**Université de Rouen**
**Master 2 GIL**
**Matière : Machine Learning en Python**

# TP 1
# Réseau routier

1. Les classes de cette séance doivent être définies dans un module `routes`.
2. Important : pour écrire une méthode pensez à utiliser les méthodes implémentées précédement dans ce même TP. Si vous trouvez que ce TP est trop long c'est que vous n'avez pas appliqué ce principe.

Exercice 1.- **La ville**

Définir la classe `Ville` qui contient :

1. l'initialiseur `__init__(nom, population)`
2. la propriété `nom` : le nom de la ville.
3. la propriété `population` : le nombre d'habitants de la ville.
4. la méthode `__eq__(self, v)` qui retourne vrai si les villes `self` et `v` sont les mêmes (même nom et même nombre d'habitants), faux sinon.
5. la méthode `__str__(self)` qui retourne une chaîne de caractères contenant le nom de la ville et, entre parenthèses, sa population.

Exercice 2.- **La route entre deux villes**

Définir la classe `Route` qui contient :

1. l'initialiseur `__init__(self, v1, v2, distance)` qui initialise une route reliant les deux villes `v1` et `v2`.
2. la propriété `ville1` : la première ville au bout de la route (l'ordre des villes est arbitraire).
3. la propriété `ville2` : la deuxième ville.
4. la propriété `distance` qui retourne la distance entres ces deux villes (longueur de la route).
5. la méthode `circuit(self)` qui retourne vrai si la route `self` relie une ville à la même ville, faux sinon.
6. la méthode `__contains__(self, v)` qui retourne vrai si la ville `v` est une des deux villes de la route, faux sinon.

7. la méthode `memes_villes(self, r)` qui retourne vrai si les routes `self` et `r` relient les mêmes villes, faux sinon. Rappel : l'ordre des villes est arbitraire.
8. la méthode `suit(self, r)` qui retourne vrai si les routes `self` et `r` sont reliées par une même ville, faux sinon.
9. la méthode `__eq__(self, r)` qui retourne vrai si les routes `self` et `r` sont les mêmes (mêmes villes et même distance), faux sinon.
10. la méthode `__lt__(self, r)` qui retourne vrai si la route `self` est (strictement) plus courte que la route `r` et qu'elles ont les mêmes villes, faux sinon.
11. la méthode `__str__(self)` qui retourne les deux villes (nom et population de chaque ville) et la distance.
12. la méthode `__add__(self, r)` qui retourne une route composée des routes `self` et `r` si ses routes se suivent. Dans le cas où une ville est commune à `self` et `r`, la méthode retourne une route qui relie les deux autres villes, une ville de self et une ville de r. Si les deux villes de `self` et `r` sont les mêmes la méthode retourne arbitrairement un circuit plutôt qu'un autre. La méthode déclanche une exception `OperationImpossible` si les routes ne se suivent pas.

    Par exemple ce code :

    ```
    paris = Ville('Paris', 10784830)
    arras = Ville('Arras', 131047)
    lyon = Ville('Lyon', 1659001)
    paris_arras = Route(paris, arras, 185)
    paris_lyon = Route(paris, lyon, 465)
    print(paris_arras + paris_lyon)
    ```

    affichera :

    ```
    Arras (131047) - Lyon (1659001): 650
    ```

Exercice 3.- **Le réseau de routes**

Définir la classe `Reseau` qui permet de représenter un réseau routier avec un nombre illimité de routes.

1. l'initialiseur `__init__(self)` qui initialise un réseau (vide).
2. la propriété `routes` : la liste des routes du réseau.
3. la méthode `__getitem__(self, i)` qui retourne la `i`-ème route du réseau. On suppose que : $0 \leq$ `i` $<$ nombre de routes contenues dans le réseau.
4. la méthode `__len__(self)` qui retourne le nombre de routes du réseau.

5. la méthode `__iter__(self)` qui retourne un itérateur sur les routes du réseau.
6. la méthode `__contains__(self, r)` qui retourne vrai si la route `r` est la même qu'une des routes du réseau, faux sinon.
7. la méthode `ajoute(self, r)` qui ajoute une route `r` au réseau et retourne le réseau lui même (`self`).
8. la méthode `__iadd__(self, res)` qui ajoute toutes les routes d'un reseau `res` au réseau et retourne le réseau lui même (`self`). Avec cette méthode spéciale on pourra écrire `res += res2`.
9. la méthode `__str__(self)` qui retourne une chaîne de caractère énumérant toutes les routes (séparées par un retour à la ligne), chaque route `r` étant représenté par `r.__str__()`.

Exercice 4.- **Les plus courts chemins**

Ajouter à `Reseau` les méthodes :

1. `bonne_route(self, r)` qui retourne vrai si la route `r` relie deux villes différentes qui ne sont pas reliées par une meilleur route dans le réseau, faux sinon. Autrement dit, `r` n'est pas un circuit et il n'y pas de route dans `self` plus courte que `r` entre les villes reliées par `r`.
2. `bonnes_routes(self)` qui retourne un reseau contenant les bonnes routes de `self`, au sens de la méthode `bonne_route(self, r)`. Chaque route ajouté à ce réseau par cette méthode doit être une bonne route dans `self` et doit être **unique** dans le réseau retourné.
3. `__or__(self, res)` qui retourne le réseau contenant les bonnes routes d'un réseau contenant les routes de `self` et de `res`.
4. `__mul__(self, res)` qui retourne les bonnes routes composées d'une route de `self` suivit par une route de `res`. Une route `r` de ce réseau est donc composée d'une route `r1` de `self` suivit par une route `r2` de `res` (`r = r1 + r2`) et c'est une bonne route dans le réseau retourné.
5. `meilleurs_routes3(self)` qui étant donné un réseau de routes directes `self`, retourne le réseau des routes les plus courtes qu'il est possible de prendre à partir de n'importe quelle ville vers n'importe quelle ville en traversant au maximum trois routes de `self`.

   Indication : une route du réseau retournée par la méthode `meilleurs_routes3` est soit une route directe de `self`, soit une route qui traverse deux routes de `self`, soit une route qui traverse trois routes de `self`.
6. `meilleurs_routes(self)` qui étant donné un réseau de routes directes `self`, retourne le réseau des routes les plus courtes qu'il est possible de

prendre à partir de n'importe quelle ville vers n'importe quelle ville en traversant autant de routes qu'on désire.

Indication : une route retournée par la méthode `meilleurs_routes` traverse au maximum les `n` routes de `self`.

7. Tester le tout avec ce code qui crée le réseau de la figure 1 et affiche les 45 routes créées par la méthode `reseau.meilleurs_routes()`.

```
paris = Ville('Paris', 10784830)
arras = Ville('Arras', 131047)
lyon = Ville('Lyon', 1659001)
nantes = Ville('Nantes', 650081)
strasbourg = Ville('Strasbourg', 467438)
marseille = Ville('Marseille', 1590867)
montpellier = Ville('Montpellier', 440896)
poitiers = Ville('Poitiers', 131499)
brest = Ville('Brest', 201741)
bordeaux = Ville('Bordeaux', 927445)

reseau = Reseau()
reseau.ajoute(Route(paris, arras, 185))
reseau.ajoute(Route(paris, lyon, 465))
reseau.ajoute(Route(paris, poitiers, 338))
reseau.ajoute(Route(paris, brest, 593))
reseau.ajoute(Route(paris, nantes, 386))
reseau.ajoute(Route(arras, nantes, 561))
reseau.ajoute(Route(arras, strasbourg, 522))
reseau.ajoute(Route(nantes, brest, 298))
reseau.ajoute(Route(strasbourg, lyon, 494))
reseau.ajoute(Route(strasbourg, marseille, 809))
reseau.ajoute(Route(strasbourg, montpellier, 797))
reseau.ajoute(Route(lyon, marseille, 315))
reseau.ajoute(Route(lyon, montpellier, 303))
reseau.ajoute(Route(marseille, montpellier, 171))
reseau.ajoute(Route(montpellier, poitiers, 557))
reseau.ajoute(Route(poitiers, bordeaux, 237))
reseau.ajoute(Route(bordeaux, nantes, 334))

print(reseau.meilleurs_routes())
```

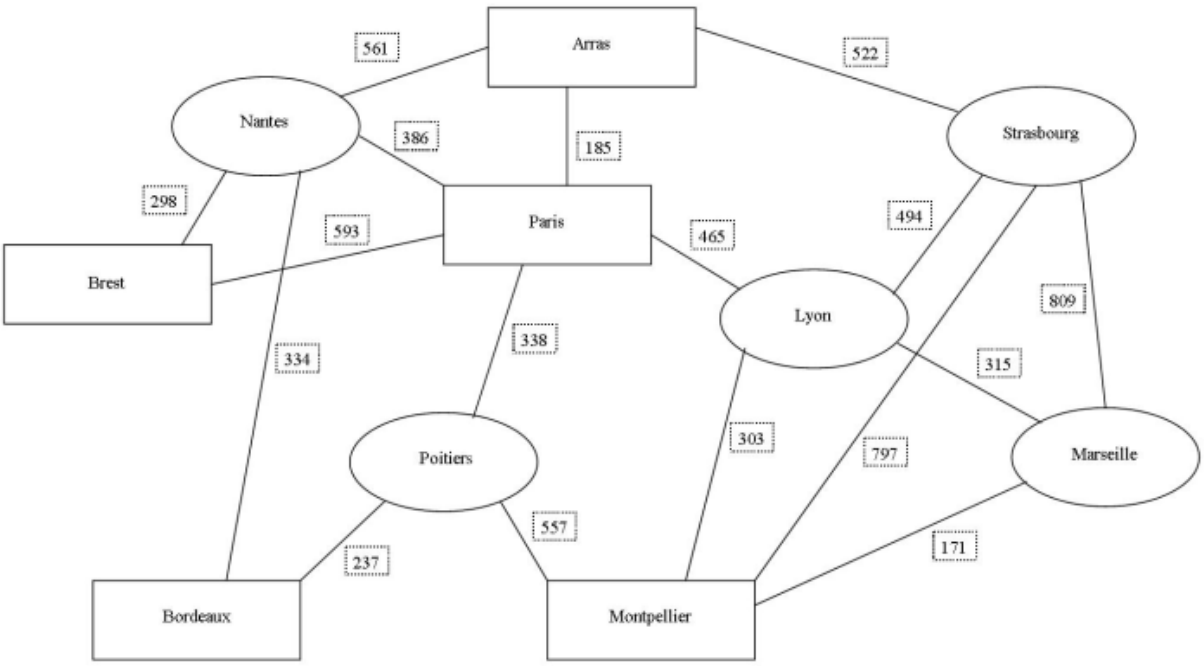


FIGURE 1 – Réseau routier