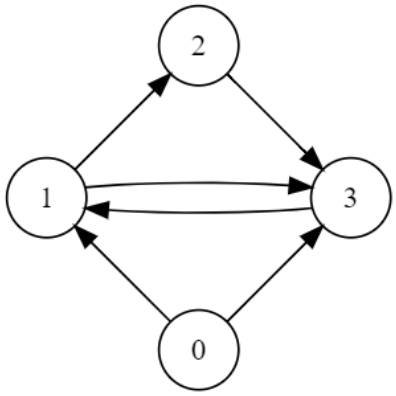
GRAPHES : Implémentation python

1. Interface

- Il existe de multiples façons de représenter un graphe en machine, selon la nature du graphe ou des opérations et des algorithmes à effectuer sur ce graphe.
- Quoi qu'il en soit, on implémentera :
  - Des opérations de constructions (création d'un graphe vide, ajout de sommets, ajout d'arcs, ...)
  - Des opérations de parcours de graphe (par les sommets, par les arcs,...)
- On commencera par les graphes orientés dans lesquels les sommets contiennent des entiers.

2. Matrice d'adjacence

- On suppose dans ce qui suit que les sommets sont des entiers strictement positifs et consécutifs.
- On considère le graphe orienté ci-contre, qui comporte 4 sommets $0, 1, 2, 3$.



- Nous allons représenter ce graphe sous python sous forme d'une liste de listes, qui ne contiendra que des booléens.
- Mathématiquement, cela peut correpondre à la matrice ci-contre :

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

  - La première ligne indique les arcs qui partent du sommet 0. Les sommets d'arrivée sont 1 et 3, ce qui est noté par le nombre 1 à la deuxième et quatrième colonne.
  - La deuxième ligne indique les arcs qui partent du sommet 1, etc...
- En python, cela peut donner la variable `adj` suivante :

```
adj=[[0,1,0,1],
     [0,0,1,1],
     [0,0,0,1],
     [0,1,0,0]]
```

- Ainsi par exemple `adj[0][3]` contient 1, ce qui correspond à l'arc $0 - 3$

Encapsulation dans un objet.

La programmation objet est encore une fois un paradigme intéréssant pour implémenter un tel graphe. La classe `Graphe_M` ci-dessous contient les méthodes suivantes :

- Une méthode constructeur qui prend en argument le nombre de sommets et qui initialise la matrice avec des 0.
- Une méthode `afficher(self)` qui permet de visualiser la matrice d'adjacence du graphe.
- Une méthode `arc(self,s1,s2)` qui renvoie 1 s'il y a un arc de `s1` vers `s2` et 0 sinon.

In [32]:

```
class Graphe_M:
    '''Graphe représenté par une matrice d'adjacence'''
    def __init__(self, n):
        self.n=n
        self.adj= [[0]*n for i in range(n)]

    def affiche(self):
        for i in range(self.n):
            print(self.adj[i])

    def arc(self,s1,s2):
        return self.adj[s1][s2]

    #Ex1
    def ajoute_arc(self, s1, s2):
        pass

    #Ex2
    def voisins(self,s):
        voisins=[]
        pass
        return voisins

    ##Ex3
    def nb_arcs(self):
        a=0
        pass
        return a
    ##Ex9
    def degre(self,s):
        pass
```

In [34]:

```
g1=Graphe_M(4)
g1.affiche() # Affiche la matrice du graphe vide
assert(g1.arc(3,1)==0)
```

```
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

Exercice 1 :

- Dans la classe `Graphe_M` , ajouter la méthode `ajouter_arc(self,s1,s2)` qui crée l'arc de `s1` vers `s2` (c'est à dire qui modifie la matrice en conséquence).
- Le test s'effectue sur le graphe présenté en exemple ci-dessus.

In [35]:

```
#test ajoute_arc()
g1.ajoute_arc(0,1)
g1.ajoute_arc(0,3)
g1.ajoute_arc(1,2)
g1.ajoute_arc(1,3)
g1.ajoute_arc(2,3)
g1.ajoute_arc(3,1)
g1.affiche()

assert(g1.arc(3,1)==1)
```

```
[0, 1, 0, 1]
[0, 0, 1, 1]
[0, 0, 0, 1]
[0, 1, 0, 0]
```

Exercice 2 :

Compléter la méthode `voisins(self, s)` qui prend en paramètre un sommet `s` et qui renvoie la liste de ses voisins, c'est dire la liste des sommets vers lesquels il existe un arc partant de `s` .

In [20]:

```
### tests voisins

print(g1.voisins(2))
print(g1.voisins(1))
```

```
[3]
[2, 3]
```

Exercice 3 :

Ecrire la méthode `nb_arcs(self)` qui renvoie le nombre d'arcs existant dans le graphe passé en argument.

In [21]:

```
# test nb_arcs

assert(g1.nb_arcs()==6)
```

Efficacité

La matrice d'adjacence est simple à mettre en oeuvre, puis à utiliser pour parcourir les sommets ou les arcs d'un graphe. Mais elle comporte quelques défauts :

- L'espace mémoire est important, puisque pour N sommets, il faut $N \times N$ éléments dans la matrice. Ainsi, un graphe de mille sommets (un réseau social en comporte des centaines de millions) nécéssite une matrice avec 1 million d'éléments, même s'il n'y a pas beaucoup d'arcs.
- Le nombre de sommets doit être connu à la création du graphe. L'ajout postérieur d'un sommet nécéssite de réécrire la matrice, ce qui est coûteux en temps.
- Les sommets sont forcément des entiers.
- Nous allons voir par la suite une représentation plus optimale, à l'aide d'un dictionnaire.

3. Dictionnaire d'adjacence

Rappels sur les dictionnaires

- Les dictionnaires ressemblent aux listes python(ils sont modifiables comme elles), mais ce ne sont pas des séquences. Les éléments enregistrés ne sont pas disposés dans un ordre immuable. Il est néanmoins possible d'accéder à n'importe lequel d'entre eux à l'aide d'un index que l'on appellera une clé, laquelle pourra être alphabétique ou numérique.
- Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type(valeurs numériques, chaînes, listes ou encore des dictionnaires, et même aussi des fonctions). ##### Exemple :

In [74]:

```
pays={'nom':'France' , 'cap':'Paris' , 'dep' : ['Ain', 'Aisne']}
print(pays['nom'])
pays['pop']=67
pays['dep'].append('Allier')
for k in pays:
    print (k , pays[k])
```

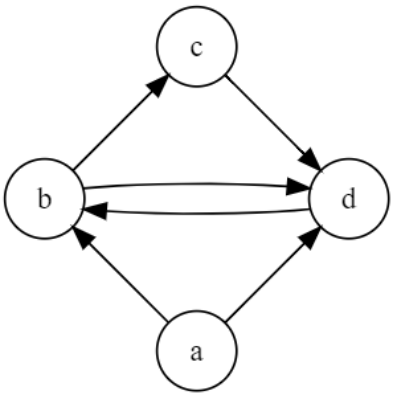
```
France
nom France
cap Paris
dep ['Ain', 'Aisne', 'Allier']
pop 67
```

Remarques :

- Des accolades délimitent un dictionnaire.
- Les éléments d'un dictionnaire sont séparés par une virgule.
- Chacun des éléments est une paire d'objets séparés par deux points : une clé et une valeur.
- La valeur de la clé `'nom'` est `France` .
- La valeur de la clé `'dep'` est une liste.
- Le parcours d'un dictionnaire s'effectue à l'aide des clés.

Pour représenter un graphe

Dans le graphe ci-contre, pour chacun des sommets, on peut au choix lister :



- Ses successeurs (ses voisins), sa représentation est alors :

```
{ 'A' : ['B','D'],
  'B : ['C','D'],
  'C' : ['D'],
  'D' : ['B']
}
```

- Ses prédecesseurs (les sommets dont il est voisin) :

```
{ 'A' : [],
  'B : ['A','D'],
  'C' : ['B'],
  'D' : ['A','B','C']
}
```

Encapsulation dans un objet

La nouvelle classe `Graphe_D` ci-dessous contient désormais les méthodes suivantes :

- Une méthode constructeur qui ne prend pas d'argument mais qui initialise le graphe avec un dictionnaire vide.
- Une méthode `afficher(self)` qui permet de visualiser le dictionnaire d'adjacence du graphe.
- Une méthode `ajouter_sommet(self,s)` qui ajoute la clé `s` dans le dictionnaire d'adjacence si elle n'est pas déjà présente.
- Une méthode `sommets(self)` qui renvoie la liste des sommets du graphe.

In [22]:

```
class Graphe_D:
    '''Graphe représenté par un dictionnaire d'adjacence'''
    def __init__(self):
        self.adj={}

    def affiche(self):
        for k in self.adj:
            print (k,self.adj[k])

    def ajouter_sommet(self,s):
        if s not in self.adj:
            self.adj[s]=[]

    #Ex 4
    def ajouter_arc(self, s1, s2):
        pass

    #Ex 5
    def arc(self,s1,s2):
        pass

    def sommets(self):
        s=[]
        for k in self.adj:
            s.append(k)
        return s

    #Ex 6
    def voisins(self, s):
        pass

    #Ex 7
    def nb_sommets(self):
        pass

    #Ex 8
    def degre(self,s):
        pass
```

In [23]:

```
#Création d'un graphe avec un dictionnaire d'adjacence
g2=Graphe_D()
for s in ['A','B','C','D']:
    g2.ajouter_sommet(s)
print(g2.sommets())
g2.affiche()
```

```
['A', 'B', 'C', 'D']
A []
B []
C []
D []
```

Exercice 4 :

- Dans la classe `Graphe_D` , écrire la méthode `ajouter_arc(self, s1, s2)` qui créé l'arc du sommet `s1` vers le sommet `s2` en ajoutant `s2` à la liste des voisins de `s1` dans le dictionnaire d'adjacence.
- Remarque : On créera les sommets s'ils n'existent pas encore.
- Les tests se font sur le graphe donné en exemple précédemment.

In [24]:

```
#tests ajouter_arc sur le graphe donné en exemple
g2.ajouter_arc('A','B')
g2.ajouter_arc('A','D')
g2.ajouter_arc('B','C')
g2.ajouter_arc('B','D')
g2.ajouter_arc('C','D')
g2.ajouter_arc('D','B')
g2.affiche()
```

```
A ['B', 'D']
B ['C', 'D']
C ['D']
D ['B']
```

Exercice 5 :

Ajouter la méthode `arc(self, s1,s2)` qui renvoie `True` si l'arc de `s2` vers `s1` existe , et `False` sinon.

In [25]:

```
#tests arcs
assert(g2.arc('B','A')==False)
assert(g2.arc('A','B')==True)
```

Exercice 6 :

Ajouter la méthode `voisins(self, s)` qui renvoie la liste des successeurs de `s` (ses voisins).

In [26]:

```
#tests voisins
assert(g2.voisins('B')==['C','D'])
assert(g2.voisins('C')==['D'])
```

Efficacité

- Utiliser un dictionnaire plutôt qu'une matrice permet de manipuler tout type de données pour les sommets.
- Il n'y a pas besoin de fixer le nombre de sommets au départ, on peut en ajouter autant que l'on veut.
- Lorsqu'il n'y a pas beaucoup d'arcs, l'espace mémoire occupé est moindre que pour la matrice.
- Le coût en temps des opérations est optimal(ajouter un arc, un sommet...).

Remarques

- Il existe de multiples façons d'implémenter des graphes, en particulier utiliser une liste d'adjacence, plutôt qu'un dictionnaire.
- On choisit l'implémentation en fonction de la situation à modéliser.
- Ce qui est important , c'est de pouvoir accéder d'une façon ou d'une autre à tous les sommets du graphe et leurs voisins.

4. Représenter un graphe non orienté

Une façon simple de représenter un graphe non orienté consiste à le représenter exactement comme un graphe orienté et s'assurant en permanence de la propriété suivante : *Il y a un arc de A vers B si et seulement si il y a un arc de B vers A.*

Autrement dit, on considère que l'on a toujours un double arc, orienté dans les deux sens.

- Cela revient à modifier ne particulier la méthode d'ajout d'arcs dans la matrice d'adjacence:

```
def ajoute_arc(self, s1, s2):
      self.adj[s1][s2]=1
      self.adj[s2][s1]=1
```

- Et dans le dictionnaire d'adjacence:

```
def ajouter_arc(self, s1, s2):
      self.ajouter_sommet(s1)
      self.ajouter_sommet(s2)
      self.adj[s1].append(s2)
      sefl.adj[s2].append(s1)
```

5. Exercices

Exercice 7 :

Dans la classe du graphe représenté avec un dictionnaire d'adjacence, ajouter la méthode `nb_sommets(self)` qui renvoie le nombre de sommets contenus dans le graphe.

In [27]:

```
#test nb_sommets
assert(g2.nb_sommets()==4)
assert(Graphe_D().nb_sommets()==0)
```

Exercice 8 :

1. Dans la classe du graphe représenté avec un dictionnaire d'adjacence, ajouter la méthode `degre(self,s)` qui prend en paramètre un sommet `s` et qui renvoie le nombre d'arcs issus de `s`. On appelle cela le degré d'un sommet.
2. Ajouter la même méthode dans la classe du graphe représenté avec une matrice d'adjacence.

In [36]:

```
#tests degre
#1. Avec un dictionnaire
assert(g2.degre('B')==2)
assert(g2.degre('D')==1)

#2. Avec une matrice
assert(g1.degre(1)==2)
assert(g1.degre(3)==1)
```