Idées pour les activités de pair-programming

Thème : dictionnaire

Exercice 1 : Chiffres romains

La numérotation romaine est basée sur 6 lettres majuscules : `I` qui vaut 1, `V` qui vaut 5, `X` vaut 10, `L` vaut 50, `C` vaut 100 et `M` vaut 1000.

1. Définir une constante `VALEUR` de type `dict` permettant de stocker les associations entre le *chiffre romain* et sa valeur décimale.
2. Écrire une fonction `decimale` qui prend une chaine de carcatères représentant l'écriture d'un nombre en numérotation romaine et qui donne sa valeur décimale.

La méthode :

Le principe est de commencer à zéro, puis d'additionner ou retrancher les valeurs de tous les symboles représentant un nombre écrit en chiffres romains en commençant par la gauche.

- Si un symbole a une valeur **supérieure ou égale** à celui qui se trouve à sa droite, il est **ajouté**. Par exemple `XVI` vaut 16 : `X` vaut plus que `V` qui est à sa droite, on ajoute donc 10, puis `V` est aussi supérieur à `I` on ajoute 5 et enfin on ajoute 1 pour le `I`
- Si un symbole a une valeur **strictement inférieure** à celui qui se trouve à sa droite, il est **retranché**. Par exemple `XIV` vaut 14 : tout d'abord 10 est ajouté car `X` a une valeur supérieure à celle de `I`, puis on retranche 1 car `I` a une valeur inférieure à celle de `V`

Travail en binome

- Celui ou celle qui a la main sur l'éditeur commence à réfléchir au code de la fonction `decimale`, pendant que l'autre réfléchit à une série d'appels possibles (pour les tests)
- N'hésitez pas à utiliser le chat pour communiquer et à demander la main sur l'éditeur

Solution et barème

```
VALEUR = {"I": 1, "V": 5, "X": 10, "L": 50, "C": 100, "D": 500, "M":
1000}


def decimale(romain):
    n = 0
    for i in range(len(romain)-1):
        valeur_i, valeur_suivant = VALEUR[romain[i]],
VALEUR[romain[i+1]]
        if valeur_i >= valeur_suivant:
            n += valeur_i
        else:
            n -= valeur_i
    return n + VALEUR[romain[-1]]
```

```python
def test():
    # Tests

    assert decimale("XVI") == 16
    assert decimale("MMXXII") == 2022
    assert decimale("CDII") == 402
    assert decimale("XLII") == 42
    print('ok')

test()
```

```
ok
```

Je propose comme points :

- 1 pt si le dictionnaire `VALEUR` est présent et correct
- 1 pt si il y a utilisation du dictionnaire dans le corps de la fonction
- 1 pt s'il y a au moins 2 tests corrects (pas forcément avec des `assert` mais juste des appels corrects)

Accès et validation du point bonus

L'accès à l'activité moodle est prévue pour 45mn. Un test avec une unique question sera ouvert sur le même créneau de 45mn : il s'agira pour l'étudiant de rentrer un code de 3 lettres que tu pourrais donner à la fin du questionnaire post-activité. Pour l'activité 1 le code est : RDB

Expression bien parenthésée

On considère dans cet exercice un parenthésage avec les couples `( )`, `{ }`, `[ ]` et `< >`. On dira qu'une expression est bien parenthésée si chaque symbole ouvrant correspond à un symbole fermant et si l'expression contenue à l'intérieur est elle-même bien parenthésée.

Exemples d'expressions bien parenthésées

- `(2 + 4) * 7`
- `tableau[f(i) - g(i)]`
- `#include <stdio.h> int main(){int liste[2] = {4, 2}; return (10*liste[0] + liste[1]);}`

Exemples d'expressions mal parenthésées

- `(une parenthèse laissée ouverte` : cette phrase est mal parenthésée car il n'y a pas de fermante associée à `(`.
- `{<(}>)` : mauvaise imbrication.
- `c'est trop tard ;-)` : pas d'ouvrante associée à `)`.

En utilisant un dictionnaire pour associer les différentes paires de délimiteurs, écrire une fonction `bien_parenthesee` qui prend une chaine de caractère en paramètre et renvoie `True` si la chaine est bien parenthésée et `False` sinon.

Aide : Vous pouvez utilisez ces deux constantes, définissant les délimiteurs ouvrants et fermants :

```
OUVRANT = '({[<'
FERMANT = ')}]>'
```

Solution et barème

```
OUVRANT = '({[<'
FERMANT = ')}]>'
DELIMITEUR = {FERMANT[i]: OUVRANT[i] for i in range(4)}

def bien_parenthesee(phrase):
    pile = []
    for c in phrase:
        if c in OUVRANT:
            pile.append(c)
        elif c in FERMANT:
            if pile == [] or DELIMITEUR[c] != pile.pop():
                return False
    return pile == []
```

Pour les points :

- 1 pt s'il y a un dictionnaire associant ouvrants et fermants
- 1 pt de plus s'il s'agit d'une construction en compréhension
- 1 pt pour l'utilisation du dictionnaire dans le corps de la fonction

```
def test_2():
    assert bien_parenthesee('(2 + 4) * 7') == True
    assert bien_parenthesee('tableau[f(i) - g(i)]') == True
    assert bien_parenthesee('#include <stdio.h> int main(){int
liste[2] = {4, 2}; return (10*liste[0] + liste[1]);}') == True
    assert bien_parenthesee('(une parenthèse laissée ouverte') ==
False
    assert bien_parenthesee('{<(}>)') == False
    assert bien_parenthesee("c'est trop tard ;-)") == False
    print('ok')

test_2()
```

```
ok
```

Accès et point bonus

Toujours sur une durée de 45mn dans le créneau qui suit la première activité. Le code pourrait être KTM.

```

```