commande 'python' dans un terminal) n'est pas pratique du tout, on préfère utiliser en général `ipython` ou un IDE (`spyder`, `pycharm` par exemple).

Opération élémentaire

```
# Affectation d une variable
a = 3
# operations usuelles
(1 + 2. - 3.5), (3 * 4 /2 ),  4**2
# Attention ! reels et entiers
1/2, 1./2
# Operations logiques
True and False or (not False) == 2>1
# chaines de caracteres
s = "abcde"
s = s + s # concatenation
# afficher un resultat
print(1+1-2,s+s)
```

Structures

N-uplet (`tuple`) : une liste d'élements ordonnés, de longueur fixe, non mutable : aucun élement ne peut être changé apres la creation du n-uplet

```
c = (1,2,3) # creation d un n-uplet
c[0],c[1]  # acces aux elements d un couple,
c + c  # concatenation de deux n-uplet
len(c) # nombre d element du n-uplet
a, b, c = 1, 2, 3 # affectation d un n-uplet de variables

s = set()  # creation d un ensemble
s = {1 ,2 ,1}
print(len(s)) #taille d un ensemble
s.add('s')   # ajout d un element
s.remove('s') # enlever un element
s.intersection({1,2,3,4})
s.union({1,2,3,4})
```

Listes : structure très importante en python. Il n'y a pas de tableau, que des listes (et des dictionnaires)

```
l = list() # creation liste vide
l1 = [ 1, 2 ,3 ] # creation d une liste avec elements
l = l + [4, 5] #concatenation
zip(l1,l2) : liste des couples
```

```
len(l) #longueur
l.append(6)     # ajout d un element
l[3]     #acces au 4-eme element
l[1:4]   # sous-liste des elements 1,2,3
l[-1],l[-2] # dernier element, avant-dernier element
sum(l)     # somme des elements d une liste
sorted(l)     #trier la liste
l = [1, "deux", 3] # une liste compposee
sub_list1 = [ x for x in l1 if x < 2] # liste comprehension
sub_list2 = [ x + 1 for x in l1 ] # liste comprehension 2
sub_list3 = [x+y for x,y in zip(l1,l1)] # liste comprehension 3
```

Dictionnaires : listes indexées par des objets (hashmap), très utilisés également. Ils permettent de stocker des couples (clé,valeur), et d'accéder aux valeurs a partir des clés (en temps efficient).

```
d = dict() # creation d un dictionnaire
d['a']=1   # presque tout type d objet peut etre
d['b']=2   # utilise comme cle, tout type d objet
d[2]= 'c'  # comme valeur
d.keys()   # liste des cles du dictionnaire
d.values() # liste des valeurs contenues dans le dictionnaire
d.items() # liste des couples (cle,valeur)
len(d)    #nombre d elements d un dictionnaire
d = dict([ ('a',1), ('b',2), (2, 'c')])  # autre methode pour creer un dictionnaire
d = { 'a':1, 'b':2, 2:'c'}  # ou bien...
d = dict( zip(['a','b',2],[1,2,'c']))  #et egalement...
d.update({'d':4,'e':5}) # "concatenation" de deux dictionnaires
```

Boucles, conditions, fonctions

Attention, en python toute la syntaxe est dans l'indentation : un bloc est formé d'un ensemble d'instructions ayant la même indentation (même nombre d'espaces précédent le premier caractère).

```
i=0
s=0
while i<10:  # boucle while
    i+=1     #indentation pour marquer ce qui fait parti de la boucle
    s+=i
s=0
for i in [1, 2, 3]: #boucle for
    j = 0          # indentation pour le for
    while j<i:      # boucle while
        j+=1        # deuxieme indentation pour le bloc while
        s = i + j
    s = s + s # retour a la premiere indentation, instruction du bloc for

def increment(x):   # definition d une fonction par le mot-cle def
    return x+1      # retour de la fonction

y=increment(5)      # appel de la fonction

def somme_soustraction(x,y=2):
    # possibilite de donner une valeur par defaut aux parametres
    return x+y,x-y             #  possibilite de retourner
                   #un n-uplet de valeurs,
                               # equivalent a (x+y,x-y)
xsom,xsub = somme_soustraction(10,5) #ou
res = somme_soustraction(10,5)
xsom == res[0],res[1]
```

Fichiers
Très simple en python :

```
##Lire
f=open("/dev/null","r")
print(f.readline())
f.close()

#ou plus simplement
with open("/dev/null","r") as f :
    for l in f:
        print l

## Ecrire
f=open("/dev/null","w")
f.write("toto\n")
f.close()

#ou
with open("/dev/null","w") as f:
    for i in range(10):
        f.write(str(i))
```

1 Exercices de prise en main

Exercice 1 – Tirage de cartes

Une carte sera représentée par un couple `(num,coul)`, avec `num` un entier entre 1 et 13 (les cartes de valeurs 1 à 9, puis 11 pour valet, 12 pour la dame, 13 pour le roi) et `coul` un caractère parmi `["C","K","P","T"]` pour cœur, carreau, pique, trèfle.
Charger le module `random`. Deux fonctions dans ce module vont nous servir dans ce tp : `random.randint(low,high)` qui permet d'engendrer un entier entre `low` et `high` non inclus, et `random.shuffle(l)` qui permet de mélanger aléatoirement et uniformément une liste (question bonus : comment faire très simplement un shuffle à partir de randint ?).

Q 1.1 Donner la fonction `paquet()` qui permet de renvoyer la liste des cartes d'un jeu à 52 cartes dans un ordre aléatoire.

Q 1.2 Faites une fonction `meme_position(p,q)` pour deux paquets `p` et `q` qui renvoie la liste des positions `i` telle que la i-ème carte du paquet `p` soit la même que celle du paquet `q`.

Q 1.3 Expérience : on dispose de deux paquets de carte que l'on mélange aléatoirement. On souhaite évaluer expérimentalement la probabilité que les deux cartes à la position `i` des deux paquets sont identiques. Calculer également la moyenne du nombre de cartes identiques à la même position lors d'une expérience. Quel résultat vous attendez ?

Q 1.4 Tracez l'évolution de la moyenne en fonction du nombre d'expériences. Pour cela, chargez le module `pyplot` et utilisez la commande `plot` :

2 Simulateur de Babyfoot

Le code fournit permet de simuler de manière très simpliste un jeu de babyfoot afin de calculer les positions optimales de tir et de défense :

- seul les défenseurs adverses sont représentés (4 rangées de 3 joueurs, 5 joueurs, 2 joueurs et 1 gardien) ;
- on considère que l'on tire toujours avec un de nos défenseurs ;

- les joueurs ne bougent pas, la configuration est fixée avant un tir ;
- la trajectoire de la balle est rectiligne, sans rebond, sans effet (la vitesse n'a donc pas d'importance) ;
- le terrain est **discrétisé** : seul des coordonnées entières sont considérées pour la position des joueurs et de la balle ;
- les constantes au début définissent la taille du terrain (`LARGEUR, LONGUEUR`), la position des rangées de joueurs (`COORDS_X`), l'écart entre les joueurs d'une même rangée (`ECARTS`), le nombre de joueurs par rangée (`NB_JOUEURS`), les dimensions du but et sa position (`LARGEUR_BUT, BUT_X, BUT_Y`), l'abscisse du tireur (`TIREUR_X`), le rayon de la balle (`RAYON_BALL`) considérée comme carré, le rayon du joueur (`RAYON_JOUEUR`) considéré également carré ;
- chaque rangée de joueurs en défense peut être positionnée en ordonnée entre `MIN_Y` et `MAX_Y` ;
- le joueur en attaque peut être positionné n'importe où en ordonnée ;
- l'objectif est bien évidement de marquer un but.

Les fonctions suivantes vous sont fournis :

- `get_config(l)` : à partir d'une liste de 4 entiers qui indiquent respectivement la position des 4 rangées de défenseurs, la fonction renvoie la position de tous les défenseurs sous la forme d'une liste de couples d'entiers, ce qu'on appellera par la suite une configuration ;
- `test_collision(coord,conf)` : cette fonction prend en paramètres un couple de coordonnées (de la balle) et une configuration ; elle renvoie vrai si il y a collision entre un joueur et la balle, faux sinon ;
- `get_tir()` : exemple de fonction qui renvoie un tir sous la forme d'un couple d'entiers indiquant l'ordonnée de l'origine du tir et l'ordonnée visée par le tir. Il n'y a pas besoin d'indiquer les abscisses, l'abscisse d'origine étant constante et l'abscisse de destination étant celle du but adverse. Pour que le tir ait une utilité, il faut bien sûr que l'ordonnée visée soit dans l'intervalle du but, i.e. entre `BUT_Y` et `BUT_Y+LARGUEUR_BUT` ;
- `simu_tir(tir,conf)` : à partir d'un tir et d'une configuration, cette fonction simule un tir pas à pas. A chaque étape, elle teste s'il y a eu une collision ou si la balle sort du terrain. Cette fonction renvoie un couple formé d'une liste de coordonnées (la trajectoire du tir) et un booléen indiquant s'il y a eu but ou non ;
- `dessine_baby(conf, traj)` : permet de renvoyer une matrice décrivant l'état du jeu à partir d'une configuration et (en option) une trajectoire. Elle peut être affichée avec la commande : `plt.imshow(img,extent = [0,LARGEUR,0,LONGUEUR],origin="lower")`

Nous noterons dans la suite B l'événement "marquer un but", $A = i$ l'événement l'attaquant est en position i et $D = \text{conf}$ l'événement la défense est en configuration conf.

Q 1.5 Combien de positions possibles pour l'attaquant ? Combien pour la défense ?

Q 1.6 Fixer la défense en une configuration c donnée et l'attaquant à une position a. Quelle type d'hypothèse avez-vous besoin de faire pour calculer $P(B|D = c, A = a)$? Calculer cette probabilité en faisant l'hypothèse d'uniformité. Choisissez une autre configuration et calculer la même probabilité.

Q 1.7 Même question pour calculer cette fois $P(B|D = c)$.

Q 1.8 Calculer la probabilité de marquer si les défenseurs sont positionnées aléatoirement uniformément et que le tireur tire toujours tout droit à partir du milieu des ordonnées.

Q 1.9 Proposer une expérience pour trouver les meilleures positions de défense.