Evaluations 2023-2024 semestre 6 session 1

Retour sur les évaluations

Examen 2023-2024 semestre 6 session 1

Retour sur les examens

Évaluations

- Examen
  - AF1
    - évalué dans les exercices I (coef 2), II (coef 1) et IV (coef 1)
    - Un Fx dans un des exercices pouvant être compensé
  - AF2
    - Évalué dans tous les exercices
    - Un Fx dans l'exercice III est non compensable
- UE
  - Prise en compte de :
    - la note de l'examen (coef 5),
    - du projet sur le coloriage de carte (coef 1)
    - du projet sur les robots (coef 1)
- Si la note de l'examen est meilleure que la note calculée avec les notes de projets, c'est la note d'examen qui prime
- Examens de session 2
  - 1 sujet (mais plusieurs exercices) différent par AF non validé
  - 1h par AF

Retour sur examen du semestre 6

Retour sur les examens

Exercice I

1) Écrire une **fonction** *est_valide* qui retourne *True* si la chaîne fournie en paramètre est validé, c'est-à-dire qu'elle ne contient que des caractères autorisés (**chiffres** et "**+**") et que les opérateurs "**+**" sont bien placés (ni au début, ni à la fin, ni en doublon) et *False* sinon

Est_valide version 1

```python
def est_valide(expression):
    """..."""
    # test chaine vide
    if expression=="":
        return False

    # test position d'un + en début ou fin de chaîne
    if expression[0] == '+' or expression[-1] == '+':
        return False

       # test caractères autorisés
    for i in range(0, len(expression)-1):
        car = expression[i]
        if car == '+':
            if expression[i+1] == "+":
                # on a deux + qui se suivent l'expression n'est pas valide ...
                return False
        elif not car.isdigit():
            # si le caractère n'est pas un '+' il faut que ce soit un chiffre
            return False

    # on teste si le dernier caratère est bien un chiffre
    if not expression[-1].isdigit():
        return False

    # si tous les autres tests sont bons, on peut retourner True
    return True
```

- Tester si le caractère ‘+’ est en début ou en fin de chaîne en dehors de l’itération, sinon cela signifie que vous le faites à chaque étape de l’itération
- Inutile de transformer les chaînes de caractères en liste
- On veut observer la valeur de expression[i+1] il faut s’arrêter un rang avant la fin pour ne pas avoir des problèmes d’indices « out of range »
- Il faut tester si tous les caractères de la chaîne sont des chiffres ou des ‘+’
- Comme on a exclu le dernier caractère il faut tester si c’est bien un chiffre.
- Return True dans tous les autres cas à la sortie de l’itération

Variante

```
# test de non doublement d'un '+'
if "++" in expression:
    return False
```

- Inutile de le tester dans l'itération,
- cela n'empêche pas de tester si les autres caractères sont des chiffres

```
def est_valide(expression):
    opdes = expression.split('+')
    for element in opdes:
        if not element.isdigit():
            return False

    return True
```

- En découpant avec un split on récupère les caractères qui ne sont pas des '+' :
  ex : pour '13+823+8' → ['13', '823', '8']
- A combiner avec la solution précédente

Problèmes classiques rencontrés

Problèmes algorithmiques

- '123' est bien formé →
- if chaine[0] == '+' or expression[-1] == '+':
- Return True
- Tester chaine[k]==chaine[k+1] and chaine[k]=='+'

if chaine.isdigit() doit retourner True

à sortir de l'itération pour ne tester qu'une fois

à la sortie de l'itération qd tout a été vérifié

inverser les tests

tester en 1er lieu si le caractère est un '+'

car sinon le test ne sert à rien

Problèmes de programmation

- Inutile de transformer les chaînes de caractères en liste
- Confusion entre *rang* et *caractère* dans l'itération sur la chaîne

Exercice I

2) Écrire une **fonction** *evalue* qui évalue la valeur d'une expression de ce type. Si l'expression n'est pas bien formée, cette fonction retourne *False.*

Evalue

```python
def evalue(expression):
    """

    :param expression: expression arithmétique exprimant une somme de valeur sous forme de chaîne
    :return:           l'expression évaluée
    """
    if not est_valide(expression):
        return False

    # on récupère les opérandes de l'expression en 'splitant' l'expression
    lvaleurs = expression.split('+')

    # on calcule la somme
    som = 0
    for val in lvaleurs:
        som += int(val)

    return som
```

- Vérifier que la chaîne est bien construite sinon l'évaluation ne pourra pas se faire correctement
- Initialisation de la liste obtenue avec le *split* inutile
- Le *split* conserve les sous-chaînes de caractères qui correspondent à des nombres
- Le *split* conserve des chaînes de caractères, il faut convertir en entier

Evalue – variante

```python
def evalue(expression):
    """

    :param expression: expression arithmétique exprimant une somme de valeur sous forme de chaîne
    :return:           l'expression avaluée
    """
    if not est_valide(expression):  # si l'expression n'est pas valide on ne peut pas l'évaluer
        return False

    # on récupère les opérandes de l'expression en 'splitant' l'expression avec '+' comme séparateur
    lvaleurs = expression.split('+')
    # on convertit les valeurs en entiers
    lvaleurs = [int(val) for val in lvaleurs]

    return sum(lvaleurs)
```

- Liste en compréhension pour convertir les caractères en entiers
- Somme des entiers de la liste

Exercice II – Bilan données expérimentales

Exemple de fichier à traiter

```
# 25 janvier 2020
0.0051 0.0274 0.8883 0.5716 0.1246
# 26 mars 2020
1.4532 1.7710 1.7266 1.8425 1.3793
# 2 avril 2020
5.2299 5.7239 5.7531 5.2266 5.6840
```

Pour obtenir

```
# 25 janvier 2020
min : 0.0051 max : 0.8883
# 26 mars 2020
min : 1.3793 max : 1.8425
# 2 avril 2020
min : 5.2266 max : 5.7531
```

Remarques générales

```
def bilan(filename):

    fdat = open(filename, 'r')
    fbil = open('bilan.out', 'w')

    for ligne in fdat:

        if ligne[0] == '#':
            fbil.write(ligne)

        else:
            nb = [float(val) for val in ligne.split()]

            fbil.write('min : ' + str(min(nb)) + 'max : ' + str(max(nb)) + '\n ')

    fdat.close()
    fbil.close()
```

- Pas de *readline* ou *lire_ligne* en plus !
  for ligne in fdat fait déjà la lecture

- *ligne* contient déjà '\n' à la fin

- *split*() plutôt que *split*(' ')
- Attention ce sont des nombres décimaux : *float* et non *int*
- *split* renvoie une liste de chaînes de caractères (donc pas besoin de mettre le résultat dans une liste supplémentaire et ne pas oublier *float*)

- write renvoie une unique chaîne de caractères => pas de ',' et ne pas oublier *str*(..) au niveau des nombres

Erreurs classiques

- ✓ Ouverture du fichier (mode d'ouverture, récupération de l'objet, etc.)
- ✓ Lecture du fichier (for ligne in fichier, sans double lecture (readline ou lire_ligne), dans un objet fichier (et non une chaîne))

E

- ✓ Fermeture d'un fichier

D

- ✓ Ecriture dans un fichier avec + et non ,
- ✓ Bonne utilisation du type (*str* pour écriture, nombre pour min et max)

C

- ✓ Utilisation de *split* adaptée (séparateur, sortie)
- ✓ Utilisation adaptée de \n

B

- ✓ *float* (et non *int*)
- ✓ Pas de ' ' en séparateur

A

Exercice III – Bataille navale

On souhaite réaliser le jeu de la bataille navale, on s'appuie sur la modélisation suivante :

- Le jeu est caractérisé par la taille du plateau de jeu (carré) et une liste de bateaux
- chaque bateau est caractérisé par : une liste d'éléments et un état général (intact=0, touché=1 ou coulé=2)
- un élément est caractérisé par sa position sur le plateau de jeu (ligne, colonne) et son état (intact=0 ou touché=1)

Correction Exercice 3 – Bataille Navale – **Classe Element**

un élément est caractérisé par sa position sur le plateau de jeu (ligne, colonne) et son état (intact=0 ou touché=1)

```
class Element:

    def __init__(self, position):

        self.position_ = position
        self.etat_ = 0


    def getEtat(self):

        return self.etat_


    def est_touche(self, coup):

        if self.position_ == coup:
            self.etat_ = 1
```

- Paramètres/attributs ligne, colonne aussi possibles
- Ne pas oublier *self*
- Pas de paramètre *etat* ou *etat* = 0, car il ne peut prendre d'autre valeur que 0

- Nom de fonction : *get_etat, getetat*, etc., mais pas « accesseur »…
- Pas *__str__* (surcharge de *print*(), *str*())

- On utilise les attributs stockés dans *self*
- Pas de valeur retournée
- Il faut modifier *self.etat_*

Correction Exercice 3 – Bataille Navale – **Classe Bateau**

chaque bateau est caractérisé par : une liste d'éléments et un état général

```
class Bateau:

    def __init__(self, lg, l, c, direc):

        self.lg_ = lg
        self.etat_ = 0

        if direc == 'h':
            self.lelts_ = [Element((l, c+i)) for i in range(lg)]
        else:
            self.lelts_ = [Element((l+i, c)) for i in range(lg)]

    def getEtat(self):
        return self.etat_

    def __lt__(self, other):

        if self.lg_ < other.lg_:
            return True

        return False
```

- Pas une sous-classe de Element
- Pas de paramètre etat ou etat = 0, car il ne peut prendre d'autre valeur que 0.
- Instanciation : Element(…) plutôt que Element.__init__(…)

- Liste d'éléments et non de tuples
- Attention à ne pas mettre 2x le 1er élément

- 2 paramètres *self* et *other* (objets de la classe bateau et non longueurs)
- Renvoie un booléen (*True* or *False*) et non un objet
- Attention au sens de la comparaison (on regarde si *self* < *other*)

Correction Exercice 3 – Bataille Navale – **Classe Bateau (suite)**

```
def est_touche(self, coup):

    for elt in self.lelts_:
        elt.est_touche(coup)
```

- Utilisation du polymorphisme ad hoc
  - Pas de *element.est_touche(elt, coup)*
  - Pas de *est_touche_element* et *est_touche_bateau*

```
def maj_etat(self):

    n = 0
    for elt in self.lelts_:
        n += elt.getEtat()

    if n == self.lg_:
        self.etat_ = 2

    elif n > 0:
        self.etat_ = 1
```

- On peut rajouter le *else* : *self.etat_* = 0

Correction Exercice 3 – Bataille Navale – **Sous-classes**

```
class Croiseur(Bateau):

    def __init__(self, l, c, direc):

        Bateau.__init__(self, 3, l, c, direc)
```

- La longueur est forcée au niveau du constructeur de la classe mère (pas de lg = 3)
- Ne pas oublier self

```
class SousMarin(Bateau):

    def __init__(self, l, c, direc):

        Bateau.__init__(self, 2, l, c, direc)
        self.plonge_ = False
```

- Attribut *plonge* en plus, qui vaut toujours *False* à l'initialisation (donc pas besoin de paramètre)

```
    def est_touche(self, coup):

        if not self.plonge_:
            Bateau.est_touche(self, coup)
```

- Seule la méthode *est_touche* est à reprendre
- Utiliser le polymorphisme d'héritage au lieu de recopier le contenu de *est_touche*

Correction Exercice 3 – Bataille Navale – **Classe Bataille**

```
class Bataille:

    def __init__(self, dimension, lbateaux):

        self.dim_ = dimension
        self.lbateaux_ = lbateaux


    def fin_partie(self):

        for bat in self.lbateaux_:

            if bat.getEtat() != 2:
                return False

        return True
```

- On s'arrête dès qu'un bateau non coulé est trouvé.
- Compter les bateaux coulés fonctionne mais est moins efficace

Évaluation

- ✓ Création de classe : class NomClasse
- ✓ Méthode constructeur : def __init__(self, …)
- ✓ Modification d'un attribut : self.att_ = …
- ✓ Appel d'une méthode : obj.methode(…)
- ✓ Création d'une sous-classe : class SousClasse(ClasseMere)
- ✓ Méthode constructeur sous classe : ClasseMere.__init__(self, …)

Niveau E

- ✓ Méthode accesseur
- ✓ Méthode __lt__
- ✓ Spécialisation
- ✓ Polymorphisme ad hoc

Niveau C

- ✓ Attribut imposé (classe et sous-classe)
- ✓ Polymorphisme d'héritage

Niveau A

Exercice IV – Coloriage de cartes

Théorème des 4 couleurs

```python
def theoreme_4_couleurs(dico_couleur):
    """

    :param dico_couleur: dictionnaire associant à chaque région sa couleur
    :return: True s'il y a au plus 4 couleurs différentes False sinon
    """
    lcouleurs = []      # liste des couleurs utilisées pour colorier la carte
    for region, coul in dico_couleur.items():
        # pour chaque région stockée dans le dictionnaire
        # on ajoute la couleur dans la liste des couleurs de façon à ce que la couleur d'apparaisse qu'une fois

        if coul not in lcouleurs:
            lcouleurs.append(coul)
            # dès que la longueur de la liste des couleurs dépasse 4 on peut retourn
            if len(lcouleurs)>4:
                return False

    return True
```

- Pas besoin de la carte, le dictionnaire contient les informations utiles
- Ou bien
  for region in dico_couleurs:
  coul = dico_couleurs[region]
- Utilisation de l'opérateur *in* Plus rapide que de faire la recherche soi-même à l'aide d'une itération

Voisin commun à deux régions

```
def voisin_commun(dico_voisins, reg1, reg2):
    """

    :param dico_voisins:    dictionnaire des voisins
    :param reg1:            région
    :param reg2:            région
    :return:                True si reg1 et reg2 ont un voisin commun
    """

    for voisin in dico_voisins[reg1]:
        if voisin in dico_voisins[reg2]:
            return True

    return False
```

- On parcourt la liste des voisins d'une des régions

- On s'arrête dès qu'on a trouvé un voisin commun

Inutile d'aller plus loin