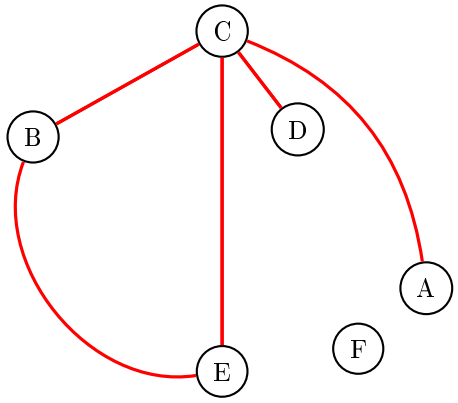
Il existe de nombreuses bibliothèques python permettant de créer et manipuler des graphes. L'une des plus utililisées est la bibliothèque **networkx**.
Cette librairie permet de définir un graphe (orienté ou pas) de différentes manières (listes ou matrices d'ajacences, ensembles de sommets et d'arcs ou d'arêtes. **Networkx** propose bon nombre de méthodes pour manipuler un graphe (ajout et suppression de sommets ou d'arêtes, listes des voisins...
Pour la visualisation du graphe **networkx** contient un module de rendu, **draw**, qui permet en conjonction avec le module d'affichage de **matplotlib** de visualiser le graphe. Il existe d'autres librairies plus complètes qui permettent de visualiser un graphe, on utilisera par exemple **graphviz**.
Pour plus de détails sur l'utilisation de **networkx** et de **graphviz** on pourra, par exemple, consulter les documentations officielles :

networkx : https://networkx.org/documentation/stable/tutorial.html
graphviz : https://graphviz.readthedocs.io/en/stable/manual.html

Partie I : Les graphes non-orientés

L'objectif de ces premières séances est de faire notre propre module, mygraph.py, pour créer et manipuler des graphes.
Comme on le verra tout au long du cours, il existe plusieures manières de représenter un graphe. On commencera par traîter des graphes non-orientés et on utilisera un dictionnaire pour représenter les listes d'adjacences du graphe, le graphe suivant par exemple :



a sera représenté grâce au dictionnaire suivant :

```
graphe = {"A " :{"C"},
          "B" : {"C", "E"},
          "C" : {"A", "B", "D", "E"},
          "D" : {"C"},
          "E" : {"C", "B"},
          "F" : {}
        }
```

On pourra consulter la documentation officielle sur les dictionnaires :

https://docs.python.org/fr/3/tutorial/datastructures.html

Notre module ressemblera donc à cela :

```
"""
Une classe Python pour creer et manipuler des graphes
"""


class Graphe(object):

    def __init__(self, graphe_dict=None):
        """ initialise un objet graphe.
```

```
            Si aucun dictionnaire n'est
            créé ou donné, on en utilisera un
            vide
        """
        if graphe_dict == None:
            graphe_dict = {}
        self._graphe_dict = graphe_dict

    def aretes(self, sommet):
        """ retourne une liste de toutes les aretes d'un sommet"""

    def all_sommets(self):
        """ retourne tous les sommets du graphe """

    def all_aretes(self):
        """ retourne toutes les aretes du graphe
        à partir de la méthode privée,_list_aretes, à définir
        plus bas.
        Ici on fera donc simplement appel à cette méthode.
        """
        return self.__list_aretes()

    def add_sommet(self, sommet):
        """ Si le "sommet" n'set pas déjà présent
        dans le graphe, on rajoute au dictionnaire
        une clé "sommet" avec une liste vide pour valeur.
        Sinon on ne fait rien.
        """

    def add_arete(self, arete):
        """ l'arete est de  type set, tuple ou list;
            Entre deux sommets il peut y avoir plus
            d'une arete (multi-graphe)
        """

    def __list_aretes(self):
        """ Methode privée pour récupérers les aretes.
            Une arete est un ensemble (set)
            avec un (boucle) ou deux sommets.
        """

    def __iter__(self):
            """ on crée un itérable à partir du graphe"""
        self._iter_obj = iter(self._graphe_dict)
        return self._iter_obj

    def __next__(self):
        """ Pour itérer sur les sommets du graphe """
        return next(self._iter_obj)

    def __str__(self):
        res = "sommets: "
        for k in self._graphe_dict:
            res += str(k) + " "
        res += "\naretes: "
        for arete in self.__list_aretes():
            res += str(arete) + " "
        return res
```

La méthode `__init__` (le constructeur) est donnée. Vous pouvez également utiliser l'itérateur fourni `__iter__`

et `__next__` ainsi que la méthode `__str__` permettant la représentation sous forme de chaine d'un objet (mais ce n'est pas obligatoire).
Il vous reste donc à écrire les six autres méthodes.
Quelques exemples d'execution avec les résultats attendus (sur le graphe défini précédemment) :

```
>>> import mygraph as gr

>>> graphe = {"A " :{"C"},
...           "B" : {"C", "E"},
...           "C" : {"A", "B", "D", "E"},
...           "D" : {"C"},
...           "E" : {"C", "B"},
...           "F" : {}
...         }

>>> g=gr.Graphe(graphe)
>>> g.all_aretes()
[set(['A ', 'C']), set(['A', 'C']), set(['C', 'B']), set(['C', 'E']), set(['C', 'D']),
↪  set(['B', 'E'])]


>>> g.all_sommets()
set(['A ', 'C', 'B', 'E', 'D', 'F'])
```