[Q1] Approximer la valeur de π (2 points)

Masquer l'énoncé

Plusieur formules existent pour approximer la valeur de π. Une de ces formules est la série de Gregory-Leibniz que voici :

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

Ecrivez une fonction Python qui prend comme argument un entier i ≥ 0 et qui calcule une approximation de π sur base des i + 1 premiers termes de cette série.

La fonction demandée est la suivante :

```
def approx_pi(i):
    """
    @pre:   i est un entier tel que i >= 0
    @post:  retourne une estimation de pi en sommant
            les i + 1 premiers termes de la série de Gregory-Leibniz
    """
```

A titre d'exemples, voici quelques résultats de l'exécution de cette fonction sur différentes valeurs de `i` :

```
>>> approx_pi(0)
4.0
>>> approx_pi(1)
2.666666666666667
>>> approx_pi(2)
3.466666666666667
>>> approx_pi(3)
2.8952380952380956
```

[Q2] RépétitionRépétition (3 points)

Masquer l'énoncé

Écrivez la fonction `repetition(s)` selon la spécification ci-dessous.

```
def repetition(s):
    """
    @pre:  s est une chaîne de caractères non-vide composée seulement de lettres.
    @post: Retourne la longueur de la plus longue séquence consécutive de lettres répétées
           dans la chaîne de caractères s, sans tenir compte de la casse des lettres.
           (C'est-à-dire qu'une même lettre en majuscule ou en minuscule doivent être considérées
           comme équivalentes.)
    """
```

A titre d'exemples, voici quelques résultats de l'exécution de cette fonction sur différentes chaînes :

```
>>> repetition("Blablabla")
1       # (B, L ou A)
>>> repetition("AABBbbccc")
4       # (B)
>>> repetition("AABBCCccc")
5       # (C)
```

Pour rappel: la méthode `str.upper()` renvoie une chaîne de caractères correspondant à `str` dont tous les caractères ont été mis en majuscule.

[Q3] Tchou Tchouuuuu (4 points)

Masquer l'énoncé

Les chemins de fer belges souhaitent optimiser leurs itinéraires ferroviaires. Chaque itinéraire est représenté comme une liste des villes connectées par cet itinéraire, par exemple :

```
P8014  = [ "Bruxelles-Nord", "Bruxelles-Central", "Bruxelles-Midi", "Liedekerke", "Denderleeuw", "Erembodegem", "Alost" ]
IC2238 = [ "Bruxelles-Nord", "Bruxelles-Central", "Bruxelles-Midi", "Liedekerke", "Denderleeuw", "Alost" ]
S10    = [ "Bruxelles-Nord", "Bockstael", "Jette", "Berchem-Sainte-Agathe", "Grand-Bigard", "Dilbeek", "Sint-Martens-Bodeg
```

Pour certains itinéraires, plusieurs gares à la fin ou au début de l'itinéraire sont identiques. C'est par exemple le cas pour les itinéraires `P8014` et `S10` qui se terminent toutes les deux par un trajet commun : `[ "Liedekerke", "Denderleeuw", "Erembodegem", "Alost" ]`.

Pour aider les chemins de fer SNCB à optimiser leurs itinéraires ferroviaires en analysant ces trajets communs entre les itinéraires, écrivez la fonction suivante :

```
def common_path_at_end(route1,route2) :
    """
    @pre:   route1 et route2 sont deux listes de strings
    @post:  retourne une liste reprenant le plus grand suffixe
            commun entre route1 et route2. Autrement dit la plus longue sous-séquence
            en partant de la fin qui contient les mêmes éléments.
            route1 et route2 n'ont pas été modifiées.
    """
```

A titre d'exemples, voici quelques résultats de l'exécution de cette fonction sur différents itinéraires :

```
>>> common_path_at_end(P8014,S10)
['Liedekerke', 'Denderleeuw', 'Erembodegem', 'Alost']
>>> common_path_at_end(P8014,IC2238)
['Alost']
```

[Q4] Gestion de code pin (4 points)

Masquer l'énoncé

Écrivez la fonction `load` selon la spécification ci-dessous.

```
def load(filename, dictionnary):
    """
    @pre    filename est le nom d'un fichier de texte,
            dictionnary est un dictionnaire avec :
                comme clés : des chaînes de caractères (les noms d'utilisateurs)
                comme valeurs : des entiers (un code pin associé à l'utilisateur)
    @post   Met à jour le dictionnaire à partir des clients repris dans le fichier de nom filename.
            Si une erreur se produit pendant la lecture du fichier, le dictionnaire n'est pas
            modifié.
            Retourne :
                True     si le fichier a été lu sans erreurs
                False    sinon
    """
```

Chaque ligne du fichier contient un nom d'utilisateur suivi du caractère `!` et du code pin de cet utilisateur. Un nom d'utilisateur est une chaîne de caractères sans espace, ni `!`. Un code pin est un nombre entier qui se compose de 4 chiffres dont le premier ne peut pas être 0.

Exemples de lignes correctes :

```
alice!1234
siegfried.nijssen@uclouvain.be!7654
```

Le contenu du dictionnaire initial n'est pas supprimé. Les informations reprises dans le fichier sont utilisées pour mettre à jour le dictionnaire. En particulier, si un utilisateur était déjà présent dans le dictionnaire, son code pin sera mis à jour. Si un utilisateur n'était pas encore dans le dictionnaire, il y sera rajouté avec son code pin.

Attention à traiter correctement les différentes situations d'erreur. Le dictionnaire **n'est pas** mis à jour dans ce cas. En particulier, si les premières lignes du fichier sont correctes, mais la dernière ligne ne l'est pas, le dictionnaire ne doit pas être modifié.

La fonction retourne la valeur `True` si tout s'est bien déroulé et `False` sinon. Par exemple si :

- le fichier ne peut pas être ouvert, ou qu'une erreur d'I/O se produit,
- une ligne du fichier ne contient pas de code pin ou contient un code pin qui n'est pas un nombre entier composé de quatre chiffres,
- toute autre erreur de format de ligne survient.

Il n'est pas demandé de traiter les erreurs liées au nom d'utilisateur. En particulier, on suppose que les noms d'utilisateur ne contiennent pas d'espace, ni le caractère `!`.

Exemples de lignes incorrectes :

```
alice!123
bob!erreur
charlie!5678!eve
```

Remarques :

- La méthode `str.strip()` retourne la chaîne `str` sans les charactères d'espacement (`' '`, `'\n'`) en début et fin de chaîne.
- La méthode `str.split(sep)` retourne une liste des sous-chaînes de `str` séparées par la chaîne `sep`.
- La fonction `create_file` est mise à votre disposition pour les tests, voir plus bas.
- Si nécessaire vous pouvez aussi définir des fonctions auxiliaires.

[Q5intro] Gestion d'énergie

Ceci est le contexte et le code utilisés dans la question 5a et la question 5b. Il est conseillé de commencer par lire ce contexte.
Pour rappel: il est possible d'ouvrir une fenêtre supplémentaire d'INGInious.

Dans ces questions on veut calculer la consommation d'électricité d'un bâtiment. Un bâtiment consiste de plusieurs appartements, chaque appartement a plusieurs pièces, dans chaque pièce on a plusieurs lampes, et chaque lampe consomme de l'électricité. Plus une lampe est puissante, plus elle consomme d'électricité.

Une implémentation partielle est fournie par les classes `Appartement`, `Piece`, `Lampe` et les sous-classes de `Lampe` ci-dessous. En plus des pièces, le code fait aussi référence à la notion de couloirs. Cette notion deviendra plus claire dans la question Q5a :

```
class Appartement :

    def __init__(self, pieces, couloirs) :
        """
        Crée un objet représentant un appartement,
        avec comme attributs:
         une liste des pièces de cet appartement
         une liste des couloirs de cet appartement
        """
        self.pieces = pieces
        self.couloirs = couloirs

    def consommation(self) :
        """
        retourne la consommation moyenne journalière de l'appartement
        """
        consom = 0
        for a in self.pieces :
            consom += a.consommation()
        for c in self.couloirs :
            consom += c.consommation()
        return consom

    def __str__(self) :
        """
        Retourne un texte permettant de représenter cet appartement
        """
        # CODE NON FOURNI

class Piece :

    def __init__(self, nom, lampes, heures=0) :
        """
        Crée un objet représentant une pièce d'une maison,
        avec comme attributs:
         le nom de la pièce,
         une liste des lampes de la pièce,
         le temps d'utilisation moyen de la pièce (nombre d'heures par jour en moyenne que
         la pièce est utilisée), 0 par défaut
        """
        # CODE NON FOURNI

    def consommation(self) :
        """
        retourne la consommation moyenne journalière de la pièce en kWh,
        soit la somme des puissances de chaque lampe de la pièce * le nombre d'heures
        d'utilisation moyenne de la pièce / 24
        """
        # CODE NON FOURNI

    def ajouter_lampe(self, lampe) :
        """
        ajoute une nouvelle lampe à la pièce
        """
        # CODE NON FOURNI
        # VOIR QUESTION 5B
```

```
    def __str__(self) :
        """
        Retourne un texte permettant de représenter cette pièce
        """
        # CODE NON FOURNI

class Lampe :

    def __init__(self, type_lampe, puissance) :
        """
        Crée un objet représentant une lampe,
        avec comme attributs:
         le type de la lampe
         sa puissance énergétique en Watt
        """
        self.__type_lampe = type_lampe
        self.__puissance = puissance

    def puissance_kWh(self) :
        """
        retourne la puissance de la lampe par heure en kWh
        """
        return self.__puissance / 1000

    def __str__(self) :
        """
        Retourne un texte permettant de représenter cette lampe
        """
        return 'Lampe (' + self.__type_lampe + ', ' + self.puissance_kWh() 'kWh)'
```

```
class AmpouleClassique (Lampe) :
    def __init__(self) :
        super().__init__("lampe à incandescence",60)

class AmpouleBasse (Lampe) :
    def __init__(self) :
        super().__init__("ampoule basse consommation",12)

class TubeFluorescent (Lampe) :
    def __init__(self) :
        super().__init__("lampe fluorescente",45)

class LED (Lampe) :
    def __init__(self) :
        super().__init__("ampoule LED",9)
```

Voici un exemple d'utilisation de ces classes :

```
>>> living = Piece("Living", [ AmpouleClassique(), AmpouleClassique(), LED() ], 8 )
>>> garage = Piece("Garage", [ TubeFluorescent() ], 1 )
>>> cuisine = Piece("Cuisine", [ AmpouleBasse(), AmpouleBasse() ], 4 )
>>> appartement = Appartement( [living, garage, cuisine], [] )
>>> print(appartement.consommation())
0.048875
```

[Q5a] La super gestion d'énergie (2 points)

Masquer l'énoncé

Écrivez la classe `Couloir`, qui hérite de `Piece`, représentant un couloir. Un couloir a une certaine longeur, et on y met 1 tube fluorescent par chaque 5m de longeur commencé.

Donc pour un couloir de 9m ou de 10m on a besoin de 2 tubes.
Mais pour un couloir de 10.1m ou de 11m on a besoin de 3 tubes car on a entamé les 5m suivants.

On suppose aussi qu'un couloir d'un appartement a une utilisation moyenne de 2h par jour.

Pour cet exercice, vous pouvez bien sûr supposer une implémentation correcte et complète des classes présentées à la question 5intro, même les fonctions pour lesquelles le code n'est pas fourni. Vous devez par conséquent utiliser leurs définitions et méthodes correctement et à bon escient.

Voici un exemple d'utilisation de `Couloir`, qui créée un objet qui représente un `Couloir` de 9 mètres. Un couloir est une pièce qui a un `TubeFluorescent` tous les 5m entamés. Ce couloir de 9m aura donc 2 `TubeFluorescent` (de 45 Watt soit 45/1000kWh) et une utilisation moyenne de 2h par jour et donc une consommation moyenne *journalière* de 2*2h/24h*45/1000 kWh).

```
>>> c = Couloir(9)
>>> c.consommation()
0.0075
```

[Q5b] Gestion d'énergie, une question liée (5 points)

Masquer l'énoncé

La classe `Piece` maintient l'ensemble de ses lampes dans une liste chaînée. Dans cette liste chaînée, les lampes sont ordonnées par ordre décroissant de puissance.

On voudrait à terme pouvoir implémenter une méthode `supprimer_lampe(self)` dans `Piece` qui nous permettrait de façon écologique de retirer la lampe qui consomme le plus dans la pièce.

Voici plus de détails sur l'implémentation de la classe `Piece` avec cette liste chaînée. La classe interne `Node` utilisée est la même que celle présentée dans le syllabus.

On vous demande d'implémenter la méthode `ajouter_lampe(self, lampe)` dans la classe `Piece` qui ajoute une lampe à la liste chainée de la pièce en respectant l'ordre décroissant en puissance des lampes dans la liste.

```
class Piece :

    class Node:
        def __init__(self, cargo=None, next=None):
            """
            Initialises a new Node object.
            @pre:  -
            @post: A new Node object has been initialised.
                   A node can contain a cargo and a reference to another node.
                   If none of these are given, a node with empty cargo (None) and no reference (None) is created.
            """
            self.__cargo = cargo
            self.__next  = next

        def value(self):
            """
            Returns the value of the cargo contained in this node.
            @pre:  -
            @post: Returns the value of the cargo contained in this node, or None if no cargo  was put there.

            """
            return self.__cargo

        def __str__(self):
            """
            Returns a string representation of the cargo of this node.
            @pre:  self is possibly empty Node object
            @post: returns a print representation of the cargo contained in this Node
            """
            return str(self.value())

        def next(self):
            return self.__next

        def set_next(self, node):
            self.__next = node
def __init__(self, nom, lampes, heures=0) :
    """
    Crée un objet représentant une pièce d'une maison,
    avec comme attributs:
     le nom de la pièce
     une liste des lampes de la pièce
     le temps d'utilisation moyen de la pièce (nombre d'heures par jour en moyenne que la pièce est utilisé), 0 par défaut
    """
    self.nom = nom
    self.heures = heures
    self.__pire_lampe = None # la tête de liste et donc la lampe qui consomme le plus
    self.__n_lampes = 0 # le nombre de lampes dans la pièce
    for l in lampes :
        self.ajouter_lampe(l)
```

```
    def consommation(self) :
        """
        retourne la consommation moyenne journalière de la pièce en kWh,
        soit la somme des puissances de chaque lampe de la pièce * le nombre d'heures
        d'utilisation moyenne de la pièce / 24
        """
        # CODE NON FOURNI

    def ajouter_lampe(self, lampe) :
        """
        ajoute un Node contenant la nouvelle lampe à la pièce dans la liste chaînée
        en cas d'ajout d'une lampe avec la même puissance d'une lampe déjà existante
        dans la liste, cette nouvelle lampe sera ajoutée avant la lampe existante
        la liste chaînée doit rester triée par ordre décroissant de puissance
        le nombre de lampes dans la pièce a été incrémenté de 1
        """
        # VOTRE REPONSE

    def supprimer_lampe(self) :
        """
        enlève la lampe en tête de liste, donc celle qui consomme le plus.
        le nombre de lampes dans la pièce a été aujusté
        ne change rien à la liste si elle est déjà vide.
        @post : retourne la lampe supprimée,
                None si la lise de lampes est vide
        """
        if self.__pire_lampe is not None:
            removed_lamp = self.__pire_lampe.value()
            self.__n_lampes -= 1
            self.__pire_lampe = self.__pire_lampe.next()
            return removed_lamp
        return None
def __str__(self) :
    """
    Retourne un texte permettant de représenter cette pièce
    """
    return 'Une Pièce {}, utilisée en moyenne {} heures contenant {} lampes'.format(self.nom, self.heures, self.__n_lampes)
```