COURS PYTHON TSI2

SOMMAIRE

Chapitre 1

Dictionary

Objectifs

- **Les incontournables :**
  - savoir

Résumé de cours

■ Introduction

Les dictionnaires permettent de mémoriser des données. Ces données peuvent être des résultats intermédiaires qui sont produits par des calculs et qui doivent être réutilisés ensuite.
Les dictionnaires sont des types construits. Rappelons les types principaux de Python.
▶ Le type `tuple` : C'est une liste ordonnée d'objets délimitée par des parenthèses. Ces objets sont indéxés par des entiers.

```
a = (1,2,3) ; a = a + (5,6)
a
    (1, 2, 3, 5, 6)
```

On peut concaténer des `tuples`.

▶ Le type `list` : C'est une liste ordonnée d'objets délimitée par des crochets. Ces objets sont indéxés par des entiers.

```
l=[3,1,4]; l.append(5)
l
    [3, 1, 4, 5]
```

▶ Le type `dict` : C'est une liste non ordonnée d'objets délimitée par des accolades, appelée dictionnaire. Ces objets sont accessibles par des clés (au lieu d'un indice). Une clé peut être n'importe quel objet, un `tuple` mais pas une liste ou un autre dictionnaire. On construit un dictionnaire en associant des successions de couples, chaque couple est constitué d'un premier élément, appelé clé, `keys` en Python et un second appelé valeur, `values` en Python.

```
d={0:1,'login':'lppr','pass':'lppr','proc':64}
d['login']
   'lppr'
for cle in d.keys(): print(cle)
  0
  login
  pass
  proc
```

```
for val in d.values(): print(val)
  1
  lppr
  lppr
  64
for cle,val in d.items(): print(cle,val)
  0 1
  login lppr
  pass lppr
  proc 64
valeur = d.pop('proc') ; valeur
  64
```

■ Implémentation

Une liste en Python est implémentée de manière à ce que certaines opérations soient très efficaces, c'est-à-dire avec une complexité en temps constante. Ces opérations sont par exemple : obtenir la longueur d'une liste, accéder à un élément et modifier un élément. La suppression et l'insertion d'un élément en fin de liste ont aussi un coût que l'on peut considérer comme constant.

Les dictionnaires ne sont pas des séquences ce qui siginifie que les éléments n'ont pas une place, repérée par un indice entier, qui permet d'ordonner l'ensemble pour le parcourir et par la même occasion accéder à un élément particulier en utilisant son indice. Dans un dictionnaire, un élément n'a ni prédécesseur ni successeur. L'objectif est de disposer d'opérations similaires à celles énumérées ci-dessus pour les listes, avec la même efficacité.

Il est important de noter que les éléments d'un dictionnaire n'ont pas à être ordonnés. Donc pour le parcours et l'affichage des clés ou des couples (clé, valeur), il n'y a pas d'ordre prévisible.

Il existe différentes manières d'implémenter un dictionnaire.

Exemple 1

Par exemple, si le dictionnaire **a pour clés des entiers naturels**, on peut utiliser un tableau. Si un indice est égal à une clé, on écrit la valeur correspondante, sinon on écrit une valeur par défaut. Dans ce cas, la complexité en espace est en $O(N)$, où N est la plus grande clé.

Rappelons le principe d'un tableau qui peut être utilisé par exemple pour stocker les adresses en mémoire des éléments d'une liste en Python.
On accède à un élément par son indice. Les adresses des éléments sont enregistrées sur des mots de taille fixe (8 ou 4 octets) de manière séquentielle. Ainsi, si l'on connait l'adresse du début, on peut obtenir l'adresse d'un élément quelconque en calculant par une opération simple sur l'indice la place où se trouve cette adresse. On prévoit un tableau assez grand pour pouvoir ajouter quelques éléments.

Illustration par la figure 1

Exemple 2

Plaçons nous dans le cas général où **les clés ne sont pas toutes des entiers naturels**. Une implémentation classique d'un dictionnaire utilise une **table de hachage**. On dispose d'une **fonction de hachage**, c'est-à-dire une fonction qui à une clé associe un entier appelé **valeur de hachage**

de la clé. Cet entier est utilisé pour calculer l'indice d'un tableau où est placée la clé.
On peut ainsi stocker suivant l'indice calculé pour chaque clé, l'adresse de la clé puis celle de la valeur et stocker la clé et la valeur aux adresses indiquées. Cela revient à utiliser une liste ordonnée suivant les indices calculés pour les clés avec successivement des clés et des valeurs et des places vides. Pour trouver un élément, on calcule l'indice, on trouve l'adresse de la clé et on vérifie. Ceci est exécuté en temps constant. On peut aussi stocker la valeur de hachage.

On doit donc disposer d'une fonction f avec $f(e) = p$ qui calcule la place p d'un élément e. Deux éléments distincts devant se trouver à deux places distinctes, la fonction f doit être bijective. Ce point a une première conséquence : si une clé `x` est mutable, sa place va changer à chaque mutation. Donc on ne peut utiliser comme clé que des objets non mutables, plus précisément non récursivement mutables.

Illustration par la figure 2

Exemple 3

Une autre possibilité est de stocker suivant l'ordre de création, la valeur de hachage calculée pour la clé, l'adresse de la clé puis celle de la valeur et stocker la clé et la valeur aux adresses indiquées. Cela revient à utiliser une liste ordonnée suivant l'ordre de création avec successivement une valeur de hachage, une adresse de clé et une adresse de valeur et ainsi de suite, sans place inoccupée. On complète alors avec un tableau d'octets où chaque octet, à la place dont l'indice est calculé pour une clé, donne la position dans la liste des trois éléments valeur de hachage, clé, valeur. Les octets ne correspondant à aucune valeur de hachage valent -1 pour indiquer une place vide et -2 pour un élément supprimé.
Donnons un exemple où seules les adresses des clés sont représentées.

Considérons le dictionnaire

```
{97: "a", 101:"e", 114:"r", 111: " o"}
```

et un tableau `T` de huit octets supposé être la capacité du dictionnaire. ce tableau `T` va donner les positions des clés dans le dictionnaire.
Soit h la fonction de hachage et une clé `c` qui est la $k^{\text{ème}}$ insérée alors si `p` est la position calculée avec `p = h(c) modulo 8` alors `T[p] = k`.
Si l'on ne considère que les clés, le tableau des données contient dans l'ordre $97, 101, 114, 111, 0, 0, 0, 0$.
L'élément 97 a pour indice 0, l'élément 101 a pour indice 1, l'élément 114 a pour indice 2 et l'élément 111 a pour indice 3.
On choisit pour h la fonction identité (ce qui est logique, on verra plus loin).
Alors `h(97) modulo 8` est `97 modulo 8` soit 1. Alors $T[1] = 0$.
Puis `h(101) modulo 8` est `101 modulo 8` soit 5. Alors $T[5] = 1$.
Puis `h(114) modulo 8` est `114 modulo 8` soit 2. Alors $T[2] = 2$.
Enfin, `h(111) modulo 8` est `111 modulo 8` soit 7. Alors $T[7] = 3$.
Le tableau T contient dans l'ordre :

$$-1, 0, 2, -1, -1, 1, -1, 3$$

En effet, $T[0]$ par exemple n'est pas affecté et est une place vide et cet octet ne correspond à aucune donnée. On peut mettre 255 à la place de -1 (car $2^8 = 256$).
Pour accéder à la clé 101, comme sa position `p = h(101) modulo 8` vaut 5, on regarde dans T à la position 5 et on y lit la valeur 1 qui est la position de 101 dans le dictionnaire.

Si l'on supprime par exemple la clé 97, on remplace dans T à l'indice 1 la valeur 0 par -2 (ou 254).

Fonctions de hachage

Comment obtenir une bonne fonction de hachage ? Cette fonction, notée h doit permettre un calcul rapide de l'image d'une clé. Les indices calculés en général par h(clé) modulo n, où n est la taille du tableau T doivent être tous distincts.
En pratique, il est très souvent impossible d'avoir une correspondance bijective entre les clés et les indices. Lorsque le même indice est obtenu pour deux clés, on parle de **collision**.
Pour résumer, on procède en deux étapes :

▶ **Étape 1**. Une fonction h de hachage associe un entier à chaque clé. Elle code donc les clés. En Python, la fonction est `hash`

```
hash(55)
   55
hash('pass') ; hash("login")
   -8560737331547147328     -4537507453712144586
hash(2**61-1) ; hash(-1)
    0     -2
```

• Si i est un entier différent de -1 et si $-2^{61}+1 < i < 2^{61}-1$ alors `hash(i)` vaut i.
• Si x est un flottant égal à p/q alors $hash(x)$ vaut $(int(p * M/q)) \% M$ avec $M = 2^{61}-1$.

```
hash(0.1)
    230584300921369408
(int(1*(2**61-1)/10)) % (2**61 -1)
    230584300921369408
```

• Si la clé est une chaîne de caractères la valeur de hachage est calculée avec une part de hasard à chaque nouvelle session et a pour valeur un entier qui s'écrit sur 64 bits.

▶ **Étape 2** Il faut ensuite une fonction qui réduise chaque entier pour obtenir un entier appartenant à $[\![0, n-1]\!]$, où n est la taille du tableau (la capacité du dictionnaire). En Python, on utilise $hash(c) \% n$

■ Gestion des collisions

En Python deux objets distincts de même valeur ont la même valeur de hachage :

```
a=(3,2)
b=(3,2)
a is b
   False
hash(a) == hash(b)
   True
```

Mais deux objets dont les valeurs ne vérifient pas le critère d'égalité peuvent aussi avoir la même valeur de hachage :

```
hash(0)
   0
hash(2**61 -1)
   0
```

Il faut donc trouver des solutions à ce problème de collision.
Pour gérer une collision, on a plusieurs méthodes. En cas de collision, on peut calculer une autre place ou prendre la première place libre qui suit. On commence par prendre un dictionnaire plus long. Une méthode de redimensionnement est utilisée par Python qui prévoit des dictionnaires dont pas plus des deux tiers de la capacité est utilisée. Si l'on atteint les deux tiers de la capacité et qu'un élément doit être ajouté, la capacité du dictionnaire est multipliée par 2.
Par exemple :

- de 1 à 5 éléments, la capacité est 8 car $(3/2) \times 5 = 7.5$ et $(3/2) \times 6 = 9$.
- Jusqu'à 10 éléments, la capacité est 16 car $(3/2) \times 10 = 15$ et $(3/2) \times 11 = 16.5$

etc.
La capacité n est ainsi une puissance de 2.
Si la taille est $n = 2^p$, alors $h \,\%\, n$ et $h \,\&\, (n-1)$ ont la même valeur.

```
556 % 16
   12
556 & 15
   12
```

En Python, avec un dictionnaire de capacité 8, en cas de collision à une place i, la nouvelle place est calculée avec la formule :

$$i = (5 * i + (h // (2 * *5)) + 1) \,\%\, 8$$

où h a pour valeur `hash(c)` si `c` est la clé.
Si les clés sont des entiers i strictement inférieurs à 32, l'entier h est i et `h // (2**5)` vaut 0 et la formule se réduit à : $i = (5 * i + 1) \,\%\, 8$.
Ainsi si $i = 4$, les 7 valeurs successives sont :

```
ind=4
for k in range(1,8) :
    ind=(5*ind+1)%8; print(ind)
5
2
3
0
1
6
7
```

On parcourt ainsi les huit places en sachant que quatre au moins sont disponibles.

Exemple 4

Illustrons un problème de collision. Soit le dictionnaire :

```
d = { "i" : 19, "n" : 14, "f" : 6 , " o " : 15}
```

Nous utilisons une liste de longueur 8 pour stocker un dictionnaire de cinq éléments maximum.

```
def dic(couples):
    liste = 8 * [None]
    for cle, val in couples:
        liste[hash(cle)%8]=(cle,val)
    return liste
```

```
dic((("i",19),("n",14),("f",6), ("o",15)))
    [None, None, ('n', 14), None, None, None, ('o', 15), None]
```

On voit que le couple `("f",6)` par exemple a disparu car `("o",15)` a été le dernier affecté en liste[6] car $hash("f")\%8$ et $hash("0")\%8$ sont les mêmes.
On modifie la fonction `dic` pour gérer les collisions.

```
def dic(couples):
    liste = 8* [None]
    for cle, val in couples :
        i = hash(cle) % 8
        while liste[i] is not None :
            i = (5*i +1) % 8
        liste[i] = (cle,val)
    return liste
dic((("i",19),("n",14),("f",6), ("o",15)))
   [None,None,('n',14),None,('o', 15),None,('i', 19),('f', 6)]
```

La place d'indice 7 est occupée après l'insertion de `("f",6)` et la nouvelle place calculée pour insérer `("o",15)` est

```
(5*7+1)%8
    4
```

C'est bien l'indice 4.

■ Manipulation

Construction

Pour l'instant, on a crée un dictionnaire en plaçant entre des accolades des couples (clé,valeur) séparés par des virgules, chaque clé et sa valeur associée étant séparées par deux points. On peut construire un dictionnaire par **compréhension** :

```
d  = {x : x**2 for x in range(1,5)}
d
   {1: 1, 2: 4, 3: 9, 4: 16}
```

Ou aussi par **insertion**. On crée un dictionnaire vide puis on insère les éléments par une boucle.

```
d = {}
for x in range(1,5):
    d[x] = x**2
d
    {1: 1, 2: 4, 3: 9, 4: 16}
d[3]
    9
```

Au passage `d[x]` renvoie la valeur dont `x` est la clé.
De même que les éléments d'une liste peuvent être des listes, les éléments d'un dictionnaire peuvent être des dictionnaires :

```
 pays = {"France":{"capitale": "Paris",
                   "population": 68014000,
                   "superficie": 643800.0},
         "Portugal" : {"capitale": "Lisbonne",
                    "population": 10302674,
                   "superficie":92300.0},
         "Italie" :{"capitale": "Rome",
                    "population" : 60359546,
                    "superficie" : 301336.0}}
pays["France"]["population"]
    68014000
pays["France"]
    {'capitale': 'Paris', 'population': 68014000, 'superficie':
       643800.0}
```

Utilisation

Accès aux éléments

Deux méthodes donnent accès aux clés ou aux valeurs, ce sont les méthodes `keys` et `values`. Et la méthode `items` donne accès à l'ensemble des couples. Le mieux c'est le faire fonctionner.

```
d = {"true":"vrai", "false" : "faux", "and" : "et", "or " : "ou"}
d.keys()
    dict_keys(['true', 'false', 'and', 'or '])
d.values()
    dict_values(['vrai', 'faux', 'et', 'ou'])
d.items()
    dict_items([('true', 'vrai'), ('false', 'faux'), ('and', 'et'
        ), ('or ', 'ou')])
```

L'accès à une valeur s'obtient comme avec les listes. La différence est qu'il faut préciser la clé à la place de l'indice.

```
d["true"]
   'vrai'
```

Appartenance

le mot clé permet de tester l'appartenance d'une clé à un dictionnaire par l'appartenance d'une valeur.

```
"vrai" in d, "and" in d
   (False, True)
```

Boucles

On peut itérer avec une boucle `for` sur un dictionnaire, la variable d'itération est une clé.

```
cles=[]
for obj in d:
    cles.append(obj)

cles
   ['true', 'false', 'and', 'or ']
```

Il est possible avec une boucle `for` d'itérer sur les clés, sur les valeurs ou sur les couples (clé, valeur) à l'aide des objets `d.keys()`, `d.values()` et `d.items()`

```
val = []
for newobj in d.values():
    val.append(newobj)
val
   ['vrai', 'faux', 'et', 'ou']
```

Propriété
On ne peut pas modifier la taille d'un dictionnaire durant une itération sans un message d'erreur.

```
dd = {0:0, 1:-5, 2:4}
for c in dd:
    dd[c+1] = dd[c] + 1

Traceback (most recent call last):
File "<ipython-input-22-04e6fc6c3560>", line 1, in <module>
    for c in dd:
RuntimeError: dictionary changed size during iteration
```

Pourtant, l'opération se fait quand-même.

```
dd
  {0: 0, 1: 1, 2: 2}
```

Nombre d'éléments
La fonction `len` renvoie la longueur d'un dictionnaire c'est-à-dire le nombre de couples (clé, valeur).

```
len(d),len(dd)
  (4, 3)
```

Suppression
Pour supprimer un élément, nous utilisons la fonction `del`

Copie
On use de `d.copy()`. La vigilance s'impose, comme avec les listes, puisque les dictionnaires sont des objets mutables.

```
d2 = d.copy()
d2
  {'true': 'vrai', 'false': 'faux', 'and': 'et', 'or ': 'ou'}
del d2["false"]
d
  {'true': 'vrai', 'false': 'faux', 'and': 'et', 'or ': 'ou'}
d2 = d
del d2["false"]
d
  {'true': 'vrai', 'and': 'et', 'or ': 'ou'}
```

Donc `d` a perdu `"false"` alors qu'on l'a enlevé qu'à `d2`.
Autre exemple où le même pb survient.

```
d3 = {"true": ["vrai", "vraie"]}
d4 = d3.copy()
d4
  {'true': ['vrai', 'vraie']}
d4["true"][1] = "vrais"
d3
  {'true': ['vrai', 'vrais']}
d4
  {'true': ['vrai', 'vrais']}
```

■ Applications dans le langage Python

Les dictionnaires sont des objets au coeur du fonctionnement du langage Python. Lorsqu'on exécute un programme, plusieurs dictionnaires sont mobilisés en arrière plan, même si le programme ne contient aucune définition explicite de dictionnaire.

Espace de noms

Les noms appartenant à un espace de noms quelconque sont les clés d'un dictionnaire. Au démarrage de l'interpréteur, le module `__builtins__` est chargé. Ce module contient tous les objets que nous pouvons utiliser directement dans le programme et le dictionnaire `__builtins__.__dict__` lié à ce module contient tous les identifiants liés à ces objets comme `help,len,True,max` etc. Donnons un exemple (la barre `__` devant et après `builtins` et `dict` est constituée de deux fois le tiret sous la touche 8.

```
__builtins__.__dict__['min'](5,2,3)
   2
```

Le contenu des modules que nous importons est représenté par un dictionnaire.
Si nous importons `math` avec `import math` alors nous obtenons le contenu, comme les fonctions disponibles par `dir(math)`. Ceci revient à demander la liste des clés du dictionnaire `math.__dict__`
Si l'on tape ces deux commandes, la dernière écrit les mêmes fonctions que la première mais avec la syntaxe d'un dictionnaire.

```
dir(math)
 ['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 ...
 'tau',
 'trunc']
```

```
math.__dict__
   {'__name__': 'math',
    '__doc__': 'This module provides access to the mathematical
       functions\ndefined by the C standard.',
 '__package__': '',
 '__loader__': _frozen_importlib.BuiltinImporter,
 '__spec__': ModuleSpec(name='math', loader=<class '
     _frozen_importlib.BuiltinImporter'>, origin='built-in'),
 'acos': <function math.acos(x, /)>,
 'acosh': <function math.acosh(x, /)>,
 'asin': <function math.asin(x, /)>,
 ...
  'e': 2.718281828459045,
 'tau': 6.283185307179586,
 'inf': inf,
 'nan': nan}
```

Tous les identifiants des variables et des fonctions que nous définissons sont stockés dans un dictionnaire que nous obtenons avec `globals()`. Tapez le et vous verrez apparaître tout notre passé du jour.
Les paramètres d'une fonction et les variables définies dans le corps d'une fonction sont stockés dans un dictionnaire créé à l'exécution de la fonction et supprimé à la fin de l'exécution. On peut observer son contenu en ajoutant dans le corps de la fonction `print(locals()`

```
def f(x,y):
    z = x+y
    print(locals())
f(3,4)
 {'x': 3, 'y': 4, 'z': 7}
```

Notion de portée
Définissons une fonction `abs` dans un programme en cours. Le nom `abs` appartient alors à l'espace de nom local de ce programme. Donc si l'on utilise `abs` dans ce programme, c'est l'espace local qui est examiné en premier et donc cette fonction est utilisée en premier. Mais la fonction `abs` connue (la valeur absolue) du module `__builtins__` n'a pas été modifiée.

```
def abs(x):
   return x
abs(-5)
    -5
__builtins__.abs(-5)
    5
```

Notion d'affectation
Écrire par exemple $x = 300$ et `globals()["x"]`$= 300$ est équivalent. On ajoute au dictionnaire la clé `x` associée à la valeur 300. Autrement dit, on ajoute au dictionnaire le couple (identité ou adresse de l'objet de type `str` et de valeur $'x'$, identité de l'objet de type `int` et de valeur `300`). Si l'on écrit ensuite $x = 500$, on remplace l'identité de l'objet de type `int` et de valeur 300 par celui de valeur 500. Le premier objet n'est pas modifié mais `x` est lié à un nouvel objet.

Exécution d'une fonction
Tapons :

```
def f(f):
    return 2*f
f(5)
    10
```

La clé `f` (le nom de la fonction) est ajoutée au dictionnaire `globals()` et liée à l'objet de type `function` et de valeur le code de la fonction.
Lors de l'exécution, un dictionnaire local lié à la fonction est créé. Ce dictionnaire contient pendant l'exécution le nom `f` (celui du paramètre), associé à la valeur 5.

```
def f(f):
    return 2*f, locals()
f(5)
    (10, {'f': 5})
```

Pour finir, les dictionnaires sont aussi utilisés dans des opérations de comptage et pour l'implémentation des graphes. Ils sont utilisés aussi pour la modélisation des jeux ou le traitement de données en tables etc.

Énoncé des exercices

Exercice 1.1 : Considérons un dictionnaire `d1` et le code suivant :

```
d2 = {}
for c,v in d1.items():
    d2[v] = c
```

Le dictionnaire `d2` a-t-il la même longueur que le dictionnaire `d1` ?

Exercice 1.2 : Soit des polynômes à coefficients entiers de degré quelconque mais qui ne contiennent pas plus de cinq monômes. On utilise un tableau de longueur $16 = 8 \times 2$ pour stocker les couples (degré, coefficient) dans lequel on pourra stocker au maximum huit couples. Les places non occupées contiennent la valeur -1. La fonction de hachage h est la fonction identité : pour tout $n \in \mathbb{N}$, $h(n) = n$. Si n est le degré, $n \,\%\, 8$ donne un indice et à cet indice, on écrit le degré (la clé), suivi du coefficient, (la valeur). Par exemple, $8 + 3x^{10} - 5x^{12}$ est stocké dans :

indice 0	0	8
indice 1	-1	-1
indice 2	10	3
indice 3	-1	-1
indice 4	12	-5

1. Donner le tableau correspondant au stockage du polynôme $2x^5 - 3x^{34} + 4x^{105}$.
2. Quel est le problème par exemple avec le polynôme $8 - 5x^2 + 3x^{10}$?
3. En cas de collision, on décide d'utiliser la première place libre suivante. Les monômes sont rentrés dans le tableau suivant l'ordre de lecture. Donner un exemple de polynôme de degré minimum qui génère une collision pour chaque monôme excepté le premier.
4. On envisage une autre possibilité de stockage avec deux tableaux, un tableau pour les couples (degré, coefficient) et un tableau pour les indices, les deux tableaux ayant pour capacité 8.
Avec le polynôme $4x^3 - 2x^5 + 4x^9$, on obtient les deux tableaux de la manière suivante :

- dans le premier, on écrit chaque degré avec le coefficient correspondant suivant l'ordre des degrés et on complète le tableau avec des 0 ;
- dans le second, on calcule $d \,\%\, 8$, où d est un degré et on place à l'indice trouvé l'indice où on trouve le couple (degré, coefficient) dans le premier tableau. On complète avec des -1.

Extraits des tableaux :

indice 0	3	4
indice 1	5	-2
indice 2	9	4
indice 3	0	0
...	...	...

indice 0	-1
indice 1	2
indice 2	-1
indice 3	0
indice 4	-1
indice 5	1

Donner les deux tableaux correspondant au stockage du polynôme $3x^5 - x^{18} + 7x^{20}$.

Exercice 1.3 : 1. Importer le module `sys` et taper `sys.hash_info`. Remarquer que si `width` vaut 64 (système 64 bits) alors `modulus` a pour valeur $2^{61} - 1$.
2. Créer une constante M de valeur `modulus`.
Écrire une fonction `hachage` qui prend en paramètres deux entiers p et q avec q non nul, représentant un flottant $f = p/q$ et qui renvoie $int(abs(p) * M \,/\, abs(q)) \,\%\, M$ si f est positif et l'opposé de cette expression si f est négatif. Si f vaut -1, la fonction renvoie -2.
3. Comparer `hachage(3,2)` avec `hash(3/2)` et `hash(1.5)`. Puis comparer `hachage(115,100)` et `hash(1.15)` et enfin comparer `hachage(-7,3)`, `hachage(7,-3)` et `hash(-7/3)`.

Exercice 1.4 : On utilise un dictionnaire pour comparer des listes de nombres qui sont tous du même type, soit du type `int` soit du type `float`.
1. On rappelle que si `d` est un dictionnaire, `d[n]` renvoie la valeur dont `n` est la clé et alors dans `d`, on a rajouté la structure `n : d[n]` qui correspond au couple `(n, d[n])`
On considère ici la fonction `occurrences` suivante :

```
def occurrences(nombres):
    d = { }
    for n in nombres:
        if n in d :
            d[n] = d[n] + 1
        else :
            d[n] = 1
    return d
```

Appliquer là à la liste `[1,5,5,2,0,-5,-4,-5,-5,7])` puis `[3,5,-2,3,3,-2,3]`.
Que renvoie cette fonction ?
2. Écrire une fonction `taille` qui prend en paramètre un dictionnaire obtenu comme ci-dessus et renvoie la longueur de la liste qui a été considérée.
3. Écrire une fonction `compare` qui prend en paramètres deux listes de nombres de même longueur et renvoie `True` si les deux listes contiennent les mêmes nombres, pas nécessairement dans le même ordre et `False` sinon. Utiliser la fonction `occurrences`.
Quelle est la complexité en temps de cette fonction ?

Indications

Ex. 1.1

On pourra prendre par exemple pour commencer pour `d1` *:*

```
d1 = {'true': 'vrai', 'and': 'et', 'or ': 'ou'}
```

Puis pour `d1` *:*

```
d1 = {'true' : 'vrai', 'and' : 'vrai', 'or' : 'ou' }
```

Ex. 1.3

Corrigé des exercices

Exercice 1.1

Donnons deux exemples

```
d1 = {'true': 'vrai', 'and': 'et', 'or ': 'ou'}
d2 = {}
for c,v in d1.items():
    d2[v] = c
d2
    {'vrai': 'true', 'et': 'and', 'ou': 'or '}

d1 = {'true' : 'vrai', 'and' : 'vrai', 'or' : 'ou' }
d2 ={}
for c,v in d1.items():
    d2[v] = c
d2
    {'vrai': 'and', 'ou': 'or'}
```

Exercice 1.2

1. $h(5) = 5$ et $5\%8 = 5$ alors à l'indice 5, on met $5, 2$.
$h(34) = 34$ et $34\%8 = 2$, on met à l'indice 2, $34, -3$.
Enfin $h(105) = 105$ et $105\%8 = 1$, on met à l'indice 1, $105, 4$.
Enfin, on met -1 et -1 à l'indice 0, Al'indice 3 et l'indice 4. Cela donne :

indice 0	-1	-1
indice 1	105	4
indice 2	34	-3
indice 3	-1	-1
indice 4	-1	-1
indice 5	5	2
indice ...	...	...

2. On a $2\%8 = 2$ et $10\%8 = 2$. Les degrés et les coefficients de x^2 et x^{10} sont placés à l'indice 2, on a un problème de collision.

3. Un exemple est le polynôme $1 + x^8 + x + x^2 + x^3$. En effet, le premier monôme utilise l'indice 0. Puis comme $8\%8 = 0$, on met le deuxième monôme à l'indice 1. Puis c'est le même pb pour le suivant. À la fin, on a le tableau :

indice 0	0	1
indice 1	8	1
indice 2	1	1
indice 3	2	1
indice 4	3	1
indice ...	...	...

4. Pour l'exemple $4x^3 - 2x^5 + 4x^9$:
Pour le premier tableau, pas de souci.
Pour le second tableau :
Pour $d = 3$, on a : $3\%8 = 3$ et à l'indice 3, on met 0 car c'est à l'indice 0 du 1er tableau que l'on a le degré 3.
Pour $d = 5$, on a : $5\%8 = 5$ et à l'indice 5, on met 1 car c'està l'indice 1 du 1er tableau que l'on a le degré 5.
Pour $d = 9$, on a : $9\%8 = 1$ et à l'indice 1, on met 2 car c'est à l'indice 2 du 1er tableau que l'on a le degré 9
Cela donne bien le tableau.

Passons à $3x^5 - x^{18} + 7x^{20}$. Le premier tableau ne pose pas de souci.
Pour le second tableau :
Pour $d = 5$, on a : $5\%8 = 5$ et à l'indice 5, on met 0 car c'est à l'indice 0 du 1er tableau que l'on a le degré 5.
Pour $d = 18$, on a : $18\%8 = 2$ et à l'indice 2, on met 1 car c'est à l'indice 1 du 1er tableau que l'on a le degré 18.
Pour $d = 20$, on a : $20\%8 = 4$ et à l'indice 4, on met 2 car c'est à l'indice 2 du 1er tableau que l'on a le degré 20.
Puis on complète avec des -1. Cela donne :

indice 0	5	3
indice 1	18	-1
indice 2	20	7
indice 3	0	0
...	...	...

indice 0	-1
indice 1	-1
indice 2	1
indice 3	-1
indice 4	2
indice 5	0
indice 6	-1
indice 7	-1

Exercice 1.3

1.

```
import sys
sys.hash_info
    sys.hash_info(width=64, modulus=2305843009213693951, inf
        =314159, nan=0, imag=1000003, algorithm='siphash24',
        hash_bits=64, seed_bits=128, cutoff=0)

2**61-1
    2305843009213693951
```

2.

```
M=2**61-1
def hachage(p,q):
    h = int(abs(p)*M/abs(q)) % M
        if p*q <0 :
            h = -h
        if p/q == -1 :
            h = -2
    return h
```

3.

```
hachage(3,2)
     1152921504606846977
hash(3/2)
     1152921504606846977
hash(1.5)
     1152921504606846977
hachage(115,100), hash(1.15)
   (345876451382053889, 345876451382053889)
hachage(-7,3),hachage(7,-3),hash(-7/3)
  (-768614336404564994, -768614336404564994, -768614336404564994)
```

Exercice 1.4

1.

```
def occurrences(nombres):
    d = { }
    for n in nombres:
        if n in d :
            d[n] = d[n] + 1
    else :
            d[n] = 1
    return d
occurrences([1,5,5,2,0,-5,-4,-5,-5,7])
     {1: 1, 5: 2, 2: 1, 0: 1, -5: 3, -4: 1, 7: 1}
occurrences([3,5,-2,3,3,-2,3])
     {3: 4, 5: 1, -2: 2}
```

La fonction `occurrences` qui prend en paramètre une liste de nombres renvoie un dictionnaire dont les clés sont les différents nombres de la liste avec pour valeur le nombre d'occurrences de chaque nombre.
Par exemple `occurrences([3,5,-2,3,3,-2,3])` vaut $\{-2:2; 3:4, 5:1\}$.

2.

```
def taille(d):
    t = 0
    for n in d :
       t = t + d[n]
    return t
taille({3: 4, 5: 1, -2: 2})
    7
```

On a recréé la fonction `len` appliquée au dictionnaire et non pas à une liste.

3.

```
 def compare(nombres1, nombres2):
    d1 = occurrences(nombres1)
    d2 = occurrences(nombres2)
        for n in d1:
            if n not in d2 :
               return False
            elif d1[n] != d2[n]:
               return False
    return True
compare([1,5,5,2,0,-5,-4,-5,-5,7],[3,5,-2,3,3,-2,3,0,5,1])
False
```

Chapitre 2

Machine learning

Objectifs

- **Les incontournables :**
  - savoir

Résumé de cours

■ Apprentissage automatique

Notion d'apprentissage

L'apprentissage automatique est un domaine de l'intelligence artificielle. Il s'agit de permettre à une machine de progresser, d'apprendre par elle-même à améliorer son fonctionnement dans un cadre de résolution d'un problème, mais sans jamais la programmer explicitement pour résoudre ce problème. Des outils mathématiques principalement des outils statistiques, sont appliqués sur des données et les résultats obtenus permettent à la machine d'améliorer peu à peu son efficacité dans la résolution du problème.

À partir de données, un modèle est construit. Par exemple, pour apprendre à reconnaître un élément dans une image, on fournit à la machine des images avec la présence de l'élément ou pas et un modèle est élaboré. Ce modèle permet à la machine, lorsqu'on lui fournit une image nouvelle, de dire si l'élément est présent ou pas. Ceci constitue la phase d'apprentissage. Ensuite, lorsque le modèle est suffisamment correct, on peut l'utiliser en poursuivant ou pas l'apprentissage. Pour pousuivre un apprentissage, il est nécessaire d'avoir un moyen de vérifier une réponse et de la certifier ou pas, afin que la machine puisse prendre en compte dans son modèle les nouvelles informations.

On distingue plusieurs formes d'apprentissage, parmi lesquelles ***l'apprentissage supervisé*** et ***l'apprentissage non supervisé***. Pour la première forme, des données sont fournies à la machine avec un classement. On dit que les données sont étquetées ou que les données sont classées. Pour la seconde forme, les données ne sont pas étiquetées et c'est la structure de ces données que la machine doit essayer de déterminer.

D'autres formes d'apprentissage existent, par exemple ***l'apprentissage par renforcement*** ou ***l'apprentissage en profondeur***.

Dans un apprentissage supervisé, à partir des caractéristiques fournies, deux dans le cas le plus simple, soit 0 ou 1 par exemple, on procède à un classement en décidant des caractéristiques d'un nouvel élément après l'examen des caractéristiques de ces k plus proches voisins. On utilise dans ce cadre ***l'algorithme des k plus proches voisins***, en anglais ***K Nearest Neighbours***.

Noter qu'une méthode de classement est différente d'une méthode de régression. Dans une méthode de regression, des données sont fournies et utilisées dans un calcul qui permet de construire une relation, ou une équation, entre les différentes caractéristiques. On peut alors par exemple estimer pour un nouvel élément la valeur cherchée.
Pour une méthode de classement, on essaie de déterminer l'appartenance à une classe en observant les classes des éléments proches.

Dans un apprentissage non supervisé, on dispose de données non classées et on essaye d'établir une classification, une structure de ces données. Les données sont séparées en plusieurs groupes avec dans chaque groupe des données qui présentent un certain degré de similarité. Ceci revient à créer une partition de l'ensemble des données. On dispose pour cela de ***l'algorithme des k-moyennes***, en anglais ***k-means***.

Notion de partition

Pour classer une donnée parmi des données appartenant à un ensemble E, il est nécessaire de disposer d'une classification, c'est-à-dire de sous-ensembles de E qui constituent une partition de cet ensemble. Chaque élément de E appartient à une et une seule classe. La question est de déterminer à quelle classe appartient la nouvelle donnée.
Établir une classification consiste à écrire une partition de E.

Définition

une partition d'un ensemble E est une famille $(E_i)_i$ de sous-ensembles non vides de E qui vérifient les deux conditions :

- $\bigcup_i E_i = E$.
- $E_i \cap E_j = \emptyset$ pour tout couple (i, j) avec $i \neq j$.

Un sous-ensemble appartenant à cette famille est une partie de E.
Le nombre de parties d'un ensemble E à n éléments est 2^n.

Écrivons un programme qui génère toutes les parties d'un ensemble E.

```
def parties(liste, resultats):
    if len(liste) == 0 :
        resultats.append([])
    else :
        liste2 = liste[1:len(liste)]
        parties(liste2,resultats)
        for k in range(len(resultats)):
            elt = resultats[k] + [liste[0]]
            resultats.append(elt)
```

```
def ens_parties(ensemble):
    rep = []
    parties(ensemble,rep)
    return rep
```

```
E = [1,2,3,4]
print(ens_parties(E))

[[], [4], [3], [4, 3], [2], [4, 2], [3, 2], [4,
    3, 2], [1], [4, 1], [3, 1], [4, 3, 1], [2,
    1], [4, 2, 1], [3, 2, 1], [4, 3, 2, 1]]
```

Explication de cette procédure.
En effet transformons `parties` en terminant par `return resultats`

```
def parties(liste, resultats):
    if len(liste) == 0 :
        resultats.append([])
    else :
        liste2 = liste[1:len(liste)]
        parties(liste2,resultats)
        for k in range(len(resultats)):
            elt = resultats[k] + [liste[0]]
            resultats.append(elt)
    return resultats

liste2 = liste[1:len(liste)]
liste2
   [2,3,4]
parties(liste2,[])
   [[], [4], [3], [4, 3], [2], [4, 2], [3, 2], [4, 3, 2]]
```

On a `liste2` est `liste` privée de `1`.
Puis comme c'est le premier `return` qui s'affiche, on affiche `[1:len(liste)]` qui est l'ensemble des parties de `liste2`.
Puis la boucle `for k in range(len(resultats)):` n'est autre ici que `for k in range(0):` et dans `elt`, on ajoute `resultats[k]` qui est le vide et `[liste[0]` qui est `1`. Ainsi dans `resultats`, on reprend `parties(liste2,[])` où on ajoute `1`.

■ Apprentissage supervisé

Algorithmes des k plus proches voisins

Un ensemble de données est connu et chacun des données appartient à une classe ou une partie bien déterminée. Cette classe est une des caractéristiques de chaque donnée liée aux autres caractéristiques.
Dans un apprentissage supervisé, un algorithme doit permettre d'émettre une prévision sur cette caractéristique à propos d'une donnée dont on ne connaît que les autres caractéristiques, donc de prédire la partie dans laquelle la donnée peut être classée.
Une méthode consiste à baser cette prédiction sur la détermination des classes de ses k plus proches voisins, où k est à préciser, en retenant la classe majoritaire.
On dispose donc d'un ensemble E de n points représentant des données classées ou étiquetées. L'ensemble E est l'ensemble d'apprentissage. On choisit un entier k, plus petit que n, et on dispose d'un point x qui n'est pas dans E. Il s'agit de trouver parmi les points de E les k plus proches de x pour classer x ou l'étiqueter. Le mot « proche » sous-entend une notion de distance. Ce peut être une distance sur les couleurs par exemple sur la quantité de rouge ou sur le niveau de gris. Dans la reconnaissance des caractères, ce peut être une distance sur les formes (tailles, boucles). Ainsi des caractères d'imprimerie comme le b ou le h peuvent être considérés comme proches. Dans tous les cas, les caractéristiques sont exprimées par des valeurs numériques et nous pouvons

utiliser une distance dans un espace de dimension d quelconque (les d coordonnées correspondent à d caractéristiques).

Le cas le plus simple est la recherche du plus proche voisin, c'est-à-dire pour $k = 1$. On l'utilise par exemple pour trouver un chemin reliant des points dans un espace. À partir d'un point origine, on cherche le point le plus proche, et ainsi de suite jusqu'à atteindre un point extrémité. Il s'agit donc d'un algorithme qui permet dans certains cas de produire le chemin le plus court entre les deux extrémités.

En pratique, nous utilisons la distance euclidienne ou plutôt le carré de la distance euclidienne. Donnons cette fonction en dimension n.

```
def d(x,y):
    n=len(x)
    s=0
    for i in range(n):
        s = s + (x[i] - y[i])**2
    return s
```

Exemple

Nous considérons un ensemble E dont un élément est représenté par une liste de deux flottants. Un élément est donc considéré comme un point dans un espace de dimension 2.
On construit une liste appelée `voisins` qui contient les k plus proches voisins d'un point x quelconque n'appartenant pas à E, les voisins étant des éléments de E.
On commence par écrire une fonction `d` qui calcule le carré de la distance euclidienne entre deux points du plan.

```
def d(x,y):
   return (x[0] - y[0])**2 + (x[1] - y[1])**2
```

La liste des points de E est triée selon l'ordre croissant des distances à un point P. On écrit donc une fonction nommée `tri` en utilisant la fonction `sorted` de Python. Commençons par apprivoiser cette fonction `sorted`.
Faisons un cas simple pour commencer.

```
sorted([5,2,3,1,4])
   [1, 2, 3, 4, 5]
```

`sorted` trie dans une liste. Le paramètre `key` que l'on peut ajouter comme attribut dans `sorted` permet de spécifier une fonction qui peut être appelée sur chaque élément de la liste avant d'effectuer des comparaisons. Donnons des exemples :

```
sorted("This is a test string from Walter".split(),key=str.lower)
   ['a', 'from', 'is', 'string', 'test', 'This', 'Walter']
studenten = [('john','A',15),('jane','B',12),('dave','B',10)]
sorted(studenten,key = lambda studenten : studenten[2])
   [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
sorted(studenten,key = lambda studenten : studenten[1])
   [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
sorted(studenten,key = lambda studenten : studenten[0])
   [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Passons à une fonction `tri` d'arguments la liste `E`, le point `P` et la fonction distance `d`.
La fonction `choix(elt)` permet de trier suivant les valeurs d'indices 1. On peut créer E comme une liste de points `(a,b)` (à la place de `[a,b]` qui marche aussi).

```
def tri(E,P,d):
    def choix(elt):
        return elt[1]
    distances = [ (p,d(p,P)) for p in E]
    return sorted(distances, key=choix)

E=[(1,2),(-1,0),(1,1),(1,-1)]
P=(0,0)
tri(E,P,d)
    [((-1, 0), 1), ((1, 1), 2), ((1, -1), 2), ((1, 2), 5)]
```

Donnons la fonction `knn` qui renvoie les k premiers points de la liste triée.

```
def knn(E,P,d,k):
    pts = tri(E,P,d)
    return [elt[0] for elt in pts[0:k]]

knn(E,P,d,3)
  [(-1, 0), (1, 1), (1, -1)]
```

Créons une liste E un peu plus élaborée que la liste E précédente. On l'appellera `points`.
Dans la syntaxe, `random()` fournit un flottant aléatoire entre 0 et 1 et donc `round(100*random(),2)` fournit un flottant avec deux chiffres après la virgule entre 0 et 100.

```
from random import random
points = []
for i in range(1000):
    x = round(100 * random(), 2)
    y = round(100 * random(), 2)
    points.append((x,y))
```

Si l'on tape `points` alors 1000 points apparaissent.

```
P=(0,0)
voisins = knn(points,P,d,4)

voisins
    [(0.96, 2.5), (4.87, 3.16), (3.86, 4.85), (5.32, 4.06)]
```

Supposons maintenant que l'on dispose d'une classification des données de E. Afin de classer la donnée P, l'algorithme lui assigne l'étiquette de la classe majoritaire parmi les voisins qui ont été déterminés.
Plus précisément, si par exemple une classe est représentée par un niveau de gris (gris clair, gris moyen, gris amiral), on prédit le niveau de gris du point en choisissant le niveau de gris qui prédomine parmi les voisins du point.
La manière de déterminer la classe prédominante parmi les voisins repose sur un comptage et une recherche de maximum.
Par exemple, créons une fonction `classe_maj` de premier argument `pts` qui sont les voisins de `P` choisis, de second argument `nb_classes` qui est en fait `len(classes)` si l'on a définit la liste `classes` et de troisième argument `partition` qui est un dictionnaire dont les clès sont les points de E sous forme de couples et les valeurs sont les numéros de classe.
Dans la procédure, on crée `cpts` qui fait le comptage, la variable `maxi` permet de rechercher le maximum, `randrange(4)` par exemple renvoie aléatoirement un entier pris parmi $0, 1, 2$ et 3.

```
def classe_maj(pts,nb_classes,partition):
      cpts = [0] * nb_classes
      for v in pts :
            classe = partition[v]
            cpts[classe] += 1
      maxi = 0
      for n in cpts:
            if n > maxi :
                  maxi = n
      choix = [i for i in range(nb_classes) if cpts[i] == maxi]
      estime = choix[randrange(len(choix))]
      return estime
```

Puis on crée des classes. Ici il y aura trois classes : $0, 1$ et 2. De plus `randrange(0,len(classes)` donne aléatoirement un entier compris entre 0 et `len(classes)` c'est-à-dire 2. La procédure va créer `partition` sous forme d'un dictionnaire.

```
classes = [0,1,2]
from random import randrange
partition = {point : randrange(0,len(classes)) for point in points}
```

Si l'on demande de retourner `partition` (On n'a affiché que les premiers et derniers par mesure d'économie de place) :

```
partition

{(29.69, 20.92): 1, (53.65, 41.9): 2, (63.62, 80.06): 2,
... (82.74, 11.6): 1, (5.5, 44.53): 0, (28.68, 51.55): 0}
```

On exécute alors la fonction `classe_maj` avec la liste des 4 plus proches voisins de P.

```
voisins = knn(points, P,d,4)

voisins
      [(0.15, 1.26), (1.46, 1.13), (3.14, 2.27), (3.32, 3.39)]

classe_maj(voisins,3,partition)
      0
```

Remarque : La notion de plus proches voisins est présente dans de nombreux domaines. Simplement dans l'espace par exemple, elle est utile pour gérer des risques de collisions ou des interactions à distance entre des objets.
Si des objets peuvent être identifiés par des caractéristiques mesurables, il est possible de définir une distance entre ces objets. On peut alors trouver les plus proches voisins d'un nouvel objet dans des questions de reconnaissance ou d'identification.
Des applications pour smartphones permettent de reconnaître une musique, une chanson, son interprète, simplement avec quelques secondes d'écoute. Cet extrait n'est pas comparé dans sa globalité avec les extraits figurant dans une énorme base de données (des millions de titres), afin d'y trouver des « voisins » et de choisir le plus proche. Ce serait beaucoup trop lent. La réussite est basée sur la définition d'un petit nombre de points caractéristiques, des marqueurs, qui permettent de définir de manière unique chaque titre. C'est à partir des marqueurs de l'extrait que la recherche est effectuée dans la base de données.

L'algorithme des k plus proches voisins permet d'émettre des prédictions. Il reste à mesurer la qualité de ces prédictions.

Matrice de confusion

Exemple 1

Considérons un test censé prédire si une personne est porteuse ou non d'une maladie. Ce test fait la bonne prédiction dans 90% des cas. Cela signifie que dans 90% des cas, si une personne est porteuse de la maladie, le test est positif et sinon le test est négatif. On parle de « vrais positifs » et de « vrais négatifs ». Dans 10% des cas, le test ne fait pas la bonne prédiction. Le résultat est donc soit positif alors que la personne n'est pas porteuse de la maladie, c'est un « faux positif » et soit négatif alors que la personne est porteuse de la maladie, on parle de « faux négatif ».
Il est intéressant de connaître la répartition de ces 4 types de résultats.

On peut schématiser avec un nuage de points. Les points gris foncés représentent la classe + et les points gris clairs représentent la classe −.
Nous allons visualiser avec le programme suivant à taper :

```
def creer_points(nb,dim,coul):
       pts=[]
       while len(pts) <nb:
           x = randint(0,dim)
           y = randint(0,dim)
           c = coul[randint(0,1)]
           if (x,y,c) not in pts :
                pts.append((x,y,c))
      return pts
```

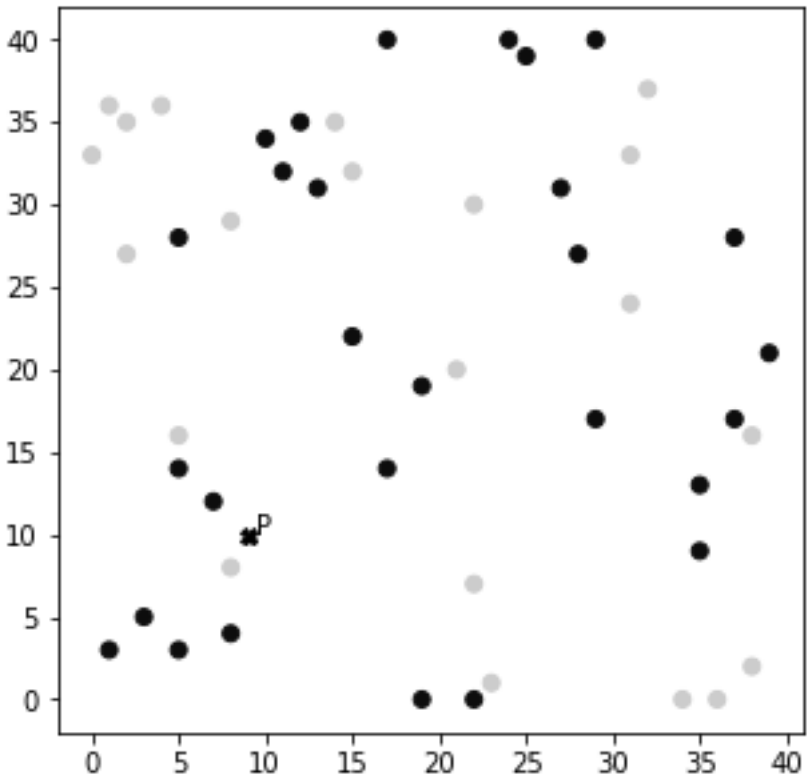
On a créé une liste de points dans le plan avec $0 \leqslant x \leqslant$ `dim` et $0 \leqslant y \leqslant$ `dim` et `c` désigne la couleur de (x, y) qui sera un niveau de gris (ici clair ou foncé). Tapez alors :

```
couleurs=["0.05", "0.8"]
points = creer_points(50,40,couleurs)
points

  [(10, 34, '0.05'), (24, 40, '0.05'), (34, 0, '0.8'),
  ...
  (39, 21, '0.05'), (5, 16, '0.8'), (2, 35, '0.8')]
```

On va créer un point P aléatoire et tracer le nuage `points` en visualisant P. On tape :

```
import matplotlib.pyplot as plt
P = (randint(0,40), randint(0,40),"k")
x = [p[0] for p in points]
y = [p[1] for p in points]
c = [p[2] for p in points]
plt.scatter(x,y,linestyle='None',color=c, marker="o");
plt.plot(P[0],P[1],P[2]+"X"); plt.text(P[0]+0.4,P[1],"P");plt.show()
```

Le point `P` n'appartient pas au nuage de points mais on peut l'y associer en affectant par exemple à `P` la couleur dominante (gris clair ou gris foncé) parmi ses k voisins les plus proches par l'algorithme des k plus proches voisins. C'est ce que l'on fera à l'exercice **2-3**. Ainsi si la couleur dominante est le gris foncé, on attribuera le gris foncé à `P` par prédiction. Mais `P` est peut-être en réalité gris clair.



On ajoute ainsi des points dont on connaît la couleur, foncée ou claire. On vérifie alors pour chaque point ajouté si sa couleur est bien celle qui est prévue ou pas par le modèle de prédiction.
On obtient quatre sortes de points : des vrais foncés et des vrais clairs (la prédiction est alors exacte) et des faux foncés et des faux clairs (la prédiction est inexacte).

		Prédiction de foncés	**Prédiction de clairs**
		Foncé	Clair
Foncés réels	Foncé	*vrais foncés*	*faux clairs*
Clairs réels	Clair	*faux foncés*	*vrais clairs*

Avec 200 points ajoutés, par exemple 150 points foncés et 50 points clairs, on peut obtenir une matrice comme : $\begin{pmatrix} 137 & 13 \\ 4 & 46 \end{pmatrix}$, appelée *matrice de confusion*.

Dans un problème de prédiction avec plus de deux classes, le principe est le même.
De manière générale, une matrice de confusion est construite ainsi :

- chaque ligne correspond à une classe réelle;
- chaque colonne correspond à une classe estimée.

Donc à l'intersection d'une ligne i et d'une colonne j, on trouve le nombre d'éléments de la classe réelle i qui ont été estimés comme appartenant à la classe j.

Exemple 2

Donnons un exemple de matrice de confusion à trois classes : $\begin{pmatrix} 178 & 13 & 9 \\ 7 & 137 & 6 \\ 3 & 5 & 92 \end{pmatrix}$.

Il y a $178 + 13 + 9 = 200$ de la classe `c = 0` réellement, $7 + 137 + 6 = 150$ de la classe `c = 1` réellement, $3 + 5 + 92 = 100$ de la classe `c = 2` réellement.
Puis, il y a 178 points réellement de la classe `c = 0` à qui on prédit qu'ils sont de la classe `c = 0`.
Le taux de prédictions correctes de la classe `c = 0` est donc $178/200 = 0.89$.

Le programme suivant `tpc` permet de renvoyer ce taux de prédictions correctes de la classe `c` avec `c` entier entre 0 et 2. Si `mc` est la matrice de confusion alors `mc[c][c]` donne le nombre de prédictions correctes de la classe `c`. On tape :

```
def tpc(mc,c):
    n = len(mc)
    nbc = 0
    nbpc = mc[c][c]
    for j in range(n):
        nbc = nbc + mc[c][j]
    return nbpc/nbc
```

On l'applique à notre matrice de confusion et aux différentes classes.

```
mc = [[178,13,9],[7,137,6],[3,5,92]]

tpc(mc,2), tpc(mc,1), tpc(mc,0)
    0.92   0.9133333333333333  0.89

tpc(mc,3)

Traceback (most recent call last):
    File "<ipython-input-53-bbaca20b13f2>", line 1, in <module>
    tpc(mc,3)
    File "<ipython-input-48-fdb413bcaa44>", line 4, in tpc
    nbpc = mc[c][c]
IndexError: list index out of range
```

Le `tpc(mc,3)` c'est juste pour voir.
Enfin, $3+5+92 = 100$ et $92/100 = 0.92$ puis $7+137+6 = 150$ et $137/150 = 0.9133333333333333$.
Donc c'est OK.

Intéressons nous now plutôt aux taux de prédictions incorrectes. Intéressons nous pour fixer les idées à la classe `c = 0` par exemple. Il y a 7 points pour lesquels on a prédit `c = 0` alors que c'est réelement `c = 1` et 3 points où on a prédit `c = 0` alors que c'est réelement `c = 2`. Donc cela fait 10 points où on prédit `c = 0` de façon éronée. Le nombre de points qui sont réellement de classe `c = 1` est $7 + 137 + 6 = 150$ et le nombre de points qui sont réellement de classe `c = 2` est $3 + 5 + 92 = 100$. Le nombre de points qui ne sont pas réellement de la classe `c = 0` est donc 250. Le rapport $10/250 = 0.04$ donne un taux de prédiction incorrecte de la classe `c = 0`.
Le programme suivant permet de faire ce rapport pour toutes les classes.

```
def tpi(mc,c):
       n = len(mc)
       nbi = 0 # total classes i differentes de c
       nbpi = 0 # total predictions incorrectes de c
       for i in range(n):
            if i != c :
                 for j in range(n):
                      nbi = nbi + mc[i][j]
                 nbpi = nbpi + mc[i][c]
       return nbpi/nbi

tpi(mc,0), tpi(mc,1), tpi(mc,2)
   (0.04, 0.06, 0.04285714285714286)

18/300
    0.06
15/350
    0.04285714285714286
```

Vérifier à la main dans le cas `c = 0` ce programme.

On a d'autres instruments de mesure dans une matrice de confusion. Par exemple :

- **Le taux d'erreurs**. C'est le nombre de prédictions incorrectes sur le nombre total de prédictions. On vise une valeur la plus proche de 0. Dans le cas de l'exemple 2, ce taux vaut $(7 + 3 + 13 + 5 + 9 + 6)/450 = 43/450 = 0.095555$.
- **La précision**. C'est le nombre de prédictions correctes sur le nombre total de prédictions. On vise une valeur la plus proche de 1. Si TE est le taux d'erreurs et P la précision, alors $P = 1 - TE$. Dans le cas de l'exemple 2, P vaut $1 - 43/450 = 0.9044444$.

Revenons au cas général.
Pour calculer la matrice de confusion, il faut disposer d'un ensemble de données à tester et de l'ensemble des résultats attendus. On compare les résultats attendus avec les résultats prédits sur les données à tester. Pour cela, on utilise un ensemble de données classées. Cet ensemble est partagé en deux-sous ensembles, l'un constituant **l'ensemble d'apprentissage** et l'autre **l'ensemble de test**. On peut choisir comme dans l'exemple 3 qui suit, les proportions 3/4 et 1/4. L'ensemble d'apprentissage est utilisé pour prédire la classe de chaque élément de l'ensemble de test avec l'algorithme des k plus proches voisins.

Exemple 3

Étape 01

On crée un ensemble E de points du plan appartenant à deux classes 0 et 1 visualisées par des disques noirs ou des carrés noirs en utilisant le code suivant. On rappelle que `random()` renvoie un flottant aléatoire compris entre 0 et 1. Et `random() < val` crée une fonction booléene.

```
from random import random
random()
     0.0976439354494626
random() <0.9
    True
random()> 0.9
    False
```

Puis, on crée un ensemble de points sous forme d'une liste de triplets (x, y, c), où x et y sont des réels pris aléatoirement entre `-dim` et `dim` puis c est la classe 0 ou 1 choisi alátoirement avec `random()`.

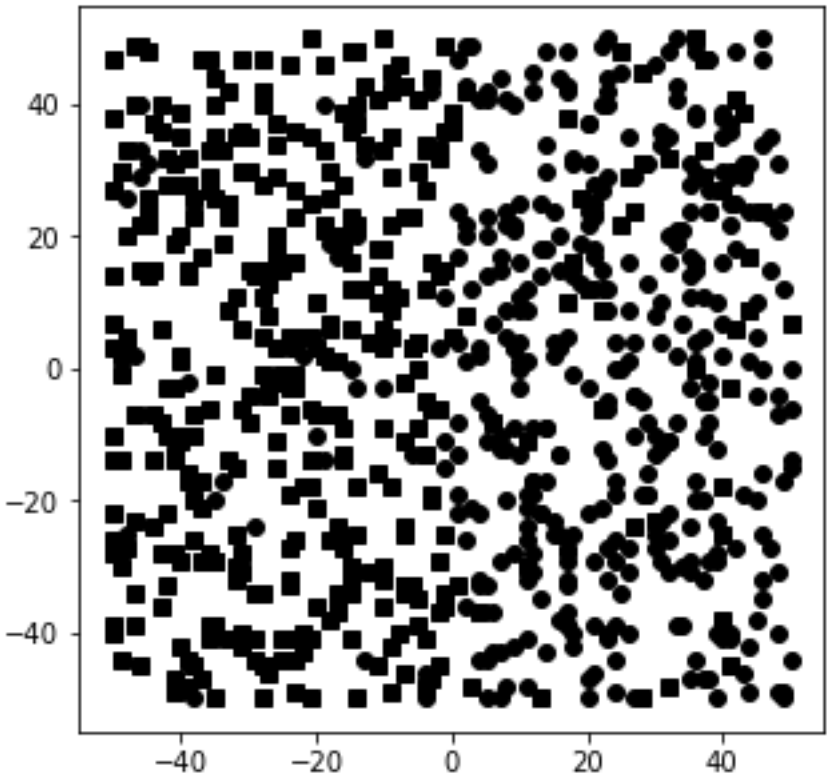
```
from random import randint
nbpoints, dim = 800, 50
classes = [0,1]
def points(n,dim) :
     ens = []
     pts = []
     while len(ens) < n:
           x = randint(-dim,dim)
           y = randint(-dim, dim)
               if x <= 0:
                   cl = classes[random() < 0.9]
               else :
                   cl = classes[random() >= 0.9]
               ens.append((x,y,cl))
     return ens
```

Faisons tourner.

```
E = points(nbpoints,dim)
E

 [(-35, -27, 1), (-32, 26, 1), ...
 (39, 42, 0), (28, 30, 0), (-13, -11, 1), (-33, 44, 1)]
```

Visualisons tout ça. Le code `plt.recParams` qui est un objet de type dictionnaire en passant, permet d'ajuster des paramètres. Ici, on définit la taille des points que l'on va dessiner. Puis dans `plt.plot`, l'attribut `color = "k"` fait dessiner en noir les points, puis `marker = "o"` fournit des disques et `marker = "s"` fournit des carrés. Donc ici les points de classe 0 seront des disques noirs et les points de classe 1 seront des carrés noirs.

```
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (5,5)

for p in E:
    if p[2] == 0 :
        plt.plot(p[0],p[1],color="k", marker= "o")
    else :
        plt.plot(p[0],p[1],color="k", marker= "s")
```



Étape 2.
On va scinder E en deux ensembles, l'ensemble de points d'apprentissage `ens_appr` et un ensemble de test `ens_test`. On va respecter la proportion 75% et 25%. L'idée est de reprendre `points` et de le retoucher. On l'appellera `new_points` pour commencer.

On scinde `ens` en deux ensembles `ens_appr` et `ens_test` dans le corps de la procédure.
Puis on remplace `len(ens)` par `len(ens_appr) + len(ens_test)`.
Puis dans la boucle `while`, après la création de `cl`, on tape `choix = randint(0,3)` et donc dans `choix`, on met aléatoirement $0, 1, 2$ ou 3. Le code `choix == 0` donne donc le quart des choix. Et on remplit `ens_test` avec ce choix et sinon on remplit `ens_appr`.
À la fin, on renvoie `ens_appr` et `ens_test`, donc deux listes distinctes.

```
def new_points(n,dim):
    ens_appr = []
    ens_test = []
    pts = []
    while len(ens_appr) + len(ens_test) < n:
        x = randint(-dim, dim)
        y = randint(-dim,dim)
        if x <= 0 :
            cl = classes[random() < 0.9]
        else :
            cl = classes[random() >= 0.9]
        choix = randint(0,3)
        if choix == 0 and len(ens_test) < n/4:
            ens_test.append((x,y,cl))
        elif choix > 0 and len(ens_appr) < 3*n/4:
            ens_appr.append((x,y,cl))
    return ens_appr, ens_test
```

Faisons un exemple avec $n = 10$ pour bien voir.

```
new_points(10,50)

  ([(-29, -18, 1), (-50, -26, 1), (-17, -23, 0), (-15, -2, 1),
  (14, 16, 1), (13, -32, 0), (-37, 10, 1), (-4, -22, 0)],
  [(11, 4, 0), (-11, 4, 1)])
```

On remarque que `ens_appr` a 8 termes et `ens_test` a 2 termes. Ce n'est pas tout à fait 3/4 et 1/4 mais il ne faut pas oublier que `randint` c'est aléatoire et donc plus n est grand, plus le 3/4, 1/4 est à peu près respecté.

Étape 3.
Création de la matrice de confusion.
On utilise les fonctions `d`, `tri` et `knn` inspirées de ce que l'on a plus haut pour l'algorithme des k plus proches voisins. On les rappelle si vous devez les retaper.

```
def d(p1,p2):
     (x1,y1,c1) = p1
     (x2,y2,c2) = p2
     return (x1-x2) ** 2 + (y1-y2) ** 2
def tri(E,P,d):
    def choix(elt):
        return elt[1]
    distances = [ (p,d(p,P)) for p in E]
    return sorted(distances, key = choix)
def knn(E,p,d,k):
    pts = tri(E,p,d)
    return [elt[0] for elt in pts[0:k]]
```

Puis on tape la procédure `matrice(A,T,k,d)`, où `A` est l'ensemble d'apprentissage et `T` l'ensemble de tests. On utilisera `A,T = new_points(nbpoints, dim)` pour récupérer `A` et `T`.

```
def matrice(A,T,d,k):
      conf = [[0,0],[0,0]]
      for p in T :
           vs = knn(A,p,d,k)
           cpt = [0,0]
           for v in vs :
                if v[2] == 0:
                    cpt[0] += 1
                else :
                    cpt[1] += 1
           if cpt[0] > cpt[1] :
                indice = 0
           elif cpt[0] < cpt[1] :
                indice = 1
           else :
                indice = randint(0,1)
           estime = classes[indice]
           if estime == p[2] :
                if estime == 0 : # VRAI POSITIF
                     conf[0][0] += 1
                else : # VRAI NEGATIF
                     conf[1][1] += 1
           else :
                if estime == 0 : # FAUX POSITIF
                     conf[1][0] += 1
                else : # FAUX NEGATIF
                     conf[0][1] += 1
      return conf
```

Comprenons cette procédure :
1) au départ, on prend pour matrice de confusion appelée `conf` la matrice nulle d'ordre 2.
2) Puis pour chaque point `p` fixé de `T`, l'ensemble des tests, on tape `vs = knn(A,p,k,d)` qui donne les k voisins les plus proches de $p \in T$ pris parmi les éléments de `A`, l'ensemble d'apprentissage.
On obtient donc une liste de k triplets du type (x, y, c) avec $c = 0$ ou $c = 1$.
3) On compte les $c = 0$ et les $c = 1$ de cette liste `vs` en rentrant dans la liste `cpt` le nombre de $c = 0$ et le nombre de $c = 1$.
4) Puis on récupère dans `indice` l'indice de la couleur la plus représentée.
Dans le cas où `cpt[0] == cpt[1]`, on s'en remet au dieu random avec `indice = randint(0,1)`
5) On met dans `estime` la valeur `classes[indice]` qui est donc la couleur la plus représentée parmi nos k voisins de `p`.
La matrice `conf` est de la forme $\begin{pmatrix} \text{Vrai +} & \text{Faux -} \\ \text{Faux +} & \text{Vrai -} \end{pmatrix}$ (voir l'exemple 1).
On suppose que la classe `c=0` est celle des positifs et la classe `c=1` est celle des négatifs.
Si `estime == p[2]` et `estime == 0`, cela signifie que l'on a un vrai positif. En effet, `p[2]` fournit la vraie valeur qui est la classe 0. Et `estime` donne aussi 0.
Si par contre, `estime == p[2]` et `estime == 1`, cela signifie que l'on a un vrai négatif. En effet, la vraie classe est 1 (donc un négatif) et `estime` donne aussi 1.
Si par contre `estime not== p[2]` et `estime == 0`, cela signifie que l'on a un faux positif. En effet, `p[2]` est alors 1, et on a un négatif et pourtant `estime` dit qu'il est positif.
Si `estime not== p[2]` et `estime == 1`, cela signifie que l'on a un faux négatif. En effet, `p[2]` est alors 0, on a affaire à un positif et pourtant le classe parmi les négatifs.

```
A,T = new_points(nbpoints, dim)
mat_conf = matrice(A,T,d,3)

mat_conf

   [[85, 12], [11, 92]]
```

On commence par remarquer que $85 + 12 + 11 + 92 = 200$. C'est le quart de `nbpoints=800` donc c'est bien la proportion voulue car tous les points comptés dans cette matrice sont uniquement les points de `ens_test`.
Tapons pour finir la procédure `taux_erreur`.

```
def taux_erreur(m):
      erreurs = m[0][1] + m[1][0]
      predictions = erreurs + m[0][0] + m[1][1]
      return erreurs/predictions

m = [[85,12],[11,92]]
taux_erreur(m)
     0.115
```

■ Apprentissage non supervisé

Algorithme des k-moyennes : de quoi s'agit-t-il ?

On dit aussi **clustering algorithm**. On regroupe des données possédant des éléments de similarité dans des groupes. L'ensemble des données étudiées doit pouvoir être encodé dans un ensemble de vecteurs ou une matrice.
À partir d'un jeu de données, on définit un nombre k de groupes ou classes. L'algorithme permet d'analyser le jeu de données et d'affecter chaque donnée à une classe, des données similaires ou proches appartenant alors à une même classe. Dans un apprentissage supervisé, on essaie de trouver une corrélation entre les valeurs des données d'un ensemble et une valeur à prédire. Ici, on essaie de trouver des ressemblances entre les données pour constituer une partition.
Pour déterminer la proximité entre deux données, on utilise une mesure de similarité ou une distance. Nous utiliserons la distance euclidienne $d(X,Y) = \sqrt{\left(\sum_{i=1}^{n} (x_i - y_i)^2\right)}$.

L'objectif est de partitionner l'ensemble des données en groupes (ou clusters) distincts, chaque groupe étant représenté par une donnée centrale dont les autres données du groupe sont proches. Ces centres peuvent être choisis au hasard parmi les données à l'initialisation. Chaque itération consiste à affiner le partitionnement. Pour cela, le centre de chacun des k groupes est ré-évalué par le calcul du barycentre de l'ensemble des données du groupe. C'est de là que vient le nom d'algorithme des k moyennes puisqu'à chaque itération k moyennes sont calculées. Les données sont alors regroupées en utilisant les nouveaux centres.

Warning : les centres calculés à chaque itération ne sont plus en général des éléments de l'ensemble des données.

Pour résumer :

- Un ensemble de données est fourni sous forme de matrice, le nombre de groupes k est choisi.
- Initialisation de manière aléatoire des centres de chaque groupe.
- Répéter les deux étapes :
  regrouper chaque donnée dans la partie définie par le centre le plus proche;
  remplacer chaque centre par le barycentre des données de son groupe.

Soit le nombre d'itérations est fixé à l'avance, soit les centres ne sont plus modifiés lors d'une itération et l'algorithme a alors trouvé une partition stable de l'ensemble des données : on dit que l'algorithme a convergé.

Le choix du nombre de groupes k est important. Il est nécessaire de tester plusieurs valeurs. Si k est trop petit, les données dans chaque groupe risque de ne pas avoir une similarité forte, et si k est trop grand, chaque groupe ne permet pas de découvrir des caractéristiques intéressantes.

Exemple

Un fruit de forme arrondie possède de nombreuses caractéristiques dont le poids et le rapport grand axe/petit axe. Les valeurs utilisées dans le programme sont aléatoires et les cerises sont plutôt grosses.

Étape 01 : création du jeu de données

Un traitement des données est effectué. Il est nécessaire afin que la caractéristique poids ne soit pas trop prévalente. Ici l'ensemble des données est noté `E` et il y a `k=3` groupes. Dans les étapes suivantes, pour conserver la généralité et adapter dans d'autres exemples, dans les procédures `genere_centres`, `partitionne` et `k_moyennes`, on prendra pour arguments les lettres `E` et `k` et non leurs valeurs.

```
from random import randrange, choice, random

E = []
classes_fruits = {}
k=3    # Nombre de classes

for i in range(50):
      poids = 65 + 20 * random()
      rapport = 1.2 + 0.20 * random()
      E.append([poids, rapport])
      classes_fruits[i] = "mandarine"

for i in range(50,100):
      poids = 85 + 30 * random()
      rapport = 1.4 + 0.4 * random()
      E.append([poids, rapport])
      classes_fruits[i] = "kiwi"

for i in range(100,400):
      poids = 10 + 5 * random()
      rapport = 1 + 0.20 * random()
      E.append([poids, rapport])
      classes_fruits[i] = "cerise"

for elt in E:
      elt[1] = 100 *elt[1]
def d(p1,p2):
     return (p1[0]-p2[0])**2 + (p1[1] - p2[1])**2

E[0:5]
  [[75.86078914576923, 134.14448987615725],
 [83.67408121188569, 121.96558970927552],
 [76.30788312031973, 120.46897920304187],
 [82.83900713275025, 136.02710420467895],
 [74.84143226834124, 130.96360604974902]]
E[48:52]
[[75.03393068205956, 127.49644192598353],
 [66.36588321546508, 122.29631153910394],
 [100.68243818256286, 172.53653599760003],
 [88.43714662901907, 163.1368537164303]]
E[98:104]
[[92.78301461131325, 149.97338776724902],
 [100.42416911047346, 172.89241817528114],
 [14.150911251377163, 119.50627300346821],
 [11.630382309749942, 110.665021051174],
 [10.058103764037662, 106.28150059271017],
 [11.842895971664955, 114.60364946718724]]
```

Étape 02 : création des centres des classes

On rappelle que `randrange(n)` renvoie un nombre entier entre 0 et $n-1$.

```
def genere_centres(E,k):
       n = len(E)
       liste = []
       for i in range(k):
            x = randrange(n)
            while x in liste :
                 x = randrange(n)
            liste.append(x)
       return liste

centres = genere_centres(E,k)
centres
   [386, 36, 328]
```

Étape 03 : création des barycentres pour transformer les centres

Une fonction **barycentre** détermine les nouveaux centres à calculer à chaque itération. Le point appelé barycentre d'un ensemble de points est plus précisement l'isobarycentre de ces points.

```
barycentre(points):
    n = len(points)
    m = len(points[0])
    s = [0] * m
    for p in points:
           for k in range(m):
               s[k] = s[k] + p[k]
    s = [s[k]/n for k in range(m)]
    return s

barycentre([[3,2] ,[4,1],[-1,0]])
    [2.0, 1.0]

(3+4-1)/3 , (2+1+0)/3
    (2.0, 1.0)
# C'est OK !
```

Étape 04 : création d'une fonction détection des centres les plus proches

Ici, on crée une fonction `ind_minimum` prend en paramètre une liste de distances entre un point et les centres de chaque groupe. Elle renvoie une liste contenant l'indice du centre le plus proche ou les indices des centres les plus proches (un point peut être équidistant de plusieurs centres).

```
def ind_minimum(dist):
        mini = min(dist)
        choix = []
        for i in range(len(dist)):
             if dist[i] == mini:
                  choix.append(i)
        return choix
ind_minimum([1.5,3,0.45,4]), ind_minimum([1.5,3,0.45,4,0.45])
       [2],  [2, 4]
```

On remarque donc que `ind_minimum` peut renvoyer une liste de plusieurs entiers.

Étape 05 : création de la partition initiale

La fonction `partitionne` qui va suivre crée la partition initiale. On retournera 3 données, la première `partition` donne une liste de trois listes (car $k = 3$ ici) qui donne une partition des fruits par leurs numéros. La seconde `new_centres` donne les centres sous forme de liste de 3 listes de longueurs 2, calculés avec la fonction **barycentre**. Enfin, la troisième est `classes` et qui donne la classe (entier $0, 1, 2$) de chaque numéro de fruit.
Noter l'usage de `choice(liste)` qui renvoie un élément de `liste` choisi aléatoirement. Ainsi `choix = choice(ind_minimum(di))` donne un des entiers de `ind_minimum(di)` choisi aléatoirement. En général, `di` a une seule valeur et le choix est vite fait !

```
def partitionne(ens, centres, distance):
      n = len(ens)
      k = len(centres)
      partition = [[c] for c in centres]
      classes = {centres[i]: i for i in range(k)}
      for i in range(n):
           if i not in centres :
               di = [distance(ens[i], ens[k]) for k in centres]
               choix = choice(ind_minimum(di))
               classes[i] = choix
               partition[choix].append(i)
      new_centres = [None] * k
      for i in range(k):
            points = [ens[j] for j in partition[i]]
            new_centres[i] = barycentre(points)
      return partition, new_centres, classes
```

Faisons now tourner `partitionne(E, centres, d)` mais pour rendre plus visible, on va séparer `partition`, `c` et `classes`.
On tape : `partition, c, classes = partitionne(E,centres,d)` puis on tape `c`, on fait Enter, puis on tape `partition[0]`, on fait Enter, puis on tape `partition[1]`, on fait Enter, puis on tape `partition[2]`, on fait Enter et enfin on tape `classes` et on fait Enter.

```
partition , c, classes = partitionne(E,centres,d)
c
[[12.291546972365632, 115.18165824251905],
 [87.31490977424329, 144.48256947424764],
 [12.717679188352552, 105.42253229808598]]

partition[0]
 [386,100,101, 103,107, ... ,388,389,390,391,394,396,398]
partition[1]
 [36,0,1,2, 3,...,95, 96,97,98,99]
partition[2]
[328,102,104,105, 106,110, 111,115, 119, ..., 393,395,397, 399]
classes
{386: 0,36: 1, 328: 2, 0: 1,1: 1,2: 1,3: 1, ...,100: 0,101: 0,102: 2,
 103: 0,104: 2,105: 2,..., 391: 0,392: 2, 393: 2,394: 0,395: 2,
 396: 0,397: 2,398: 0,399: 2}
```

Étape 06 : création et exécution de la fonction algorithme des k-moyennes

Notons là `k_moyennes`.

```
def k_moyennes(ens,centres, distance, max_iter):
     n = len(E)
     k = len(centres)
     partition , c, classes = partitionne(ens,centres, distance)
     iteration = 0  # ITERATIONS
     evolution = True
     while evolution and (iteration < max_iter) :
            evolution = False
            iteration = iteration + 1
            for i in range(n):
                 di = [distance(ens[i],k) for k in c]
                 choix = choice(ind_minimum(di))
                 if choix != classes[i]:  # MODIFICATION
                      partition[classes[i]].remove(i)
                      classes[i] = choix
                      partition[choix].append(i)
                      evolution = True
           for i in range (k): # CALCUL DES CENTRES
                 points = [ens[j] for j in partition[i]]
                 c[i] = barycentre(points)
     return partition , c
```

```
k_moyennes(E,centres,d,12)
  ([[386,100,101,103,107,...,326,358,127],
  [36,0,1, 2,3,4,...,96,97, 98,99],
  [328,102,104,106,...,387,392,393,395,397,399]],
  [[12.453171836462957, 114.55225531885658],
  [87.31490977424329, 144.48256947424764],
  [12.597918797216167, 104.60694308896458]])
```

```
# Interessons nous aux centres obtenus par iteration only
partition, c = k_moyennes(E,centres,d,0)
c
 [[12.291546972365632, 115.18165824251905],
 [87.31490977424329, 144.48256947424764],
 [12.717679188352552, 105.42253229808598]]
```

```
partition, c = k_moyennes(E,centres,d,1)
c
 [[12.433330629086598, 114.82249651342866],
 [87.31490977424329, 144.48256947424764],
 [12.609494073394732, 104.93418057878941]]
```

```
partition, c = k_moyennes(E,centres,d,2)
 c
 [[12.450391073965603, 114.582098122653],
 [87.31490977424329, 144.48256947424764],
 [12.600062904636333, 104.6438749719822]]
```

```
partition, c = k_moyennes(E,centres,d,12)
c
 [[12.453171836462957, 114.55225531885658],
 [87.31490977424329, 144.48256947424764],
 [12.597918797216167, 104.60694308896458]]
```

```
partition, c = k_moyennes(E,centres,d,40)
c
  [[12.453171836462957, 114.55225531885658],
  [87.31490977424329, 144.48256947424764],
  [12.597918797216167, 104.60694308896458]])
```

Énoncé des exercices

Exercice 2.1 : Trier des points

On dispose d'un ensemble de points dans l'espace muni d'un repère orthonormé d'origine O. Un point possède des coordonnées reprśentées par une liste du type `[x,y,z]`. L'objectif est de trier ces points en fonction de leur distance (au carré) à un point donné P, de la plus petite à la plus grande.

1. Écrire une fonction `distance` qui prend en argument deux listes représentant les coordonnées de deux points quelconques et renvoie le carré de la distance euclidienne entre ces deux points.

2. Écrire une fonction `compare` qui prend en paramètres trois listes `p,p1,p2` représentant dans l'ordre le point P et deux points quelconques P_1, P_2 et qui renvoie -1 si P_1 est plus proche de P que P_2, 1 si P_2 est plus proche de P que P_1 et 0 si les deux points sont équidistants de P.

On pose :

```
liste = [[1, 1, -1], [2, 0, 0], [1, -1, 1], [2, 1, -1]]

P = [0,0,0]
```

Calculer `compare(P,liste[i],liste[j])` pour tout i et j possibles avec $i < j$.

3. On considère la procédure suivante qui a pour argument un point `p` et une liste nommée `new_liste` composée de listes de trois points, qui applique compare à `p` et à deux points quelconques de `new_liste` et qui renvoie la liste `new_liste` avec les points triés par ordre croissant des distances entre ces points et le point `p`.

```
def tri_points(p,new_liste):
    for i in range(len(new_liste)-1) :
        i_mini = i
        mini = new_liste[i]
        for j in range(i+1, len(new_liste)):
            if compare(p,mini, new_liste[j]) == 1:
                i_mini = j
                mini = new_liste[j]
        new_liste[i], new_liste[i_mini] = new_liste[i_mini],
            new_liste[i]
    return new_liste
```

L'appliquer à `tri_points(P,liste)` de la question **2.**

Retrouver le résultat avec ce que vous avez trouvé à la question **2.**

Exercice 2.2 : 1. Taper les instructions suivantes :

```
In [1]: example_ens = [1,6,3,4,5,0,2,-2,-7, 45]
In [2]: d = {elt : False for elt in example_ens}
In [3]: d
```

Puis tapez et commentez :

```
In [4]: from random import randrange ; d[example_ens[0]] = 1
In [5]: d

In [6]: i = randrange(1,len(example_ens))
In [7]: i

In [8]: d[example_ens[i]] = 2
In [9]: example_ens[i], example_ens[1] = example_ens[1], example_ens[i]
In [10]: d

In [11]:example_ens
```

2. On considère la procédure suivante :

```
In [12]: def distribute_incomplete(ens,k):
    ...:     n = len(ens)
    ...:     d = { elt : False for elt in ens}
    ...:     d[ens[0]] = 1
    ...:     for p in range(1,k) :
    ...:         i = randrange(p,n)
    ...:         d[ens[i]] = p+1
    ...:         ens[i], ens[p] = ens[p], ens[i]
    ...:     return d
```

On pose ici :

```
In [13]: example_ens = [1, 45, 3, 4, 5, 0, 2, -2, -7, 6]
# Puis on tape :
In [14]: distribute_incomplete(example_ens,4)
# On obtient :
Out[15]:
{1:1,45: False,3: False,4: False,5: False,0: False,2: 2,-2:False,-7:3,6:4}
```

Faire à la main ce que la machine a fait presque spontanément ! Que vaut `example_ens` après la boucle ?

3. Écrire alors une fonction `distribute` qui prend en paramètres un ensemble donné `ens` sous la forme d'une liste et un nombre entier `k` non nul qui reprend `distribute_incomplete` en y ajoutant une boucle `for elt in ens[k:n]:` dans laquelle on mettra `d[elt] = randrange(1,k+1)` qui règle le problème des éléments de `d` restés `False`.

Tapez ensuite `distribute(example_ens,2)` et `distribute(example_ens,5)`.
Que fait donc `distribute` ?

4. Écrire une fonction `unterteilen` qui prend en arguments un ensemble donné `ens` sous la forme d'une liste et un entier non nul `k` et qui renvoie la partition (nommée `parties` dans la procédure) en `k` parties de l'ensemble `ens` (contenant au moins `k` éléments) générée aléatoirement à l'aide de la fonction `distribute`.

Indication : on commencera par affecter à la variable `d` le dictionnaire `distribute(ens, k)`.
Puis on rentre dans `parties` une liste remplie de `k` fois la liste vide `[]`.
Puis on fera une boucle `for elt in d:` et dans cette boucle, on affectera chaque `elt` dans la liste de `parties` qui est en position `d[elt] - 1`.

5. Appliquer `unterteilen` à `examples_ens` avec `k=3`, `k=6` et `k=1`.

Exercice 2.3 : 1. L'idée ici est de faire un dessin qui ressemble à la figure créé quand on a parlé de matrice de confusion. On avait eu un nuage de points avec deux niveaux de gris. On désire plus précisément dans cet exercice avoir un nuage de points avec trois niveaux de gris.
Reprenez la procédure `creer_points` d'arguments `nb`, `dim` et `coul` qui renvoie maintenant une liste de triplets (x, y, c) avec `c = randint(0,2)` et donc `c` peut prendre trois valeurs au lieu de deux.
Appliquer avec :

`couleurs = ["0.01", "0.5", "0.8"]` et `creer_points(50,40,couleurs)`

On tapera ensuite `points = creer_points(50,40,couleurs)`
2. On crée ici un point `P` comme dans le cours et on veut tracer le nuage de points où apparaîtra `P`. On tapera donc :

```
In [1]: import matplotlib.pyplot as plt
In [2]: P = (randint(0,40), randint(0,40),"k")
In [3]: x = [p[0] for p in points]
In [4]: y = [p[1] for p in points]
In [5]: c = [p[2] for p in points]
In [6]: plt.scatter(x,y, linestyle='None', color=c, marker="o");
plt.plot(P[0],P[1], P[2] + "X"); plt.text(P[0]+0.4,P[1],"P"); plt.show()
```

3. On va utiliser l'algorithme des k plus proches voisins afin d'affecter à `P` la couleur dominante parmi ses $k = 4$ voisins les plus proches. On cherche donc à prédire le niveau de gris de `P`.
a. Construire une procédure `distance(p1,p2)` qui renvoie le carré de la distance des points `(x1,y1,c1) = p1` et de `(x2,y2,c2)= p2`
b. Construire une procédure `tri(E,P,d)` et `knn(E,p,d,k)` comme dans le cours.
c. On pose `k = 4` et `vs = knn(points,P,distance,k)`. Taper ces codes et vérifier que `vs` donne une liste cohérente.
Taper ensuite la fonction `couleur_maj` d'arguments `vs` et `coul` qui détermine la couleur majoritaire parmi les couleurs des voisins ou plutôt l'indice de cette couleur majoritaire :

```
In [1]: def couleur_maj(vs,coul):
            cpt = [0] * len(coul)
            for v in vs :
                for i in range(len(coul)):
                    if v[2] == coul[i] :
                        cpt[i] += 1
            ind_maxi = 0
            for i in range(len(cpt)):
                if cpt[i] > cpt[ind_maxi]:
                    ind_maxi = i
            return ind_maxi
```

Tapez alors `couleurs[couleur_maj(vs,couleurs)]`

Indications

Corrigé des exercices

Exercice 2.1

1.

```
def distance(p1,p2):
    return (p2[0]-p1[0])**2+(p2[1]-p1[1])**2+(p2[2]-p1[2])**2
distance([1,1,-1],[2,0,0])
    3
```

2.

```
def compare(p,p1,p2):
       d1 = distance(p1,p) ; d2 = distance(p2,p)
       if d1 < d2 :
            return -1
       elif d2 < d1 :
            return 1
       else :
            return 0
```

```
P = [0,0,0] ; compare(P,liste[0],liste[1])
   -1
compare(P,liste[0],liste[2])
    0
compare(P,liste[0],liste[3])
    -1
compare(P,liste[1],liste[2])
    1
compare(P,liste[1],liste[3])
   -1
compare(P,liste[2],liste[3])
   -1
```

Donc si l'on pose : `d_i = distance(P,liste[i])` alors $d_0 < d_1$, $d_0 = d_2$, $d_0 < d_3$, $d_2 < d_1$ et $d_1 < d_3$.
Soit $d_0 = d_2 < d_1 < d_3$.

3.

```
tri_points(P,liste)
     [[1, 1, -1], [1, -1, 1], [2, 0, 0], [2, 1, -1]]
```

Exercice 2.2

1.

```
In [1]: example_ens = [1,6,3,4,5,0,2,-2,-7,45]
In [2]: d = {elt : False for elt in example_ens}
In [3]: d
Out[3]: {1: False, 6: False, 3: False, 4: False, 5: False, 0:
    False,
         2: False,-2: False, -7: False, 45: False}
In [4]: d[example_ens[0]]
Out[4]: False
In [5]: d[example_ens[0]] = 1
In [6]: d
Out[6]: {1: 1, 6: False, 3: False, 4: False, 5: False, 0: False,
         2: False, -2: False, -7: False, 45: False}
In [7]: from random import randrange
In [8]: i = randrange(1,len(example_ens))
In [9]: i
Out[9]: 9
In [10]: d[example_ens[i]] = 2
In [11]: d
Out[11]: {1: 1, 6: False, 3: False, 4: False, 5: False, 0: False,
         2: False, -2: False, -7: False, 45: 2}
In [12]: example_ens[i], example_ens[1] = example_ens[1],
    example_ens[i]

In [13]: d
Out[13]: {1: 1, 6: False, 3: False, 4: False, 5: False, 0: False,
         2: False, -2: False, -7: False, 45: 2}
# d n'a pas ete modifie
In [14]: example_ens[i]
Out[14]: 6
In [15]: i
Out[15]: 9
In [16]: example_ens
Out[16]: [1, 45, 3, 4, 5, 0, 2, -2, -7, 6]
# example_ens a ete modifie
```

2.

```
In [17]: def distribute_incomplete(ens,k):
    ...:     n = len(ens)
    ...:     d = { elt : False for elt in ens}
    ...:     d[ens[0]] = 1
    ...:     for p in range(1,k) :
    ...:         i = randrange(p,n)
    ...:         d[ens[i]] = p+1
    ...:         ens[i], ens[p] = ens[p], ens[i]
    ...:     return d
    ...:

In [18]: distribute_incomplete(example_ens,4)
Out[18]:
{1: 1,
 45: False,
 3: False,
 4: False,
 5: False,
 0: False,
 2: 2,
 -2: False,
 -7: 3,
 6: 4}
In [19]: example_ens
Out[19]: [1, 2, -7, 6, 5, 0, 45, -2, 3, 4]
```

Ici `example_ens = [1,45,3,4,5,0,2,-2,-7,6]` Nommons `ens` pour simplifier cette liste.
On commence à `d[ens[0]` de mettre `1` et de plus `len(example_ens) = 10`
Puis on commence la boucle avec `k = 4` donc les `p` vont de 1 à 3.
On prend `p = 1` et `i = randrange(1,10)` et `d[ens[i]] = 2` On voit dans le `d` retourné à la fin que `i = 6` a été choisi. Enfin, on permute `ens[6]` et `ens[1]` soit 2 et 45 dans `ens`.
On prend `p = 2` et `i = randrange(2,10)` et `d[ens[i]] = 3` On voit dans le `d` retourné à la fin que `i = 8` a été choisi. Enfin, on permute `ens[8]` et `ens[2]` soit -7 et 3 dans `ens`.
On prend `p = 3` et `i = randrange(3,10)` et `d[ens[i]] = 4` On voit dans le `d` retourné à la fin que `i = 9` a été choisi. Enfin, on permute `ens[9]` et `ens[3]` soit 6 et 4 dans `ens`.
On obtient bien `[1, 2, -7, 6, 5, 0, 45, -2, 3, 4]` à la fin.

3.

```
from random import randrange
def distribute(ens,k):
    n = len(ens)
    d = { elt : False for elt in ens}
    d[ens[0]] = 1
    for p in range(1,k):
        i = randrange(p,n)
        d[ens[i]] = p+1
        ens[i],ens[p] = ens[p],ens[i]
    for elt in ens[k:n] :
        d[elt] = randrange(1,k+1)
    return d
```

```
distribute(example_ens,2)
  {1: 1, 6: 1, 3: 1, 4: 1, 5: 2, 0: 2, 2: 2, -2: 1, -7: 2, 45: 1}
distribute(example_ens,5)
  {1: 1, -7: 1, 3: 4, 4: 2, 5: 4, 0: 3, 2: 2, -2: 5, 6: 4, 45: 5}
```

`distribute` génère aléatoirement une partition en k parties de l'ensemble contenant au moins k éléments.

4.

```
def unterteilen(ens,k):
    d = distribute(ens,k)
    parties = [[] for i in range(k)]
    for elt in d :
        parties[d[elt] -1].append(elt)
    return parties
```

Explication : Si `elt` appartient à une classe notée `1`, alors `d[elt] = 1` et donc `parties[d[elt] -1]` = parties[0] et donc on met `elt` dans la première sous-liste, celle d'indice 0, etc.

5.

```
unterteilen(example_ens,3)
   [[1, 5, 6], [3, -2, -7, 45], [0, 4, 2]]
unterteilen(example_ens,6)
   [[1, 3], [0], [-7, 45], [4, 6], [5], [-2, 2]]
unterteilen(example_ens,1)
   [[1, 0, -7, 4, 5, -2, 3, 2, 6, 45]]
```

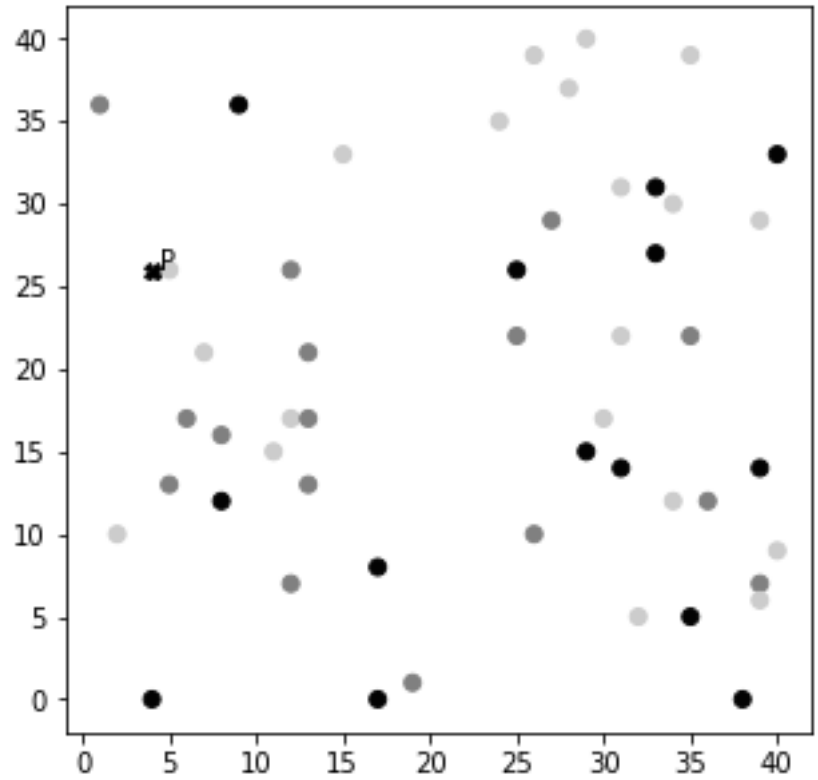
Exercice 2.3

1.

```
from random import randint
def creer_points(nb,dim,coul):
     pts=[]
     while len(pts) <nb:
         x = randint(0,dim)
         y = randint(0,dim)
         c = coul[randint(0,2)]
         if (x,y,c) not in pts :
             pts.append((x,y,c))
    return pts

couleurs = ["0.01", "0.5 ", "0.8"]
creer_points(50,40,couleurs)

  [(36, 5, '0.8'), (9, 19, '0.01'), ...
  ... (14, 38, '0.01'), (1, 38, '0.01')]

points = creer_points(50,40,couleurs)
```

2.

```
import matplotlib.pyplot as plt
P = (randint(0,40), randint(0,40),"k")
x = [p[0] for p in points]
y = [p[1] for p in points]
c = [p[2] for p in points]

plt.scatter(x,y, linestyle='None', color=c, marker="o");
plt.plot(P[0],P[1], P[2] + "X");
plt.text(P[0]+0.4,P[1],"P"); plt.show()
```



3.
a.

```
def distance(p1,p2):
      (x1,y1,c1) = p1
      (x2,y2,c2) = p2
      return (x1-x2) ** 2 + (y1-y2) ** 2

distance((1,2,0),(4,5,7))
   18
```

b.

```
def tri(E, P, d):
       def choix(elt):
             return elt[1]
       distances = [(p,d(p,P)) for p in E]
       return sorted(distances, key = choix)
```

```
def knn(E,p,d,k):
      pts = tri(E,p,d)
      return [elt[0] for elt in pts[0:k]]
k = 4
vs = knn(points, P, distance, k)
vs
  [(22, 21, '0.5 '), (23, 25, '0.5 '), (20, 20, ' 0.8 '), (25,
     23, '0.5 ')]
```

c.

```
def couleur_maj(vs,coul):
     cpt = [0] * len(coul)
     for v in vs :
          for i in range(len(coul)):
               if v[2] == coul[i] :
                    cpt[i] += 1
     ind_maxi = 0
     for i in range(len(cpt)):
          if cpt[i] > cpt[ind_maxi]:
              ind_maxi = i
     return ind_maxi
```

```
choix_couleur = couleurs[couleur_maj(vs,couleurs)]
choix_couleur
     '0.5 '
```

On trouve un gris moyen. Si l'on recommence :

```
points=creer_points(50,40,couleurs)
knn(points,P,distance,4)
   [(23, 23, ' 0.8 '), (24, 23, ' 0.8 '), (25, 20, '0.01'), (21,
      17, '0.5 ')]
choix_couleur = couleurs[couleur_maj(knn(points,P,distance,4),
   couleurs)]
choix_couleur
    ' 0.8 '
```

On trouve un gris foncé. Donc la prédiction reste aléatoire.

graphics graphicx pstricks,pst-node tikz

Chapitre 3

Algorithms and games

Objectifs

- **Les incontournables :**
  - savoir

Résumé de cours

■ Introduction

Dans la recherche sur les jeux, on y étudie en particulier des notions de stratégie et d'équilibre. La recherche informatique s'est aussi développée avec les jeux sur les graphes qui ont des applications dans plusieurs domaines d'informatique théorique et des mathématiques en permettant de modéliser certains problèmes.
À partir d'une situation donnée, des joueurs à tour de rôle prennent une décision parmi un ensemble fini de décisions possibles, chaque décision amenant à une nouvelle situation. Nous considérons principalement dans la suite des **jeux à deux joueurs** satisfaisant certaines conditions :

- chaque joueur a la même vue d'ensemble de la situation (jeu à information complète);
- une décision est prise en fonction de la situation présente et pas des situations passées (jeu sans mémoire);
- cette décision ne dépend que de la situation et pas du joueur (jeu impartial);
- dans une situation donnée, une décision amène toujours à la même nouvelle situation (jeu sans hasard).

Les jeux sont orientés par des graphes orientés finis. Une situation, ou une position, est un sommet du graphe. Une décision est le choix d'un arc amenant à une nouvelle position. Dans un **jeu d'accessibilité**, chaque joueur souhaite atteindre un sous-ensemble de sommets particulier en se déplaçant à tour de rôle sur le graphe. Un jeu d'accessibilité à deux joueurs est modélisé par un **graphe biparti**.

■ Graphes bipartis

Définition Un graphe G est biparti si l'ensemble de ses sommets peut être divisé en deux sous-ensembles disjoints G_1 et G_2 tels que chaque arête de G a une extrémité dans G_1 et une extrémité dans G_2.

Exemple 1

FIGURE 1

On remarque dans la **figure 1** que $G_1 = \{1, 2, 3\}$ et $G_2 = \{4, 5, 6, 7\}$.

On peut employer aussi une technique de coloration ou utiliser la notion de cycle.

Exemple 2

FIGURE 2 + FIGURE 3

En effet, on démontre (théorème admis) qu'un graphe est biparti si et seulement s'il ne contient pas de cycles de longueurs impaires.

Dans la **figure 2**, la coloration du graphe est possible avec seulement deux couleurs sans que deux sommets voisins n'aient la même couleur. Le graphe possède un cycle pair $ABFEA$ (le nombre d'arcs constituant le cycle est pair).

Dans la **figure 3**, une coloration avec deux couleurs sans que deux sommets voisins n'aient la même couleur) est impossible. Et le graphe possède un cycle impair $ABEA$.

Donnons un programme pour tester si un graphe possède un cycle impair. On utilise une file. Et en particulier un `deque` du package `collection`. Le mot `deque` vient de double-ended queue, c'est une généralisation des piles et des files. On peut faire des ajouts et retraits à chaque extremité contrairement aux files qui sont FIFO (first in first out) et piles qui sont LIFO (last in first out). La commande `popleft()` retourne l'élément le plus à gauche.

```
from collections import deque
def cycle_impair(graphe, sommet):
    niveaux = {s : None for s in graphe}
    niveaux[sommet] = 0
    file = deque()
    file.append(sommet)
    while len(file) > 0 :
          sommet = file.popleft()
          for s in graphe[sommet]:
               if niveaux[s] is None :
                   niveaux[s] = niveaux[sommet] + 1
                   file.append(s)
               elif niveaux[s] == niveaux[sommet]:
                   return True
    return False
```

On teste les graphes de la **figure 2** et **figure 3**. Les graphes sont représentés par des dictionnaires.

```
g1 = {'A' : ['B','E'] , 'B' : ['A', 'C', 'D', 'F' ], 'C' : ['B'], 'D' :
    ['B'] , 'E' : ['A', 'F'], 'F' : ['B', 'E' ]}
cycle_impair(g1,'A')
    False
```

```
g2 = {'A' : ['B','E'] , 'B' : ['A', 'C', 'D', 'E', 'F' ], 'C' : ['B'],
   'D' : ['B'] , 'E' : ['A', 'B', 'F'], 'F' : ['B', 'E' ] }
cycle_impair(g2,'A')
    True
```

■ Jeux à deux joueurs

Définitions, vocabulaire

On considère un jeu à deux joueurs J_1 etJ_2 sur un graphe orienté fini $G = (S, A)$, où S est l'ensemble fini des sommets et A l'ensemble des arcs (ou arêtes). Chaque joueur possède son ensemble de sommets, respectivement S_1 et S_2. On appelle **graphe du jeu** ou **arène** le triplet (G, S_1, S_2). Les ensembles S_1 et S_2 constituent une partition de S : $S_1 \cap S_2 = \emptyset$ et $S_1 \cup S_2 = S$.

On dit que les sommets de S_1 sont contrôlés par J_1 et les sommets de S_2 sont contrôlés par J_2.

Le graphe du jeu est évidemment biparti : une arête joint un sommet contrôlé par un joueur avec un sommet contrôlé par l'autre joueur.

On marque chaque sommet par un entier (chaque sommet est étiqueté) $0, 1, 2, ...$ Une partie est alors un chemin a priori infini dans le graphe (on suppose les deux joueurs immortels). Un jeton est placé sur un sommet initial. Le joueur qui contrôle ce sommet déplace le jeton selon une arête qu'il a choisit jusqu'à un sommet voisin et c'est l'autre joueur qui prend le relai dans le déplacement du jeton et ainsi de suite.

Dans un jeu d'accessibilité, un joueur J_1 gagne lorsqu'il atteint un sous-ensemble de sommets $F \subset S$. Un sommet de F est un sommet final. L'ensemble des sommets de F sont les états gagnants pour le joueur J_1. Il y a bien entendu un sous-ensemble de sommets constitué des états gagnants pour l'autre joueur et un sous-ensemble de sommets consitué des états de partie nulle (aucun des joueurs ne perd ni ne gagne).
Un jeu est dit **à somme nulle** si toute partie gagnée par l'un des deux joueurs est perdue par l'autre joueur.

Un jeu d'accéssibilité à somme nulle est alors un quadruplet (G, S_1, S_2, F), où (G, S_1, S_2) est une arène et F l'ensemble des sommets gagnants pour J_1. Si W est l'ensemble des chemins contenant un sommet de F, on peut définir le jeu par le quadruplet (G, S_1, S_2, W). L'ensemble W est appelé **condition de gain**.

Exemple 1

FIGURE 4

Ici, on a des cercles pour les sommets contrôlés par J_1 et des carrés pour les sommets contrôlés par J_2.
Un jeton est déposé sur le sommet 0 qui est contrôlé par J_1. C'est donc J_1 qui commence et il peut déplacer le jeton soit jusqu'au sommet 1, soit jusqu'au sommet 2. Ces deux sommets sont contrôlés par J_2 qui commence alors.
Une partie est par exemple le chemin

0 1 3 1 4 1 3 1 4 1 3 1 4 ...

On remarque que sur ce chemin, on reste sur un cycle pair.

Le joueur J_1 gagne la partie s'il atteint un sommet d'un certain sous-ensemble F de S. Dans ce cas, le joueur J_2 perd la partie. Pour que J_2 ne perde pas la partie, il faut que le chemin ne passe pas par un des sommets de F.

Exemple 2 : jeu de Nim

Le jeu de Nim remonte à la chine antique où on l'appelait sous le nom de **fan-fan** et en Afrique sous le nom de **tiouk-tiouk**. Le nom actuel est une déformation de Nimm ! = Prends !

Le jeu de Nim se joue avec plusieurs tas d'objets identiques, classiquement des allumettes (ou des jetons, des pièces etc.) Chaque joueur à tour de rôle choisit un tas et le nombre d'objets, au moins un, qu'il retire de ce tas. Le dernier joueur à retirer des objets a gagné la partie.
Il existe différentes variantes, par exemple prenons le cas d'un seul tas de 15 objets et chaque joueur peut retirer 1, 2 ou 3 objets du tas.

On peut faire le cheminement suivant :

- 15 objets dans le tas à cette étape : J_1 joue et retire 3 objets ;
- 12 objets dans le tas à cette étape : J_2 joue et retire 2 objets ;
- 10 objets dans le tas à cette étape : J_1 joue et retire 2 objets ;
- 8 objets dans le tas à cette étape : J_2 joue et retire 3 objets ;
- 5 objets dans le tas à cette étape : J_1 joue et retire 1 objet ;
- 4 objets dans le tas à cette étape : J_2 joue et retire 1 objet ;
- 3 objets dans le tas à cette étape : J_1 joue et retire 3 objets et gagne la partie.

La stratégie qui permet à J_1 de gagner est de laisser à J_2 un multiple de 4 objets dans le tas.

Exemple 3 : jeu de Marienbad

Ce jeu est en fait la somme de quatre jeux de Nim à un seul tas mais en version inverse, c'est-à-dire que le joueur qui prend le dernier objet a perdu la partie.
Exercice. *Les quatre tas t_A, t_B, t_C et t_D sont constitués respectivement de* 1, 3 , 5 *et* 7 *allumettes. Chaque joueur peut retirer à tour de rôle des allumettes, au moins une, dans un seul tas. Le joueur qui prend en dernier a perdu.*
Illustrer par des dessins où à chaque étape on visualise les quatre tas. Au départ du jeu, le joueur J_1 prend 3 *allumettes dans t_D. Puis ensuite J_2 prend* 2 *allumettes dans t_B. Puis ensuite J_1 prend* 3 *allumettes dans t_C. Puis ensuite J_2 prend* 2 *allumettes dans t_D. Puis J_1 prend les* 2 *dernieres allumettes de t_D. C'est à J_2 de jouer. Peut-il encore gagner ou est-ce cuit pour lui ?*

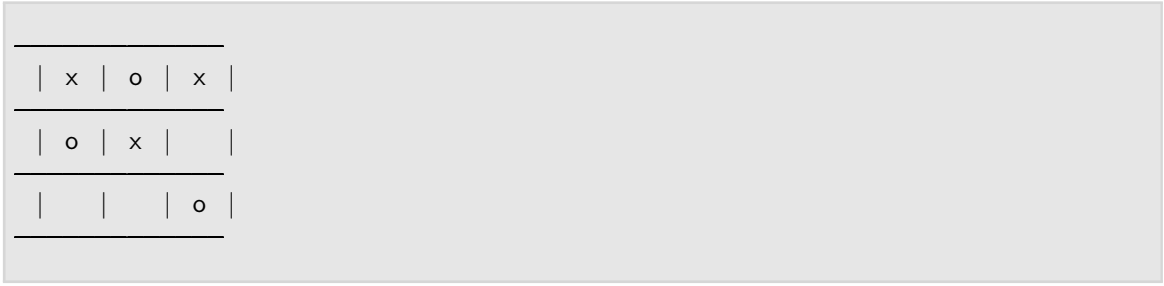
Exemple 4 : jeu tic-tac-toe ou encore jeu du morpion ou jeu oxo

Un symbole **x** ou **o** est attribué à chaque joueur. Les deux joueurs tracent alternativement dans une case vide d'une grille carrée de neuf cases leur symbole. Le gagnant est le premier à obtenir une ligne horizontale, verticale ou en diagonale de trois symboles identiques.
Ce jeu a plusieurs variantes. La grille peut etre plus grande, un joueur doit aligner 5 symboles identiques pour marquer un point etc.

Donnons un programme affichant le jeu de base à 9 cases. On fait la barre verticale avec Alt Gr 6

```
jeu = {i: " " for i in range(9)}
def affiche(jeu):
        print(13 * "—")
        print(" | "+ jeu[0] + " | " + jeu[1] + " | " + jeu[2] + " | ")
        print(13* "—")
        print(" | "+ jeu[3] + " | " + jeu[4] + " | " + jeu[5] + " | ")
        print(13 * "—")
        print(" | "+ jeu[6] + " | " + jeu[7] + " | " + jeu[8] + " | ")
        print(13* "—")
```

```
jeu[0] ="x";jeu[3] ="o" ; jeu[4] ="x" ; jeu[8]="o" ; jeu[2]="x"; jeu
    [1]="o"
affiche(jeu)
```



Exemple 5 : jeu de chomp ou du chocolat

Le jeu a été inventé en 1952 par Fred Schuh en termes de choix de diviseurs à partir d'un entier donné puis réinventé par David Gate sous sa forme actuelle. to chomp = mâcher.

Une tablette de chocolat est composée de n rangées horizontales et m rangées verticales. Si la tablette a pour dimension n et m, chaque carré est représenté par le couple (p, q) avec $1 \leqslant p \leqslant n$ et $1 \leqslant q \leqslant m$.
On suppose que le carré $(1, 1)$ ne doit pas être mangé car il est empoisonné. On le met en gris. L'idée est que chaque joueur à tour de rôle mange des carrés, ce qui réduit la tablette à chaque étape. Maintenant il y a plusieurs façons de réduire la tablette en enlevant des carrés. Donnons pour commencer le protocole classique du jeu de chomp.

Protocole 1 (le classique) : Un joueur choisit un carré et retire tous les carrés qui se trouvent à droite et en dessous au sens large, c'est-à-dire qu'il retire aussi le carré (a, b). On dit qu'il mange (ou chomp) les carrés enlevés. Ainsi si le carré choisi est (a, b), le joueur retire tous les carrés (p, q) tels que $p \geqslant a$ et $q \geqslant b$. Si après le coup d'un joueur, il ne reste que le carré $(1, 1)$, comme on ne peut pas manger un carré empoisonné, ce joueur est déclaré gagnant car l'autre joueur est bloqué et a donc perdu.

***Exercice** : dessiner une tablette de 20 carrés avec $n = 4$ et $m = 5$. Le joueur J_1 choisit le carré $(3, 5)$. Retirer les carrés adéquats et dessiner la tablette restante. Puis le joueur J_2 choisit le carré $(5, 2)$. Retirer les carrés adéquats. Puis J_1 choisit $(1, 2)$. Retirer les carrés adéquats. Puis J_2 joue et choisit $(2, 1)$. Conclusion ?*

Protocole 2 On fait une autre variante du jeu précédent. Un joueur coupe suivant une verticale (respectivement une horizontale) et mange les rangées à droite de la verticale (respectivement en dessous de l'horizontale). Le premier joueur qui obtient après sa coupe une tablette carrée a gagné dans cette variante.

■ Stratégie

Une stratégie permet de préciser ce qui doit être joué dans chaque situation. Élaborer une stratégie peut être très compliqué et on se contente de stratégies qui ne prennent en compte que le sommet où se trouve le jeton et qu'on appelle **stratégie positionnelle**.

Sur un graphe, une stratégie pour un joueur J_i est une fonction γ qui a tout sommet s de S_i associe un sommet de S avec $(s, \sigma(s)) \in A$. Cette fonction précise le sommet à atteindre depuis la position s. Un joueur J_i respecte une stratégie si pour toute partie $s_0 s_1 s_2 ...$ et tout $k \geqslant 0$, si $s_k \in S_i$ alors $s_{k+1} = \sigma(s_k)$.

Exercice : *reprenons le jeu de chomp avec le protocole 1 et supposons que J_1 a pour stratégie de choisir au départ le carré $(2, 2)$. En déduire les choix possibles pour J_2 puis J_1 quand c'est de nouveau son tour. Qui va gagner ?*

■ Stratégie gagnante

Dans un jeu d'accessibilité, le joueur J_1 gagne une partie si la partie contient un sommet d'un ensemble F donné. On dit qu'une stratégie est gagnante pour J_1 depuis un sommet s_k si pour toute partie contenant s_k la partie est gagnée par J_1 s'il respecte la stratégie depuis s_k, et ceci, quelle que soit la stratégie suivie par l'autre joueur. On dit alors que s_k est une **position gagnante** pour J_1.
Une stratégie est gagnante pour J_1 si la stratégie est gagnante pour J_1 depuis s_0.
Voir l'exercice précédent où le choix de $(2, 2)$ comme s_0 est une stratégie gagnante pour J_1.

Donnons un autre exemple.
Exemple Considérons la figure suivante dans lequel les sommets en forme de cercle sont controlés par J_1 et les sommets en forme de carré par J_2.

FIGURE 5

On pose un jeton sur le sommet marqué 0, contrôlé par J_1 et la partie débute sur ce sommet. Chacun son tour, les deux joueurs déplacent le jeton en suivant une arête. Une partie est gagnée par J_1 si le jeton arrive sur le sommet marqué 6.
Une stratégie gagnante pour J_1 est de déplacer le jeton sur le sommet 1. Le joueur J_2 ne peut alors déplacer le jeton que sur l'un des sommets 3 ou 4. Le joueur J_1 peut alors tranquillement aller au sommet 6 au tour suivant.

■ Algorithme min-max

Introduction de l'algorithme mini-max : position du problème

Considérons l'exemple de jeu de Nim à un tas. On suppose ici que le tas initial a 5 objets. Chaque joueur peut retirer un, deux ou trois objets du tas. Le premier qui ne peut plus retirer un objet perd la partie. Il n'y a pas de partie nulle.
On peut représenter le jeu par le graphe biparti qui suit avec à gauche les sommets contrôlés par J_1 et ¨ a droite ceux contrôlés par J_2.
Les sommets sont étiquetés par le nombre d'objets restants.

FIGURE 6

Une autre manière de représenter le jeu est de faire un arbre de décision. Les sommets de l'arbre appelés **noeuds** sont des positions et les arêtes sont les coups joués par les joueurs. Un noeud particulier est la **racine** de l'arbre. Les **enfants** d'un noeud n sont les noeuds correspondant aux positions qui suivent la position correspondant à n. Ce sont les racines des sous-arbres de n. Les feuilles de l'arbre sont les noeuds correspondants aux positions finales. Les feuilles n'ont pas d'enfants, pas de sous-arbres. Le nombre de feuilles est égal au nombre de parties différentes. Pour le jeu en exemple, l'arbre obtenu est représenté avec les sommets contrôlés par J_2 en gris.

FIGURE 7

Remarque : On remarque qu'une case blanche avec 0 signifie que J_1 ne peut plus rien retirer et il a perdu (il y a 6 telles cases) et qu'une case grise avec 0 signifie que J_2 ne peut plus rien retirer et il a perdu (il y a 7 telles cases). Cela signifie que J_1 a plus de chance de gagner que J_2. Et cela sans que J_1 démarre avec une stratégie gagnante. Donc J_1 a un gros avantage d'être en premier.

On peut parcourir l'arbre, comme on parcourt un graphe, pour chercher à partir de chaque position le meilleur coup à jouer.
On utilise pour cela un algorithme comme l'algorithme **min-max** ou encore appelé **minimax**.

Principe de l'algorithme mini-max

On se place du côté d'un joueur, par exemple J_1 (car on a vu qu'il a plus une tête de winner que le joueur J_2).
Chaque feuille de l'arbre indique si J_1 est gagnant ou perdant. On ajoute donc une marque ou une valeur au noeud, $+1$ s'il est gagnant et -1 s'il est perdant.
Dans un jeu où il peut y avoir des parties nulles, on ajoute 0. Ensuite, on définit les valeurs pour les autres noeuds de manière récursive.

- Si le noeud est contrôlé par J_1, sa valeur est le maximum des valeurs de ses sous-arbres.
- Si le noeud est contrôlé par J_2, sa valeur est le minimum des valeurs de ses sous-arbres.

Le but est pour J_1 de maximiser ses gains et pour J_2 de minimiser ses pertes.

Exercice : *Reprendre l'arbre de la figure* 7*, les feuilles étant les cases numérotés* 0*, commencer par affecter les feuilles blanches de* -1 *(*J_1 *a perdu) et on écrit dans la case alors* $0 : -1$ *et les feuilles grises de* $+1$ *(*J_1 *a gagné) et on écrit alors dans la case* $0 : +1$*. Puis suivre l'algorithme définit plus haut en remplaçant un noeud numérotée* n *par* $n : -1$ *ou* $n : +1$ *puis en remontant jusqu'à la racine qui est une case blanche numérotée* 5*. Dans cette case met-on* $5 : +1$ *ou* $5 : -1$ *?*

L'objectif est de déterminer une stratégie ou des positions gagnantes. Si un noeud a une valeur positive, alors J_1 dispose à partir de cette position d'une stratégie pour gagner la partie. Sur l'exemple J_1 a une stratégie gagnante depuis la position initiale.

Énoncé des exercices

Exercice 3.1 : Graphe biparti ou non

1. Dessiner un graphe dont les 10 sommets sont des ronds numérotés de 0 à 9, que l'on placera de façon à peu près équidistants sur un grand cercle de telle manière que le rond marqué 1 soit en haut sur le cercle et qu'on mette les ronds 2, 3, 4, 5, 6, 7, 8, 9 et 0 dans le sens inverse du sens trigonométrique.
Les arêtes (uniques entre chaque rond) sont telles que :
Le rond 0 est relié avec les ronds $1, 8$ et 9.
Le rond 1 est relié avec les ronds $0, 3, 4, 7$.
Le rond 2 est relié avec les ronds $3, 5$ et 7.
Le rond 3 est relié avec les ronds $1, 2, 6$ et 9.
Le rond 4 est relié avec les ronds $1, 6, 8$ et 9.
Le rond 5 est relié avec les ronds $2, 8$ et 9.
Le rond 6 est relié avec les ronds $3, 4$ et 7.
Le rond 7 est relié avec les ronds $1, 2, 6$ et 8.
Le rond 8 est relié avec les ronds $0, 4, 5$ et 7.
Le rond 9 est relié avec les ronds $0, 3, 4$ et 5.

2. Ce graphe est-il biparti ? On pourra utiliser la méthode de coloration.

3. Définir un dictionnaire représentant ce graphe et utiliser la procédure `cycle_impair` du cours pour vérifier votre résultat.

Exercice 3.2 : Jeu de Nim avec un seul tas et quatre objets Chaque joueur retire un ou deux objets du tas à tour de rôle. Le dernier joueur qui retire des objets a gagné la partie.

1. Définir un graphe `G` sous forme d'un dictionnaire dont chaque clé, représentant un sommet, est un couple composé du numéro du joueur (1 ou 2) qui contrôle le sommet et du nombre d'objets restant dans le tas. Les valeurs associées aux clés sont des listes de couples où chaque couple représente un sommet après la prise du joueur qui contrôle le sommet représenté par la clé.

Le premier joueur est le joueur 1 et il y a 4 objets. Le sommet de départ est donc représenté par $(1, 4)$. Le joueur 1 peut retirer un ou deux objets et les deux sommets atteints sont contrôlés par le joueur 2. Alors `G = (1,4):[(2,3),(2,2)],...`

2. On veut retourner par une procédure le dictionnaire `G` demandé à **1**.

a. Pour cela, on commence par construire une procédure `nim(n,j,G)` récursive où `n` est le nombre de sommets (donc d'objets), `j` est le numéro d'un joueur (1 ou 2) et `G` le graphe que l'on initialise à `G =` ensuite. On remarque si `j` est un joueur alors `1+j%2` est l'autre joueur.
Ainsi dans la boucle principale de `nim(n,j,G)`, si `if n>1:`, on tape :

```
G[(j,n)] = [(1+j%2,n-2),(1+j%2,n-1)]
nim(n-1,1+j%2,G)
nim(n-2,1+j%2,G)
```

Il reste à faire le cas `elif n == 1:` et le cas `else :`

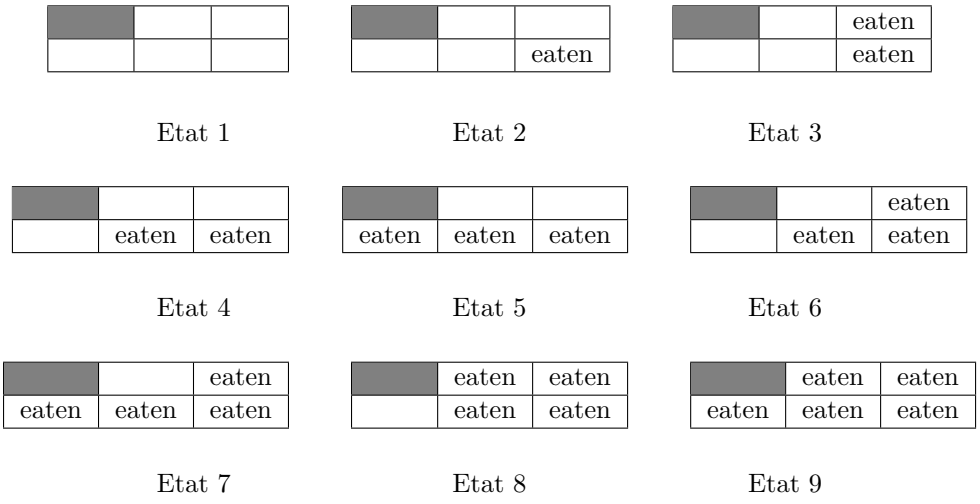
b. Taper ensuite :

```
def jeu(n):
    G={}
    nim(n,1,G)
    return G
```

Retrouver le graphe `G` pour $n = 4$ de la question **1**.

Exercice 3.3 : Jeu de chomp

On considère le jeu de chomp avec une tablette composée de 2 lignes et 3 colonnes. Le carré $(1,1)$ est grisé et empoisonné. Les différents états de la tablette sont numérotés.



Pour simplifier le graphe et sauver les joueurs, on considère qu'un joueur ne peut pas manger le carré emposonné. Donc l'état 9 est un état final. Le joueur qui se trouve devant cet état a perdu la partie car il ne peut plus jouer.

1. Dessiner le graphe du jeu. On pourra faire deux colonnes. Un sommet est un couple avec pour premier élément le numéro du joueur et pour second élément létat de la tablette. Dans la première colonne, on mettra les huits sommets `(1,i)` contrôlés par J_1 les uns sous les autres en commençant par `(1,1)` et dans la deuxième colonne, on mettra les huits sommets `(2,i)` contrôlés par J_2 les uns sous les autres commençant par `(2,2)`. Puis on placera les différentes arêtes entre les deux colonnes.

Remarque : *attention à deux notations identiques pour deux types d'objets. Le sommet* `(i,j)` *désigne que le joueur numéro* i *est à l'étape* j *et le carré* (i,j) *désigne le carré de la tablette qui est à la ligne* i *et colonne* j.

2. En déduire le graphe `G` du jeu sous la forme d'un dictionnaire.

3. On suppose que J_1 depuis le sommet `(1,1)` mange le carré $(2,3)$ et atteint donc le sommet `(2,2)`. Au tour de J_2. Justifier que J_2 n'a que quatre possibilités. Et prévoir ce que fera J_1 à chaque fois pour gagner en un coup ou en deux coups suivants. Conclure que l'on a trouvé une stratégie gagnante pour J_1.

Indications

Corrigé des exercices

Exercice 3.1

1. A FAIRE

2. On colore les sommets $1, 2, 6, 8$ et 9 en gris. On voit que pour deux sommets reliés par une arête un est grisé et l'autre reste blanc.

3. On retape le code.

```
from collections import deque
def cycle_impair(graphe, sommet):
    niveaux = {s : None for s in graphe}
    niveaux[sommet] = 0
    file = deque()
    file.append(sommet)
    while len(file) > 0 :
          sommet = file.popleft()
          for s in graphe[sommet]:
               if niveaux[s] is None :
                    niveaux[s] = niveaux[sommet]
                       + 1
                    file.append(s)
               elif niveaux[s] == niveaux[sommet
                  ]:
                    return True
    return False
```

```
g= {0: [1,8,9], 1 : [0,3,4,7], 2: [3,5,7], 3 :
    [1,2,6,9], 4: [1,6,8,9], 5: [2,8,9], 6 :
    [3,4,7], 7 :[1,2,6,8], 8 : [0,4,5,7], 9 :
    [0,3,4,5]}
cycle_impair(g,0)
   False
```

Exercice 3.2

1. On obtient :

$\{(1,4) : [(2,2),(2,3)], (2,3) : [(1,1),(1,2)], (1,2) : [(2,0),(2,1)], (2,1) : [(1,0)], (1,0) : [], (2,0) : [], (1,1) : [(2,0)], (2,2) : [(1,0),(1,1)]\}$

2.

a.

```
def nim(n,j,G):
    if n > 1:
        G[(j,n)] = [(1+j%2,n-2),(1+j%2,n-1)]
        nim(n-1,1+j%2,G)
        nim(n-2,1+j%2,G)
    elif n == 1:
        G[(j,n)] = [(1+j%2,n-1)]
        nim(n-1,1+j%2,G)
    else:
        G[(j,n)] = []
```

b.

```
def jeu(n):
    G={}
    nim(n,1,G)
    return G
```

```
jeu(4)

{(1, 4): [(2, 2), (2, 3)],
 (2, 3): [(1, 1), (1, 2)],
 (1, 2): [(2, 0), (2, 1)],
 (2, 1): [(1, 0)],
 (1, 0): [],
 (2, 0): [],
 (1, 1): [(2, 0)],
 (2, 2): [(1, 0), (1, 1)]}
```

Exercice 3.3

1. VOIR FEUILLE

2. IDEM

3. On suppose que J_1 depuis le sommet `(1,1)` mange le carré $(2,3)$ et atteint donc le sommet `(2,2)`. Au tour de J_2.

• J_2 se déplace en `(1,3)` alors J_1 se déplace en `(2,6)` et J_2 ne peut jouer que `(1,7)` ou `(1,8)` et J_1 conclut avec `(2,9)`.

• J_2 se déplace en `(1,4)` alors comme le cas précédent, J_1 se déplace en `(2,6)` et J_2 ne peut jouer que `(1,7)` ou `(1,8)` et J_1 conclut avec `(2,9)`.

• J_2 se déplace en `(1,5)` alors J_1 se déplace en `(2,9)` et conclut.

• J_2 se déplace en `(1,8)` alors J_1 se déplace en `(2,9)` et conclut.

Dans tous les cads, on a trouvé une stratégie gagnante pour J_1.