Titre

Exercice 1

Partie A : Réseau

1. Le terme *architecture* fait référence au matériel. Lors d'une communication, les données sont découpées en *paquets*. Le terme qui désigne l'ensemble des règles de communication utilisées pour réaliser un service particulier sur le réseau est *protocole*.

2. L'élément A joue le rôle de *passerelle* entre un réseau local et le réseau global : c'est un *routeur*. B et C sont quant à eux des *switch*.

3. Le poste 3 est sur le réseau local, 192.168.11.0, donc la ligne peut être :

Matériel	Adresse IP	Masque	Passerelle
Poste 3	192.168.11.xx	255.255.255.0	192.168.11.1

où xx est n'importe quel nombre compris entre 2 et 9 inclus, et 22 et 253 inclus.

22 semble être le choix logique.

Partie B : Routage réseaux

1. Les réseaux *directement connectés* à R1 sont ceux qui sont à distance nulle. Leurs IP sont

   - 10.0.0.0
   - 172.16.0.0
   - 192.168.0.0

2. Il suffit de repérer à quels réseaux les adresses IP appartiennent :

Adresse IP destination	Interface Machine ou Port
192.168.1.55	192.168.0.1
172.18.10.10	172.15.0.1

3. Il s'agit bien du protocole RIP puisqu'on a aucune information sur les débits des connexions.

Routeur	Métrique	Route
R2	0	R1 - R2
R3	0	R1 - R3
R4	1	R1 - R2 - R4
R5	1	R1 - R3 - R4
R6	1	R1 - R3 - R6
R7	2	R1 - R2 - R4 - R7

Pour la ligne de R7 on aurait tout aussi bien pu indiquer R1 - R3 - R6 - R7.

Exercice 2

1. La liaison n'est pas valide car dans cette liste figure `["Luchon", "Muret"]`, qui n'est pas une *liaison directe*.

2.
```
liaisonJoueur1 = [["Tarbes", "St Gaudens"],
                  ["Toulouse", "Castres"],
                  ["Toulouse", "Castelnaudary"],
                  ["Castres", "Mezamet"],
                  ["Castelnaudary", "Carcassonne"]]
```

3. a. `assert listeLiaisons, "la liste est vide"`

   b. `construireDict(listeLiaison)` renvoie

   ```
   {
       'Toulouse': ['Muret', 'Montauban'],
       'Gaillac': ['St Sulpice'],
       'Muret': ['Pamiers']
   }
   ```

   Ce n'est pas la bonne valeur, car il manque des clés.

   En fait, telle que la fonction est programmée, les villes figurant en deuxième position ne sont jamais mises en clé.

   c. On peut procéder de plusieurs manières.

   On peut rendre le code symétrique en rajoutant ce bloc en ligne 16 :

   ```
   if not villeB in Dict.keys():
       Dict[villeB] = [villeA]
   else:
       destinationsB = Dict[villeB]
       if not villeA in destinationsB:
           destinationsB.append(villeA)
   ```

   Toutefois, une solution bien plus courte et élégante consiste à remplacer la ligne 7 par :

   ```
   for liaison in listeLiaisons + [[y, x] for [x, y] in listeLiaisons]:
   ```

   Ce qui « symétrise » la liste sur laquelle on itère.

Exercice 3

1. Le langage utilisé est le SQL (*Simple Query Language*).

2. a. `ATOMES` a pour attributs

- Z : INT
- nom : VARCHAR
- Sym : VARCHAR
- L : INT
- C : INT
- masse_atom : DECIMAL

VALENCE a pour attributs

- Col : INTEGER
- Couche : TEXT

b. Les attributs Z, nom et Sym peuvent jouer le rôle de clé primaire car chacun de ces attributs identifie de manière unique un élément.

C peut être une clé étrangère faisant réference à Col, clé primaire de VALENCE.

c. On a le schéma suivant :

ATOMES(Z INT, nom VARCHAR , Sym VARCHAR , L INT, C INT, masse_atom DEC)

VALENCE(COL INT, Couche TEXT)

3. a. On obtient la table suivante :

nom
aluminium
argon
chlore
magnesium
sodium
phosphore
soufre
silice

b. On obtient une table avec l'attribut C, où tous les entiers de 1 à 18 sont présents.

4. a.
```sql
SELECT  nom,masse_atom FROM ATOMES;
```

b.
```sql
SELECT Sym FROM ATOMES
JOIN VALENCE ON ATOMES.C = VALENCE.col
WHERE VALENCE.couche = "s";
```

5.
```sql
UPDATE ATOMES
SET masse_atom = 39.948
WHERE Sym = "Ar";
```

Exercice 4

1. a. Un fichier CSV (*Comma Separated Values*) est un fichier texte contenant des informations (liste ou liste de listes) séparées par des virgules.

   b. `prenom` et la valeur de retour sont de type `str`.

2. a. `import csv`

   b. `assert type(prenom) is str, "L'argument doit être de type str."`

   c. Voici ce que l'on peut faire :

```
def genre(prenom):

    if not (type(prenom) is str):
        return ""

    liste_M = ['f', 'd', 'c', 'b', 'o', 'n', 'm', 'l', 'k',
               'j', 'é', 'h', 'w', 'v', 'u', 't', 's', 'r',
               'q', 'p', 'i', 'þ', 'z', 'x', 'ç', 'ö', 'ã',
               'â', 'ï', 'g']
    liste_F = ['e', 'a', 'ä', 'ü', 'y', 'ë']

    if prenom[len(prenom)-1].lower() in liste_M :
        return "M"
    elif prenom[len(prenom)-1].lower() in liste_F :
        return "F"
    else :
        return "I"
```

   Ainsi si l'argument n'est pas un str, la fonction renvoie `""` mais ne produit pas d'erreur.

3. Il suffit de remplacer `prenom[len(prenom)-1]` par `prenom[-2:]`.