#1

Classe Produit

Écrivez une **classe** nommée `Produit` qui encapsule un produit composé d'un **numéro** d'identification (entier), d'une **description** textuelle et d'un **prix** (virgule flottante).

Votre classe doit supporter les cinq (5) méthodes suivantes:

1. Un **constructeur** qui accepte trois arguments dans cet ordre: le **numéro** d'identification du produit, sa **description** textuelle et son **prix**.
2. Une méthode de conversion en **chaîne de caractères** qui produit le format:

   ```
   #nid (prix$): description
   ```

   où *nid* est le numéro d'identification du produit, *prix* est son prix et *description* sa description.
3. Une méthode nommée `numero` qui retourne le **numéro** d'identification du produit.
4. Une méthode nommée `prix` qui accepte un argument **optionnel** (valeur par défaut `None`) permettant de fixer le prix du produit. Lorsqu'appelé sans argument, cette méthode retourne simplement le **prix** actuel du produit. Lorsqu'appelée avec argument, elle retourne le nouveau prix fixé.
5. Une méthode nommée `description` qui retourne la **description** du produit.

Attention: prenez garde de ne **pas** nommer vos variables d'instances avec le nom de vos méthodes!

#2 Écrivez une classe nommée `Inventaire` qui encapsule un inventaire de produits. Votre classe doit supporter les quatre (4) méthodes suivantes:

1. Un constructeur qui accepte en argument un **itérable** de couples (*produit*, *quantité*). Le produit de ce couple est un objet qui possède les trois (3) méthodes suivantes: `numero`, `prix` et `description`, comme les instances de la classe `Produit` de l'exercice précédent, et où la quantité associée à ce produit est un entier positif ou nul. Notez qu'il n'est pas nécessaire d'avoir une solution fonctionnelle de l'exercice précédent pour compléter celui-ci, car le correcteur vous fournira de tels objets. Faites en sorte que cet itérable soit optionnel en lui donnant la valeur `None` par défaut.
2. Une méthode `ajouter` qui accepte deux arguments: un *produit* et une *quantité* de ce produit à ajouter à l'inventaire. Notez que le produit à ajouter peut être nouveau ou déjà existant. Son numéro d'identification est unique, de sorte que s'il existe déjà dans l'inventaire, il faut ajouter la quantité spécifiée à la quantité de produit existant. Sinon, il faut ajouter un nouveau produit à l'inventaire. Rendez l'argument *quantité* optionnel en lui donnant une valeur par défaut de 1. La méthode doit **retourner** la quantité totale de ce produit actuellement dans l'inventaire.
3. Une méthode `quantite` qui accepte un numéro d'identification en argument et qui **retourne** la quantité actuelle du produit correspondant dans l'inventaire. Si le produit n'existe pas dans l'inventaire, la méthode doit retourner `None`.
4. Une méthode `valeur_totale` qui retourne la valeur total de l'inventaire, en faisant la somme pour chaque produit de son prix par sa quantité (vous pouvez supposer que les produits possèdent tous une méthode `prix`).

Notez que vous devez détecter les erreurs potentielles suivantes en soulevant les exceptions spécifiées:

1. Dans la méthode `ajouter`, vous devez soulever une exception de type `TypeError` si le produit ne possède pas de méthode `numero`. Un moyen de faire cette vérification est de d'abord vérifier avec la fonction hasattr que le produit possède bien un attribut dont l'identifieur est *numero*, puis de vérifier avec la fonction callable que cet attribut est bien un objet que l'on peut appeler comme une fonction. Ne cherchez pas midi à quatorze heures, c'est très simple à faire, il vous suffit de consulter la documentation des deux fonctions standards pour découvrir comment.
2. Également dans `ajouter`, vous devez soulever une exception de type `ValueError` si la quantité à ajouter est négative.

Et puisque votre constructeur a aussi la tâche d'ajouter des produits à l'inventaire initialement vide, celui-ci devrait normalement faire appel à la méthode `ajouter` pour construire l'inventaire initial.

Indice: puisque les produits sont identifiés par leur numéro, et que ces numéros ne sont pas nécessairement consécutifs, vous devriez utiliser un dictionnaire pour conserver votre inventaire. Ce dictionnaire pourrait par exemple associer le numéro d'identification du produit au couple (produit, quantité).

#3 Écrivez le code source d'une **fonction** nommée `compter` qui accepte en argument un objet **itérable** dont les éléments sont potentiellement eux-mêmes itérables, **récursivement**, et qui **retourne** le nombre **total** d'objets. Votre fonction doit reconnaître **quatre** types particuliers d'objets :

1. la **liste** ;
2. le **tuple** ;
3. l'**ensemble** ;
4. et le **dictionnaire**.

Votre fonction doit **initialiser** une variable pour accumuler le compte, puis **itérer** sur les éléments de l'itérable. Lorsqu'un élément correspond à l'un ou l'autre des **types** ci-dessus, votre fonction doit **ajouter** au compte le résultat d'un traitement **particulier** sur cet élément. Autrement, pour **tout** autre élément, elle ajoute simplement la **valeur 1** au compte actuel.

Dans le cas d'une **liste** ou d'un **tuple**, le traitement consiste à faire un appel **récursif** directement sur l'objet. Dans le cas d'un **ensemble**, le traitement consiste à déterminer le nombre d'éléments de l'ensemble. Dans le cas d'un **dictionnaire**, le traitement consiste à faire un appel **récursif** sur les **valeurs** de l'objet.

À la fin de l'itération, la fonction **retourne** simplement la valeur du compte accumulé. Par exemple :

```
compter([1, [2, [3, 4, 5], {'a': 6, 'b': (7, 8, 9), 'c': {'x': 10}}], (11, 12)])
```

doit retourner `12`.

Rappel: la fonction isinstance permet de déterminer si un **objet** est oui on non une **instance** d'une classe.

#4 Tout le monde connaît le **Jeu** du **chat** et de la **souris**. Le chat doit être **assez** près de la souris, soit à une **distance** inférieure ou égale à un certain seuil d, pour pouvoir la capturer. Mais depuis peu, il y a un **chien** dans la maison, ce qui peut poser **problème** pour capturer la souris. En effet, si le chien se trouve **entre** le chat et la souris, alors peu importe la distance qui les sépare, le chat ne peut **pas** capturer la souris.

Vous devez définir une **fonction** nommée `chasser_souris` qui accepte en argument une chaîne de caractères (de longueur arbitraire) ainsi qu'un nombre entier positif ou nul. La chaîne de caractères représente l'**environnement** du jeu alors que le nombre spécifie la **distance** maximale d de capture. Faites en sorte que la distance de capture ait par **défaut** la valeur $d = 5$.

Les différents acteurs du jeu sont **représentés** dans l'environnement par les **caractères** suivants :

- le `C` désigne le **chat** ;
- le `D` désigne le **chien** ;
- le `S` désigne la **souris** ;
- et le `.` désigne un emplacement **libre**.

Selon la situation, votre fonction doit **retourner** l'une des trois chaînes de caractères suivantes :

- `Souris en fuite!` si la souris est en **dehors** de la zone de capture, peu importe qu'elle soit protégée ou non par le chien ;
- `Souris protégée!` si le chien se situe **entre** la souris et le chat et que la souris est à distance de capture ;
- `Souris capturée!` **autrement**.

La distance de **capture** est définie par le nombre **maximal** de caractères pouvant séparer la souris et le chat pour que ce dernier puisse la capturer. Par exemple, une distance de $d = 0$ signifie que le chat et la souris doivent être **adjacents** dans la chaîne de caractères pour que le chat puisse capturer la souris.

Dans le cas d'une distance **négative**, faites en sorte que votre fonction soulève une exception de type `ValueError`.

Notez finalement que le chien n'est **pas** toujours présent dans l'environnement. Vous trouverez par ailleurs quelques environnements de test dans la cellule de contexte.

Corrigé:
#1

Notez que cette solution n'est généralement **pas** unique.

```python
class Produit:
    def __init__(self, nid, desc, prix):
        """
        :param nid: le numéro d'identification du produit.
        :param desc: la description du produit.
        :param prix: le prix du produit.
        """
        self._nid = nid
        self._desc = desc
        self._prix = prix

    def __str__(self):
        return f'#{self._nid} ({self._prix}$): {self._desc}'

    def numero(self):
        return self._nid

    def prix(self, prix=None):
        """
        :param prix: le nouveau prix du produit; None signifie que le prix reste inchangé.
        :returns: le prix de ce produit.
        """
        if prix is not None:
            self._prix = prix
        return self._prix

    def description(self):
        return self._desc
```

#2

```python
class Inventaire:
    def __init__(self, items=None):
        """Construire un inventaire.

        :param items: un itérable de couples (produit, quantité)."""
        self.items = {}
        if items is not None:
            for produit, quantité in items:
                self.ajouter(produit, quantité)

    def ajouter(self, produit, quantité=1):
        """Ajouter  à l'inventaire.

        :param produit: l'instance de produit que l'on veut ajouter à l'inventaire.
        :param quantité: la quantité de produit à ajouter (par défaut 1).
        :returns: la quantité totale de ce produit dans l'inventaire.

        :raises TypeError: si produit n'a pas de méthode numero.
        :raises ValueError: si la quantité est négative."""
        if not hasattr(produit, 'numero') or not callable(produit.numero):
            raise TypeError(f"Le produit {produit} n'a pas de méthode 'numero'.")

        if quantité < 0:
            raise ValueError(f"La quantité {quantité} est invalide.")

        nid = produit.numero()
        item = self.items.get(nid)
        if item is not None:
            self.items[nid] = (produit, item[1]+quantité)

        else:
            self.items[nid] = (produit, quantité)

        return self.quantite(nid)

    def quantite(self, nid):
        """Lire la quantité en inventaire.

        :param nid: le numéro d'identification du produit.
        :returns: la quantité de ce produit dans l'inventaire."""
        _, n = self.items.get(nid, (None, None))
        return n

    def valeur_totale(self):
        """Calculer la valeur totale de l'inventaire.

        :returns: la valeur calculée."""
        total = 0
        for produit, quantité in self.items.values():
            total += produit.prix()*quantité
        return float(total)
```

#3

Notez que cette solution n'est généralement **pas** unique.

```python
def compter(itérable):
    """Fonction qui compte récursivement les éléments d'un itérable"""
    res = 0
    for item in itérable:
        # traiter chaque élément de l'itérable
        if isinstance(item, (list, tuple)):
            # l'item est une list ou un tuple; on fait un appel récursif
            res += compter(item)
        elif isinstance(item, set):
            # l'item est un ensemble; on additionne son nombre d'éléments
            res += len(item)
        elif isinstance(item, dict):
            # l'item est un dictionnaire; on fait un appel récursif sur ses valeurs
            res += compter(item.values())
        else:
            # autrement, on additionne 1 au compte
            res += 1
    return res
```

#4

```python
def chasser_souris(env, d=5):
    if d < 0:
        raise ValueError("La distance de capture doit être positive ou nulle.")

    # trouver le chat
    index_chat = env.index('C')

    # trouver la souris
    index_souris = env.index('S')

    # vérifier la présence du chien
    if env.count('D'):
        index_chien = env.index('D')

    else:
        index_chien = None

    if abs(index_chat - index_souris) > d+1:
        return 'Souris en fuite!'

    elif index_chien is not None:
        if index_chat < index_chien < index_souris or index_souris < index_chien < index_chat:
            return 'Souris protégée!'

    return 'Souris capturée!'
```