EXERCICE 3 (6 points)

Cet exercice traite de programmation orientée objet en Python et d'algorithmique.

Un pays est composé de différentes régions. Deux régions sont voisines si elles ont au moins une frontière en commun. L'objectif est d'attribuer une couleur à chaque région sur la carte du pays sans que deux régions voisines aient la même couleur et en utilisant le moins de couleurs possibles.
La figure 1 ci-dessous donne un exemple de résultat de coloration des régions de la France métropolitaine.

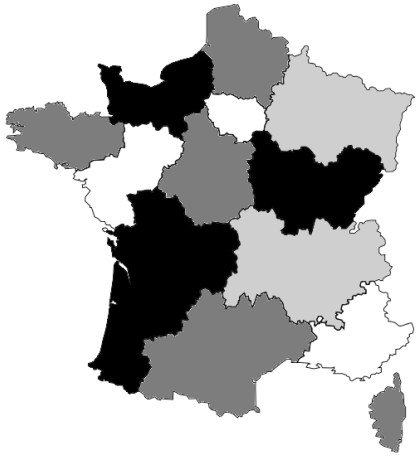

Figure 1 – Carte coloriée des régions de France métropolitaine

On rappelle quelques fonctions et méthodes des tableaux (le type `list` en Python) qui pourront être utilisées dans cet exercice :

- `len(tab)` : renvoie le nombre d'éléments du tableau `tab` ;
- `tab.append(elt)` : ajoute l'élément `elt` en fin de tableau `tab` ;
- `tab.remove(elt)` : enlève la première occurrence de `elt` de `tab` si `elt` est dans `tab`. Provoque une erreur sinon.

Exemple :

- `len([1, 3, 12, 24, 3])` renvoie `5` ;
- avec `tab = [1, 3, 12, 24, 3]`, l'instruction `tab.append(7)` modifie `tab` en `[1, 3, 12, 24, 3, 7]` ;
- avec `tab = [1, 3, 12, 24, 3]`, l'instruction `tab.remove(3)` modifie `tab` en `[1, 12, 24, 3]`.

Les deux parties de cet exercice forment un ensemble. Cependant, il n'est pas nécessaire d'avoir répondu à une question pour aborder la suivante. En particulier, on pourra utiliser les méthodes des questions précédentes même quand elles n'ont pas été codées.

Pour chaque question, toute trace de réflexion sera prise en compte.

Partie 1

On considère la classe `Region` qui modélise une région sur une carte et dont le début de l'implémentation est :

```
1  class Region:
2      '''Modélise une région d'un pays sur une carte.'''
3      def __init__(self, nom_region):
4          '''
5          initialise une région
6          : param nom_region (str) le nom de la région
7          '''
8          self.nom = nom_region
9          # tableau des régions voisines, vide au départ
10         self.tab_voisines = []
11         # tableau des couleurs disponibles pour colorier
   la région
12         self.tab_couleurs_disponibles = ['rouge', 'vert',
   'bleu', 'jaune', 'orange', 'marron']
13         # couleur attribuée à la région et non encore
   choisie au départ
14         self.couleur_attribuee = None
```

1. Associer, en vous appuyant sur l'extrait de code précédent, les noms `nom`, `tab_voisines`, `tab_couleurs_disponibles` et `couleur_attribuee` au terme qui leur correspond parmi : *objet*, *attribut*, *méthode* ou *classe*.

2. Indiquer le type du paramètre `nom_region` de la méthode `__init__` de la classe `Region`.

3. Donner une instruction permettant de créer une instance nommée `ge` de la classe `Region` correspondant à la région dont le nom est « Grand Est ».

4. Recopier et compléter la ligne 6 de la méthode de la classe `Region` ci-dessous :

```
1  def renvoie_premiere_couleur_disponible(self):
2      '''
3      Renvoie la première couleur du tableau des couleurs
   disponibles supposé non vide.
4      : return (str)
5      '''
6      return ...
```

5. Recopier et compléter la ligne 6 de la méthode de la classe `Region` ci-dessous :

```
1  def renvoie_nb_voisines(self) :
2      '''
3      Renvoie le nombre de régions voisines.
4      : return (int)
5      '''
6      return ...
```

6. Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 6 :

```
1  def est_coloriee(self):
2      '''
3      Renvoie True si une couleur a été attribuée à cette
   région et False sinon.
4      : return (bool)
5      '''
6      ...
```

7. Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 8 :

```
1  def retire_couleur(self, couleur):
2      '''
3      Retire couleur du tableau de couleurs disponibles de
   la région si elle est dans ce tableau. Ne fait rien
   sinon.
4      : param couleur (str)
5      : ne renvoie rien
6      : effet de bord sur le tableau des couleurs
   disponibles
7      '''
8      ...
```

8. Compléter la méthode de la classe `Region` ci-dessous, à partir de la ligne 7, en utilisant une boucle :

```
1  def est_voisine(self, region):
2      '''
3      Renvoie True si la region passée en paramètre est une
   voisine et False sinon.
4      : param region (Region)
5      : return (bool)
6      '''
7      ...
```


Partie 2

Dans cette partie :

- on considère qu'on dispose d'un ensemble d'instances de la classe `Region` pour lesquelles l'attribut `tab_voisines` a été renseigné ;
- on pourra utiliser les méthodes de la classe `Region` évoquées dans les questions de la partie 1 :
  - `renvoie_premiere_couleur_disponible`
  - `renvoie_nb_voisines`
  - `est_coloriee`
  - `retire_couleur`
  - `est_voisine`

On a créé une classe `Pays` :

- cette classe modélise la carte d'un pays composé de régions ;
- l'unique attribut `tab_regions` de cette classe est un tableau (type `list` en Python) dont les éléments sont des instances de la classe `Region`.

9. Recopier et compléter la méthode de la classe `Pays` ci-dessous à partir de la ligne 7 :

```
1  def renvoie_tab_regions_non_coloriees(self):
2      '''
3      Renvoie un tableau dont les éléments sont les régions
   du pays sans couleur attribuée.
4      : return (list) tableau d'instances de la classe
5  Region
6      '''
7      ...
```

10. On considère la méthode de la classe `Pays` ci-dessous.

```
1  def renvoie_max(self):
2      nb_voisines_max = -1
3      region_max = None
4      for reg in self.renvoie_tab_regions_non_coloriees():
5          if reg.renvoie_nb_voisines() > nb_voisines_max:
6              nb_voisines_max = reg.renvoie_nb_voisines()
7              region_max = reg
8      return region_max
```

a. Expliquer dans quel cas cette méthode renvoie `None`.

b. Indiquer, dans le cas où cette méthode ne renvoie pas `None`, les deux particularités de la région renvoyée.

Exercice 4

Thème abordé : programmation objet en langage Python

Un fabricant de brioches décide d'informatiser sa gestion des stocks. Il écrit pour cela un programme en langage Python. Une partie de son travail consiste à développer une classe `Stock` *dont la première version est la suivante :*

```
class Stock:
  def __init__(self):
    self.qt_farine = 0  # quantité de farine initialisée à 0 g
    self.nb_oeufs = 0   # nombre d'œufs (0 à l'initialisation)
    self.qt_beurre = 0  # quantité de beurre initialisée à 0 g
```

1. Écrire une méthode `ajouter_beurre(self, qt)` qui ajoute la quantité `qt` de beurre à un objet de la classe `Stock`.

On admet que l'on a écrit deux autres méthodes `ajouter_farine` et `ajouter_oeufs` qui ont des fonctionnements analogues.

2. Écrire une méthode `afficher(self)` qui affiche la quantité de farine, d'œufs et de beurre d'un objet de type `Stock`. L'exemple ci-dessous illustre l'exécution de cette méthode dans la console :

```
>>> mon_stock = Stock()
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 0
>>> mon_stock.ajouter_beurre(560)
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 560
```

3. Pour faire une brioche, il faut 350 g de farine, 175 g de beurre et 4 œufs. Écrire une méthode `stock_suffisant_brioche(self)` qui renvoie un booléen : `VRAI` s'il y a assez d'ingrédients dans le stock pour faire une brioche et `FAUX` sinon.
4. On considère la méthode supplémentaire `produire(self)` de la classe `Stock` donnée par le code suivant :

```
def produire(self):
    res = 0
    while self.stock_suffisant_brioche():
        self.qt_beurre = self.qt_beurre - 175
        self.qt_farine = self.qt_farine - 350
        self.nb_oeufs = self.nb_oeufs - 4
        res = res + 1
    return res
```

On considère un stock défini par les instructions suivantes :

```
>>> mon_stock=Stock()
>>> mon_stock.ajouter_beurre(1000)
>>> mon_stock.ajouter_farine(1000)
>>> mon_stock.ajouter_oeufs(10)
```

a. On exécute ensuite l'instruction

```
>>> mon_stock.produire()
```

Quelle valeur s'affiche dans la console ? Que représente cette valeur ?

b. On exécute ensuite l'instruction

```
>>> mon_stock.afficher()
```

Que s'affiche-t-il dans la console ?

5. L'industriel possède `n` lieux de production distincts et donc `n` stocks distincts. On suppose que ces stocks sont dans une liste dont chaque élément est un objet de type `Stock`. Écrire une fonction Python `nb_brioches(liste_stocks)` possédant pour unique paramètre la liste des stocks et renvoie le nombre total de brioches produites.

Exercice 3 (6 points)

Programmation orientée objet en Python et algorithmique

Partie 1

1. `nom`, `tab_voisines`, `tab_couleurs_disponibles` et `couleur_attribuee` sont des *attributs* (de la classe `Region`)
2. Le paramètre `nom_region` est de type `str` (chaine de caractères)
3. `ge = Region("Grand Est")`
4. `return self.tab_couleurs_disponibles[0]`
5. `return len(self.tab_voisines)`
6. `return self.couleur_attribuee != None`

> **Note**
>
> On peut aussi écrire :
>
> ```
> if self.couleur_attribuee != None:
>     return True
> else:
>     return False
> ```

7.
```
if couleur in self.tab_couleurs_disponibles:
    self.tab_couleurs_disponibles.remove(couleur)
```

> **Note**
>
> Comme rappelé dans l'énoncé `remove` provoque une erreur lorsque l'élément à enlever ne se trouve pas dans la liste. On vérifie donc la présence de la couleur à l'aide de `in` avant de la supprimer.

8.
```
for voisine in self.tab_voisines:
    if voisine == region:
        return True
return False
```

Partie 2

9.
```
non_coloriees = []
for region in self.tab_regions:
    if not region.est_coloriee():
        non_coloriees.append(region)
return non_coloriees
```

> **Note**
>
> On peut proposer une version utilisant les listes définies par compréhension :
>
> ```
> return [region for region in self.tab_regions if not region.est_coloriee()]
> ```

10. a. Cette méthode renvoie `None` lorsqu'il n'y a plus aucune région non coloriée.

    b. La région renvoyée n'est pas encore coloriée et possède le plus de regions voisines.

Exercice 4

programmation objet en langage Python

1.
```
def ajouter_beurre(self,qt):
    self.qt_beurre += qt
```

2.
```
def afficher(self):
    return f"farine : {self.farine} \n oeuf : {self.nb_oeufs} \n beurre : {self.qt_beurre}"
```

3.
```
def stock_suffisant_brioche(self):
    return self.farine >= 350 and self.beurre >= 175 and self.nb_oeufs >= 4
```

4. a. La valeur qui s'affiche dans la console est **2**, cette valeur est le nombre maximal de brioches qu'on peut produire avec 1000 g de beurre, 1000g de farine et 10 oeufs (on est limité par le nombre d'oeufs).

   b. Dans la console, on aura l'affichage suivant :

```
>>> mon_stock.afficher()
farine : 300
oeuf : 2
beurre : 650
```

5.
```
def nb_brioches(liste_stocks):
    total = 0
    for stock in liste_stocks:
        total += stock.produire()
    return total
```