Chapitre 5

Le jeu de Juniper Green

5.1. Nouveaux thèmes abordés dans ce chapitre

- objets
- classes
- méthodes
- listes
- opération sur les listes
- tuples

5.2. Règles du jeu

Le jeu a été créé par Richard **Porteous**, enseignant à l'école de Juniper Green, auquel il doit son nom. Il s'est réellement fait connaître grâce à Ian **Stewart**, qui en décrit les règles dans la revue *Pour la Science*, dans le numéro de juillet 1997.

Ce jeu comporte quatre règles :

1. Le joueur 1 choisit un nombre entre 1 et *Nmax*.
2. À tour de rôle, chaque joueur doit choisir un nombre parmi les multiples ou les diviseurs du nombre choisi précédemment par son adversaire et inférieur à *Nmax*.
3. Un nombre ne peut être joué qu'une seule fois.
4. Le premier nombre choisi doit être pair.

Le perdant est le joueur qui ne trouve plus de multiples ou de diviseurs communs au nombre choisi par l'adversaire.

Exemple de partie

```
Jouons avec des nombres entre 1 et 20.
Je choisis comme nombre de départ 2
Nombres valides: [1, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Que jouez-vous ? 4
Nombres valides: [1, 8, 12, 16, 20]
Je joue 8
Nombres valides: [1, 16]
Que jouez-vous ? 16
Nombres valides: [1]
Je joue 1
Nombres valides: [3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20]
Que jouez-vous ? 11
Bravo!
```

Remarque : les nombres en gras ont été entrés au clavier par le joueur.

5.3. Les objets en Python (premier contact)

En Python, tout est objet ! On rencontre souvent cette phrase dans les livres sur Python et cela en dit long sur l'importance de ce concept...

Qu'est-ce qu'un objet ? Dans un premier temps, nous allons dire que c'est une structure de données qui contient des fonctions permettant de la manipuler. On appelle ces fonctions des méthodes.

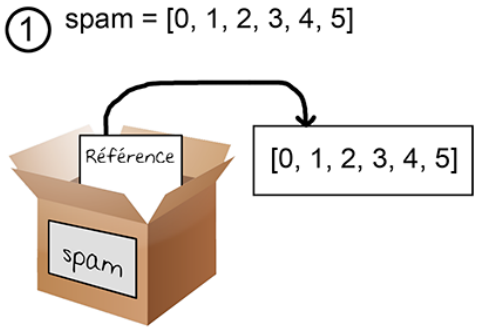
Comme l'idée de cet ouvrage est de partir d'un programme pour découvrir de nouveaux concepts, nous allons prendre comme premier exemple les listes.

Les listes forment une classe (la classe 'list'). Un objet est issu d'une classe. Tous les objets de cette classe se ressembleront, mais ils auront leur existence propre. Par exemple, toutes les listes auront le même aspect (voir § 5.4), mais elles n'auront pas le même contenu.

Autre exemple de classe venant de la vie courante : une Peugeot 3008 et une Ferrari Testarossa sont des objets de la classe 'voiture'.

5.4. Les listes

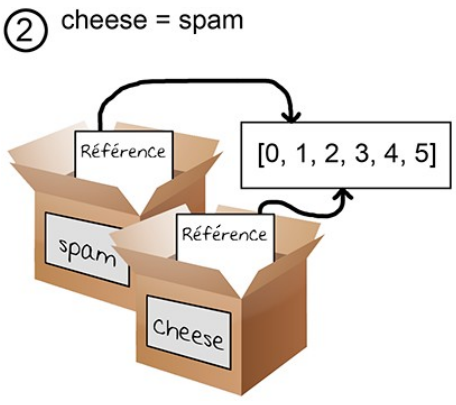
Comme son nom l'indique, une liste est une suite ordonnée de valeurs, qui peuvent être de types différents. Les éléments sont listés entre deux crochets et séparés par des virgules. On peut aussi voir une liste comme un tableau unidimensionnel où les éléments sont repérées par un indice. Ce dernier donne la position de l'élément dans la liste. Attention ! Comme dans beaucoup de langages, l'indice du premier élément est 0 et non pas 1.



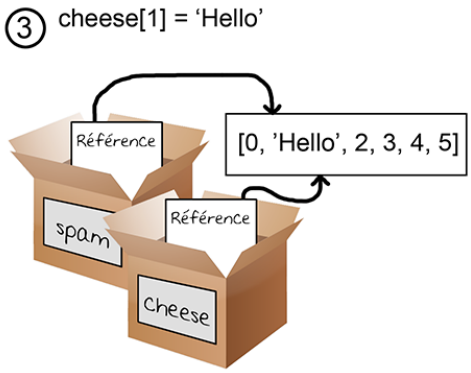
Dans le chapitre 2, nous avions représenté une variable par une boîte contenant une valeur. Dans le cas d'une liste, la boîte contient en fait une référence vers une liste, et non pas la liste elle-même. Cela a une grande importance lorsque l'on veut copier une liste. Si l'on se contente d'écrire

```
cheese = spam
```

on n'aura pas copié la liste, mais la référence vers cette liste, comme indiqué sur le schéma ci-dessous :



Cela implique que si l'on modifie une des deux listes, l'autre sera modifiée de la même façon (voir ci-dessous).



5.4.1. Création des listes

Si l'on tape `type(spam)`, on obtiendra la réponse `<class 'list'>`

Pour définir une liste vide, l'instruction est logiquement :

```
spam = []
```

On peut évidemment directement créer une liste avec des éléments. Par exemple :

```
spam = [2,3,5,7,11]
```

On peut faire des listes de listes, par exemple pour représenter un tableau bidimensionnel.

Les éléments de la liste peuvent être de types différents (ici, l'ordre, un entier, une chaîne de caractères, un réel, un caractère et une liste) :

```
spam = [3,"salut",3.1415,"x",[1,2,3]]
```

Si tous les éléments de la liste doivent avoir la même valeur au départ, on peut multiplier la valeur désirée par le nombre d'éléments que l'on veut. Si on veut une liste de 10 zéros, on écrira :

```
spam = [0]*10
```

ce qui donnera la liste `[0,0,0,0,0,0,0,0,0,0]`.

La fonction `range()` génère par défaut une séquence de nombres entiers de valeurs croissantes, et différant d'une unité. Si vous appelez `range()` avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, en commençant à partir de zéro (c'est-à-dire que `range(n)` génère les nombres entiers de 0 à n–1). Ainsi,

```
spam = list(range(10))
```

créera la liste `[0,1,2,3,4,5,6,7,8,9]`.

On peut aussi utiliser `range()` avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus « sophistiquées » :

```
spam = list(range(3,12))
```

produira la liste `[3,4,5,6,7,8,9,10,11]`. On voit que `range(a,b)` génère tous les nombres entiers de a jusqu'à b–1.

```
spam = list(range(3,12,2))
```

produira la liste `[3,5,7,9,11]`. Donc, `range(a,b,p)` génère des nombres entiers de a à b–1 avec un pas de p.

5.4.2. Manipulation des listes

La fonction `len` s'applique aussi aux chaînes de caractères, aux tuples, ... Ce n'est pas une méthode spécifique aux listes.

La fonction intégrée `len` permet de connaître la longueur (*length* en anglais) de la liste :

```
spam = [2,3,5,7,11]
len(spam)
```

donnera comme résultat 5. Les indices de la liste `spam` vont donc de 0 à 4.
Pour ajouter un élément en fin de liste, on a la méthode `append()` :

```
spam.append(13)
```

qui donnera la liste `[2,3,5,7,11,13]`. Remarquez la forme caractéristique d'une méthode : `objet.méthode()`.
On peut aussi enlever un élément de la liste avec la méthode `remove()` :

```
spam.remove(5)
```

donnera comme résultat `[2,3,7,11,13]`.
Il ne faut pas confondre cette instruction avec `del(spam[5])`, qui permet d'enlever l'élément d'indice 5, donc le 6ème élément de la liste. Par exemple, si `spam=[2,3,5,7,11,13]`, alors

```
del(spam[5])
```

donnera comme résultat `[2,3,5,7,11]`.
On peut aussi facilement concaténer deux listes avec un simple + :

```
spam = [2,3,5,7,11]
cheese = [9,8,7,6,5]
fusion = cheese + spam
```

donnera la liste `[9,8,7,6,5,2,3,5,7,11]`.
On peut trier la liste avec la méthode `sort()` :

```
fusion.sort()
```

donnera la liste `[2,3,5,5,6,7,7,8,9,11]`.
Il arrive parfois que l'on doive inverser la liste :

```
fusion.reverse()
```

donnera la liste `[11,9,8,7,7,6,5,5,3,2]`.

5.4.3. Accès aux éléments d'une liste

On peut accéder à un élément d'une liste en précisant son indice, mais on peut aussi accéder à plusieurs éléments en donnant un intervalle ; le résultat sera alors une liste. Reprenons notre liste spam et observons quelques exemples :

```
spam = [2,3,5,7,11,13]
print(spam[2])
```

donnera comme résultat `5`. C'est un nombre entier.

```
print(spam[1:3])
```

donnera comme résultat `[3,5]`. C'est une liste composée des éléments `spam[1]` et `spam[2]`.

```
print(spam[2:3])
```

donnera comme résultat `[5]`. C'est une liste, comme l'indiquent les crochets, contenant un seul élément.

```
print(spam[2:])
```

donnera comme résultat `[5,7,11,13]`.

```
print(spam[:2])
```

donnera comme résultat `[2,3]`.

```
print(spam[-2])
```

donnera comme résultat `11`. C'est le deuxième élément depuis la fin de la liste.

```
print(spam[-1])
```

donnera comme résultat `13`. C'est le dernier élément de la liste.

On peut savoir si un élément appartient à une liste avec l'instruction `in`.

```
5 in spam
```

donnera comme résultat `True`. On peut aussi connaître l'indice de cet élément :

```
spam.index(5)
```

donnera comme résultat `2`. L'élément 5 est bien en 3ème position de `[2,3,5,7,11,13]`.

On peut modifier un élément de la liste. Par exemple,

```
spam[0]=1
```

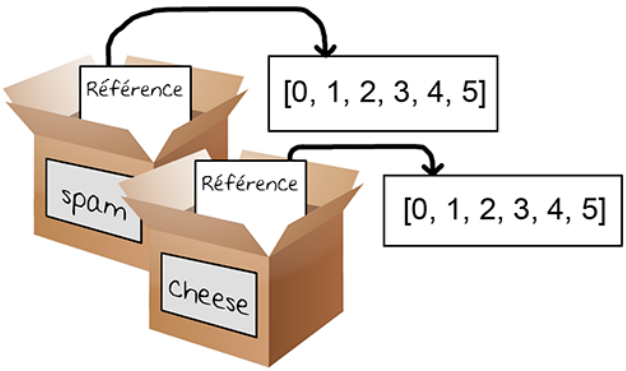
modifiera la liste `spam` ainsi :

```
[1,3,5,7,11,13]
```

5.4.4. Copie d'une liste et insertion dans une liste

Il y a plusieurs façons de copier une liste. La plus courte est :

```
cheese = spam[:]
```



Rappelons encore une fois, car c'est important, que `cheese = spam` ne copie pas la liste, mais crée une nouvelle référence sur la liste `spam`.

Insertion d'un ou de plusieurs éléments n'importe où dans une liste

La technique du « slicing » (découpage en tranches) permet de modifier une liste à l'aide du seul opérateur `[]`. On peut insérer un ou des éléments dans une liste. Par exemple,

```
capitales = ['Paris', 'Rome', 'Berne', 'Vienne']
capitales[2:2] = ['Berlin']
```

donnera comme résultat :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne']
```

et

```
capitales[5:5] = ['Oslo', 'Londres']
```

donnera comme résultat :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne', 'Oslo', 'Londres']
```

Attention aux particularités suivantes :

- Si vous utilisez l'opérateur `[]` à la gauche du signe égal pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une « tranche » dans la liste cible (c'est-à-dire deux index réunis par le symbole **:**), et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égal *doit lui-même être une liste*. Si vous n'insérez qu'un seul élément, il vous faut donc le mettre entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément `capitales[1]` n'est pas une liste (c'est la chaîne `'Rome'`), alors que l'élément `capitales[1:3]` en est une.

Suppression / remplacement d'éléments

On peut aussi supprimer et/ou remplacer des éléments par « slicing ». Partons de cette liste :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne', 'Oslo', 'Londres']
```

Remplaçons la tranche [2:5] par une liste vide, ce qui correspond à un effacement.

```
capitales[2:5] = []
```

produira la liste :

```
['Paris', 'Rome', 'Oslo', 'Londres']
```

Remplaçons une tranche par un seul élément. Notez encore une fois que cet élément doit lui-même être une liste.

```
capitales[1:3] = ['Madrid']
```

produira la liste :

```
['Paris', 'Madrid', 'Londres']
```

Remplaçons une tranche de deux éléments par une autre qui en compte trois.

```
capitales[1:] = ['Lisbonne', 'Prague', 'Moscou']
```

produira la liste :

```
['Paris', 'Lisbonne', 'Prague', 'Moscou']
```

5.4.5. Les boucles for

En Python, une boucle `for` parcourt simplement une liste :

```
liste = ['a','b','c']
for lettre in liste:
    print(lettre,end=' ')
```

écrira `a b c`.
On peut aussi faire de simples compteurs avec une boucle `for` :

```
for chiffre in range(5):
    print(chiffre,end=' ')
```

écrira `0 1 2 3 4`.

Exercice 5.1

Créez une liste de 15 nombres entiers générés au hasard entre 1 et 10. Certains nombres pourront y figurer à plusieurs exemplaires. Écrivez un programme qui recopie cette liste dans une autre, en omettant les doublons. La liste finale devra être triée.

Exercice 5.2

Soient les listes suivantes :

```
liste1 = [31,28,31,30,31,30,31,31,30,31,30,31]

liste2 = ['Janvier','Février','Mars','Avril','Mai','Juin','Juillet','Août',
          'Septembre','Octobre','Novembre','Décembre']
```

Écrivez un programme qui *insère* dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant :

```
liste2 = ['Janvier',31,'Février',28,'Mars',31, ...]
```

Insérez ensuite le nombre 29 entre 28 et 'Mars'.

```
liste2 = ['Janvier',31,'Février',28,29,'Mars',31, …]
```

Exercice 5.3

Réécrivez le code de l'exercice 2.11 (calcul des probabilités au jeu « Risk ») en utilisant des boucles `for` et des listes.

5.5. Code du programme

juniper-green.py

```
# Jeu de Juniper-Green

from random import randint

def multiples(n):
    #renvoie la liste des multiples de n <= Nmax
    mult=[]
    i=2
    while i*n <= Nmax :
        if i*n in possibles:    # on l'ajoute seulement s'il n'a pas été joué
            mult.append(i*n)
```

```
        i += 1
    return mult

def diviseurs(n):
    #renvoie la liste des diviseurs de n
    div = []
    i=n
    while i >= 1:
        if n%i == 0 and n//i in possibles:    # on l'ajoute s'il n'a pas été joué
            div.append(n//i)
        i-=1
    return div


Nmax = 20
possibles = list(range(1,Nmax+1))     # liste des nombres pas encore utilisés
mon_nombre=2*randint(1,Nmax/2)        # l'ordinateur choisit un nombre pair
possibles.remove(mon_nombre)          # on enlèvera de la liste "possibles" tous les
                                      # nombres joués

# Début du jeu

print("Jouons avec des nombres entre 1 et",Nmax)
print("Je choisis comme nombre de départ",mon_nombre)
valides = diviseurs(mon_nombre) + multiples(mon_nombre)
while valides != []:
    print("Nombres valides:",valides)
    ton_nombre=int(input("Que jouez-vous ? "))
    while ton_nombre not in valides:
        ton_nombre=int(input("Incorrect. Que jouez-vous ? "))
    possibles.remove(ton_nombre)
    valides = diviseurs(ton_nombre) + multiples(ton_nombre)
    if valides == []:
        print("Bravo!")
    else:
        mon_nombre = valides[randint(0,len(valides)-1)]
        print("Nombres valides :",valides)
        print("Je joue",mon_nombre)
        possibles.remove(mon_nombre)
        valides = diviseurs(mon_nombre) + multiples(mon_nombre)
        if valides == []:
            print("Vous avez perdu!")
```

5.6. Analyse du programme

Comme les informations concernant les listes ont été données avant de montrer le code, nous ne verrons ici que le fonctionnement général du programme. Le lecteur devrait pouvoir comprendre le code assez aisément.

Une liste, nommée `possibles`, contient tous les nombres qui n'ont pas encore été joués. Au début, elle contient tous les nombres de 1 à `Nmax`. Chaque fois qu'un nombre a été joué, il est retiré de cette liste.

Le programme utilise la liste `valides` pour savoir quels nombres on peut jouer. Le jeu s'arrêtera quand cette liste sera vide. Au début du jeu, elle contient tous les diviseurs et les multiples du nombre de départ choisi par l'ordinateur (`mon_nombre`).

Ces deux listes sont des variables globales.

À tour de rôle, chaque joueur (la machine ou le joueur humain) choisit un nombre dans la liste `valides`, tant qu'elle n'est pas vide. Ce nombre est retiré de la liste `possibles`. La liste `valides` est mise à jour, et on recommence jusqu'à la victoire d'un des deux joueurs.

Exercice 5.4

Modifiez le code du § 5.5.

Avant de commencer le jeu, le programme demandera au joueur d'entrer la valeur de `Nmax`. Il demandera aussi si le joueur veut jouer en premier ; il faudra évidemment adapter le programme en conséquence.

Exercice 5.5

Modifiez le code de l'exercice 5.4.

Dans la version de l'exercice 5.4, l'ordinateur tire simplement au hasard un nombre parmi ceux qui sont valides. Trouvez une meilleure stratégie et programmez-la !

Exercice 5.6

Une variante pour jouer seul au jeu de Juniper Green consiste à trouver la plus longue partie possible pour un `Nmax` donné.

Programmez une méthode *probabiliste* : faites des milliers de parties en jouant des coups (légaux) au hasard et gardez en mémoire la partie la plus longue, que vous afficherez à la fin.

Note : la partie trouvée ne sera pas forcément la plus longue, mais plus vous ferez de parties, plus vous vous en approcherez.

Il semble que la longueur maximale soit de 41 coups.

Record actuel : 38 coups (11.12.2014, plusieurs fois égalé depuis mais jamais battu)

```
Je joue 5000000 parties avec des nombres entre 1 et 50
Plus longue partie: 38 coups
22, 1, 17, 34, 2, 26, 13, 39, 3, 33, 11, 44, 4, 24, 6, 30, 15, 45, 9, 18, 36, 12,
48, 16, 32, 8, 40, 20, 10, 50, 25, 5, 35, 7, 28, 14, 42, 21
```

Autres parties de 38 coups :

(5.12.2019)

```
46, 23, 1, 33, 11, 22, 44, 2, 26, 13, 39, 3, 36, 12, 24, 4, 28, 14, 42, 21, 7,
35, 5, 25, 50, 10, 20, 40, 8, 32, 16, 48, 6, 30, 15, 45, 9, 27
```

(19.12.2019)

```
34, 17, 1, 26, 13, 39, 3, 33, 11, 22, 44, 2, 40, 20, 10, 50, 25, 5, 35, 7, 21,
42, 14, 28, 4, 8, 32, 16, 48, 24, 12, 36, 18, 9, 45, 15, 30, 6
```

Exercice 5.7 : les vignettes Panini

Collectionner les vignettes autocollantes vendues à l'occasion de la coupe du monde de football : voici une activité (onéreuse) à laquelle s'adonnent avec joie de nombreux enfants (et leurs parents).

Imaginons que l'on commence un nouvel album. Il faudra coller 700 vignettes. Supposons que l'on achète les vignettes à l'unité. Évidemment, on ne sait pas quelle vignette on achète : elle est scellée dans une pochette. Si on ne fait pas d'échanges, combien de vignettes faudra-t-il acheter pour remplir l'album ?

Répondez à cette question à l'aide d'un programme Python.

Exercice 5.8 : Le Verger

Le **Verger** (Obstgarten) est un jeu de société coopératif créé par Anneliese Farkaschovsky. Il a été édité la première fois en 1986 par la société Haba.

Le jeu contient :

- un plan de jeu présentant 4 arbres fruitiers (prunier, pommier, poirier, cerisier) ainsi qu'un corbeau au centre.
- 40 fruits (10 pour chaque arbre du plan de jeu) en bois à disposer sur chacun des arbres.
- 4 paniers pour les joueurs (1 panier par joueur).
- un puzzle corbeau de 9 pièces que l'on reconstituera au fil du jeu.
- un gros dé.

But du jeu

Le but du jeu est de récupérer tous les fruits avant d'avoir reconstitué le puzzle corbeau.

Règles du jeu

Le plus jeune joueur commence en lançant le dé. Ce dé a quatre faces de couleur (jaune pour la poire, vert pour la pomme, bleu pour la prune et rouge pour la cerise), ainsi qu'une face « panier », et une face « corbeau ».

- S'il tombe sur une face couleur, il prend le fruit correspondant à la couleur, et le met dans son panier. S'il n'y a plus de fruit sur l'arbre, le joueur passe son tour.
- S'il tombe sur « panier », il prend deux fruits de son choix.
- S'il tombe sur « corbeau », il place une des pièces de puzzle corbeau sur le plateau de jeu.

Le jeu continue avec le joueur suivant.

Le gagnant

Les joueurs gagnent tous ensemble s'ils ont réussi à cueillir tous les fruits des arbres.

Les joueurs perdent ensemble si le puzzle corbeau de 9 pièces est reconstitué entièrement avant qu'ils n'aient pu récupérer tous les fruits.

Programme demandé

Écrivez un programme qui simulera 100'000 parties de ce jeu. Vous indiquerez comme résultat le pourcentage de victoires des joueurs, ainsi que le nombre moyen de coups dans une partie.

Exercice 5.9 : Les cartes de loto

0	1	2	3	4	5	6	7	8
2		23		43	51			81
	16	24		48		61	77	
4	19		38			64		88

Écrivez un programme produisant une carte de loto (voir l'exemple ci-dessus, avec en rouge les numéros des colonnes). Il faudra tout d'abord trouver un algorithme permettant de créer une carte en respectant ces contraintes :

- Une carte contient 9 colonnes et 3 lignes.
- Il y a sur la carte 15 numéros différents choisis parmi les nombres de 1 à 90.
- Chaque ligne contient 5 numéros (et donc 4 espaces vides).
- Il y a toujours au moins un numéro par colonne.
- Il peut y avoir 3 numéros dans une colonne, mais seulement dans la colonne 8.
- La colonne 0 contient les numéros de 1 à 9.
- La colonne 1 contient les numéros de 10 à 19.
- La colonne 2 contient les numéros de 20 à 29.
- …
- La colonne 7 contient les numéros de 70 à 79.
- La colonne 8 contient les numéros de 80 à 90.

5.7. Tuples

Python propose un type de données appelé tuple, qui est une liste non modifiable.
Du point de vue syntaxique, un tuple est une collection d'éléments séparés par des virgules :

```
alpha = 'a', 'b', 'c', 'd', 'e'
print(alpha)
```

donnera comme résultat :

```
('a', 'b', 'c', 'd', 'e')
```

Bien que cela ne soit pas nécessaire, il est fortement conseillé de mettre le tuple en évidence entre parenthèses pour améliorer la lisibilité du code. Comme ceci :

```
alpha = ('a', 'b', 'c', 'd', 'e')
```

Les opérations que l'on peut effectuer sur des tuples sont les mêmes que celles que pour les listes, si ce n'est que les tuples ne sont pas mutables. Il est donc impossible d'ajouter ou de supprimer des éléments d'un tuple.

5.7.1. Utilité des tuples

Les tuples trouvent principalement leur utilité dans deux cas :

- ils sont utilisés quand une fonction doit renvoyer plusieurs valeurs ;
- comme ils ne sont pas mutables, ils peuvent servir de clés pour un dictionnaire (voir chapitre 6), contrairement aux listes.

Le code ci-dessous définit une fonction appelée `cercle` qui prend en paramètre le rayon `r` du

cercle et qui renvoie deux valeurs : la circonférence et l'aire du cercle. En utilisant un tuple comme type de la valeur de retour, une fonction peut retourner plusieurs valeurs.

```
import math

def cercle(r):
    #Retourne le couple (p,a) : périmètre et aire d'un cercle de rayon r
    return (math.pi*2*r,math.pi*r*r)

(p,a) = cercle(4)
print("périmètre =",p)
print("aire =",a)
```

Répétons que les parenthèses des tuples ne sont pas indispensables. Voici la version sans parenthèses :

```
import math

def cercle(r):
    #Retourne le couple (p,a): périmètre et aire d'un cercle de rayon r
    return math.pi*2*r,math.pi*r*r

p,a = cercle(4)
print("périmètre =",p)
print("aire =",a)
```

Exercice 5.10

Réécrivez le code de l'exercice 4.4 du chapitre précédent, en utilisant un tuple pour désigner les trois coups possibles (pierre, papier ou ciseaux), à la place d'une fonction.

5.8. Ce que vous avez appris dans ce chapitre

- Vous avez pour la première fois rencontré la notion d'objet, qui est très importante en Python.
- Les listes sont des structures de données très utiles. Les principales instructions ont été présentées pour les créer et les manipuler.
- L'instruction `for` permet de parcourir une liste.
- Un tuple ressemble beaucoup à une liste, à part qu'il n'est pas modifiable.