Révision d'un texte

Écrivez une **classe** nommée `Revision` qui encapsule les différentes révisions d'un texte et implante les méthodes suivantes:

1. Un **constructeur** qui accepte en argument la version initiale du texte.
2. Une méthode `revise(texte)` qui crée une nouvelle révision avec le texte `texte` et **retourne** le numéro de cette révision.
3. Une méthode `revisions()` sans argument qui **retourne** un fonction génératrice permettant d'itérer sur les différentes révisions du texte, de la plus **récente** à la plus ancienne.
4. Une méthode de conversion en **chaîne de caractères** permettant de produire la révision la plus récente du texte.
5. Une méthode permettant à la fonction standard `len` de déterminer le nombre **total** de révisions, y compris la version initiale.
6. Une méthode permettant d'utiliser l'opérateur `[]` afin d'accéder **directement** à une révision par son numéro (en lecture seulement).

Les détails internes de votre classe sont à votre discrétion, dans la mesure où vous implantez les méthodes spécifiées ci-dessus. Votre **constructeur** doit accepter en argument une chaîne de caractères représentant la version initiale du texte. Cette dernière portera le numéro **zéro** et les révisions subséquentes seront numérotées de 1 à . Par exemple, on doit pouvoir construire une instance associée à un texte initial de cette façon:

```
texte = Revision("Ceci est la version initiale")
```

et utiliser la méthode `revise` pour ajouter une **nouvelle** révision du texte encapsulé. Par exemple:

```
print('révision', texte.revise("Ceci est la 1re révision"), '=',
texte)
```

doit produire la sortie:

```
révision 1 = Ceci est la 1re révision
```

L'appel `texte.revise("Ceci est la 1re révision")` crée une 1re révision du texte et retourne la valeur entière 1. Lorsqu'on passe `texte` en argument à la fonction `print` , cette dernière faire appel à la méthode `__str__` de notre instance afin de produire la révision la plus **récente** du texte.

La méthode `revisions` doit permettre d'**itérer** sur toutes les révisions du texte, dans l'ordre de la plus **récente** à la plus **ancienne**, par exemple dans une boucle `for` :

```
for revision in texte.revisions():
      print(revision)
```

qui doit produire:

```
Ceci est la 1re révision
Ceci est la version initiale
```

Finalement, on doit aussi pouvoir **itérer** sur les différentes révisions de la façon suivante:

```
for i in range(len(texte)):
      print(texte[i])
```

ce qui doit produire comme précédemment:

```
Ceci est la 1re révision
Ceci est la version initiale
```

Dans le cas où l'utilisateur vous transmettrait du texte sous une forme autre qu'une chaîne de caractères, votre classe doit **soulever** une exception de type `TypeError` .

```python
#Révision d'un texte
class Revision:

    def __init__(self, texte):
        if not isinstance(texte, str):
            raise TypeError
        self._revisions = [texte]

    def __str__(self):
        return self._revisions[-1]

    def __len__(self):
        return len(self._revisions)

    def __getitem__(self, position):
        liste = list(reversed(self._revisions))
        return liste[position]

    def revise(self, texte2):
        self._revisions.append(texte2)
        return len(self._revisions) - 1

    def revisions(self):
        return reversed(self._revisions)

```

Horodateur

Définissez un décorateur nommé `horodateur` qui, à **chaque** appel de la fonction décorée, affiche la **date** de l'appel ainsi que la valeur de ses **arguments**. Implantez votre décorateur sous la forme d'une **fonction** qui accepte en argument une autre fonction.
Sur une même ligne, le format d'affichage doit comprendre les éléments suivants:

1. la **date** du jour sous le format `jj/mm/aaaa` ;
2. un espace;
3. le **nom** de la fonction décorée;
4. une parenthèse ouvrante;
5. les arguments **positionnels** utilisés lors de l'appel, séparés par des virgules
6. les arguments **nommés** utilisés lors de l'appel, séparés par des virgules
7. une parenthèse fermante.

Par exemple:

```
@horodateur
def ma_fonction(*args, **kargs):
      pass
```

appelée avec les arguments:

```
ma_fonction(1, 5, 3, a=9, z=13)
```

doit produire l'affichage:

```
15/12/2018 ma_fonction(1, 5, 3, a=9, z=13)
```

Notez bien que ceci n'est qu'un exemple pour un appel effectué le 15 décembre 2018, pour une fonction spécifique et des arguments également spécifiques. Votre décorateur doit être **général**, c'est-à-dire produire l'affichage qui correspond à la **date du jour**, au nom de la **fonction décorée** ainsi qu'aux valeurs **effectives** des arguments utilisés lors de l'appel. Nous testerons avec d'autres noms de fonction et d'autres combinaisons d'arguments. N'oubliez pas non plus que votre décorateur doit **appeler** la fonction décorée avec l'ensemble de ses arguments (positionnels et nommés) et **retourner** son résultat.

Dans le cas des **arguments nommés**, assurez-vous de les énumérer dans l'ordre **lexicographique** (l'ordre des mots d'un dictionnaire français).
Indices:

1. pour obtenir la date du jour, faites appel à la fonction `today` de la classe `datetime.date` (cette classe est d'ailleurs déjà importée dans le contexte de cet exercice);
2. pour connaître le nom d'un objet de la classe `function` , il suffit d'accéder à son attribut `__name__` ;
3. affichez les valeurs des arguments reçus en utilisant `str` pour les convertir en chaînes de caractères, et utiliser la méthode `join` pour produire des séquences de chaînes séparées par des virgules;
4. pour le tri lexicographique, utilisez simplement sorted (https://docs.python.org/fr/3/library/functions.html#sorted).

```python
#horodateur
from datetime import date


def horodateur(func):
    def fct(*args, **kargs):
        today = date.today()
        liste = []
        for i in args:
            liste.append(f"{i}")
        for i in sorted(kargs):
            liste.append(f"{i}={kargs[i]}")
        chn = ", ".join(liste)
        return f"""{today.day}/{today.month}/{today.year} {func.__name__}({chn})"""
    return fct
```

Somme de dictionnaires

Définissez une **fonction** nommée `sommation` qui accepte en argument une **liste** de **dictionnaires**, et qui retourne en sortie un dictionnaire **unique** dont les clés correspondent à l'intégrale des clés des dictionnaires reçus en entrée, et dont les valeurs correspondent aux sommes des valeurs associées à ces clés. Par exemple, pour la liste suivante:

```
[
{'a': 2, 'c': 5, 'z': -3},
{'z': 7, 'b': 1},
{'c': 23},
{'f': 0, 'z': 2},
]
```

la sortie de votre fonction doit être:

```
{'a': 2, 'b': 1, 'c': 28, 'f': 0, 'z': 6}
```

Les dictionnaires d'entrée peuvent contenir des clés **arbitraires**, mais toutes les valeurs associées à ces clés sont des **entiers**. Si certaines valeur ne sont **pas** des entiers, votre fonction doit poursuivre son travail en ignorant ces valeurs et, à la fin seulement, soulever une exception de type `TypeError` . Cette exception doit alors contenir un attribut nommé `result` associé au dictionnaire du résultat :

Notez que pour affecter un résultat `r` à un attribut `y` d'un objet `x` (y compris une exception), il vous suffit d'écrire `x.y = r` . Notez aussi que votre fonction doit **retourner** le dictionnaire résultat (ou l'exception), et **non** l'afficher.

```python
#Sommes de dictionnaires
def sommation(liste):
    result = {}
    erreur = False
    for dico in liste:
        for k, v in dico.items():
            if not isinstance(v, int):
                erreur = True
            else:
                result[k] = result.get(k, 0) + v
    if erreur:
        erreur = TypeError("Au moins une valeur est invalide")
        return erreur
    else:
        return result
```

Liste triée

Définissez une **classe** nommée `ListeTrie` qui **hérite** la majorité de son comportement de la classe standard `list` , mais qui possède la caractéristique supplémentaire de toujours conserver ses éléments **triés**. Tout comme la liste standard, le constructeur de votre classe doit accepter un **itérable** et s'en servir pour initialiser la liste. Donnez à cet itérable la valeur par défaut `[]` , afin que l'on puisse construire une liste vide **sans** argument. Votre constructeur doit aussi accepter un argument supplémentaire optionnel, mais **obligatoirement** nommé `croissant` , avec une valeur par défaut `True` . Lorsque cet argument est `True` , les éléments de votre liste doivent être triés dans l'ordre **croissant**, et lorsqu'il est `False` , il doivent être triés dans l'ordre **décroissant**.

Pour trier les éléments de votre liste, vous n'avez qu'à faire appel à la méthode sort (https://docs.python.org/fr/3/library/stdtypes.html#list.sort) de `list` qui possède un argument optionel permettant de choisir l'ordre du trie. Notez bien que cette méthode trie les éléments **en place** dans la liste et qu'elle ne retourne jamais **rien**. Outre le constructeur de la classe, vous devez surdéfinir la méthode `append` afin de toujours insérer l'élément à la bonne position dans la liste, de même que l'opérateur `+` (méthode `__add__`), afin que le résultat de la concaténation de deux listes triés produise toujours une liste triée. Dans le cas de deux listes qui seraient triés dans des ordres différents, utilisez l'ordre de la liste de **gauche** pour déterminer l'ordre de la liste résultante.

Voici quelques exemples d'utilisation:

```
print(ListeTrie([1, 7, 5, 3]))
```

produira:

```
[1, 3, 5, 7]
```

et:

```
print(ListeTrie([1, 7, 5, 3], croissant=False))
```

produira:

```
[7, 5, 3, 1]
```

De même:

```
print(ListeTrie([1, 7, 5, 3]) + ListeTrie([6, 2, 4]))
```

produira:

```
[1, 2, 3, 4, 5, 6, 7]
```

et:

```
x = ListeTrie()
x.append(3)
x.append(-1)
x.append(2)
print(x)
```

produira:

```
[-1, 2, 3]
```

Notez bien qu'en **héritant** de la classe `list` , vous vous trouvez à hériter d'une liste. Il importe de vous en servir et **non** d'en créer une nouvelle!

```python
#Liste triée
class ListeTrie(list):
    def __init__(self, iterable=[], *, croissant=True):
        self.croissant = croissant
        if  not self.croissant:
            self.iterable = list(reversed(sorted(iterable)))
        else:
            self.iterable = sorted(iterable)

    def __str__(self):
        return str(self.iterable)

    def __add__(self, other):
        for i in other.iterable:
            self.iterable.append(i)
        if  not self.croissant:
            return list(reversed(sorted(self.iterable)))
        else:
            return list(sorted(self.iterable))

    def append(self, valeur):
        self.iterable.append(valeur)
        if  not self.croissant:
            return list(reversed(sorted(self.iterable)))
        else:
            return list(sorted(self.iterable))
```