Examen, 2h, avec documents

NOM : **Prénom :**

Consignes : Téléphones, ordinateurs et calculatrices interdits.
Documents de cours/TD/TP autorisés.
Aucune question possible : en cas d'ambiguïté dans le sujet, signaler le choix fait sur la copie.
Sauf indication contraire, le langage utilisé sera Python.

Conseil : Lire complètement le sujet avant de commencer à y répondre.

Barème indicatif :

exercice	1	2	3	4
points	4	3	7	6

Exercice 1 : Suite de Fibonacci
Les termes de la suite de Fibonacci sont définis par la relation de récurrence suivante :

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \text{ si n} \geq 2$$

1. Écrire un sous-programme qui calcule le n^{e} terme de cette suite en utilisant la récursivité.
2. Écrire un sous-programme qui calcule le n^{e} terme de cette suite de manière itérative.

Exercice 2 : Multiples de 3
1. Écrire un générateur qui permet d'avoir tous les multiples de 3.
2. Écrire un sous-programme qui affiche les 10 premiers multiples de 3 en utilisant le générateur précédent.

Exercice 3 : Sélectionner les éléments d'une liste
Dans cet exercice, on utilisera le langage Python.
1. Écrire un sous-programme `elements_positifs` qui prend en paramètre une liste et qui retourne la liste des élément positifs ou nuls de cette liste.

Par exemple, appliqué sur la liste [1, -1, 0, -4, 2.5, -3], il retournera [1, 0, 2.5].

2. Écrire un programme de test avec `pytest` de ce sous-programme qui vérifie l'exemple précédent.
3. Écrire un sous-programme `entiers_pairs` qui retourne la liste des entiers pairs d'une liste d'entiers.

Par exemple, appliqué sur [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], le résultat de ce sous-programme sera [0, 2, 4, 6, 8].

4. Écrire un sous-programme `chaines_en_minuscules` qui retourne la liste des chaînes de caractères en minuscules (on utilisera la fonction `islower()` définie sur la classe **`str`**) d'une liste d'objets.

Par exemple, appliqué sur [1, 'X', 'un', 1.2, 'deux'], le résultat de ce sous-programme sera ['un', 'deux'].

5. On remarque que ces sous-programmes sont sur le même modèle : on crée une nouvelle liste à partir d'une liste existante en ne conservant que les éléments qui respectent un certain critère (être positif ou nul, être pair, être une chaîne en minuscules). On peut donc écrire une fonction générale **`filter`** qui prend en paramètre un critère : elle retourne la liste des éléments de la liste reçue en paramètre qui respectent le critère.

5.1. Quel est le type de ce « critère » en Python ?

5.2. Écrire le sous-programme `filtrer`.

5.3. Réécrire les sous-programmes `elements_positifs` et `chaines_en_minuscules` en utilisant filtrer.

6. Dans tous les sous-programmes précédents, nous avons travaillé sur des listes. Est-ce qu'ils pourraient fonctionner sur d'autres types ? Quel serait le type le plus général sur lequel ils fonctionnent ?

Exercice 4 : Expressions arithmétiques

On considère les expressions arithmétiques limitées aux constantes entières, la somme et la différence de deux expressions arithmétiques. Deux traitements sont définis, le premier qui affiche l'expression en notation postfixe, le second qui évalue l'expression. Ces expressions et traitements sont modélisés en OCaml (listing 1) et en Python (listing 2).

1. On veut ajouter la multiplication. Indiquer quelles modifications il faut réaliser sur les listings 1 et 2 et les réaliser.

2. On considère l'expression suivante (2+3)*4. Écrire l'expression OCaml et l'expression Python qui construisent cette expression. On l'appellera `e`.

3. Écrire un nouveau traitement (en OCaml et Python) qui calcule le nombre de traits utilisés par les opérateurs d'une expression. On compte 0 traits pour une constante (pas d'opérateur), un trait pour une soustraction, deux traits pour une addition et 3 traits pour une multiplication.

```
type expression =
    Constante of float
    | Somme of (expression * expression)
    | Difference of (expression * expression)
;;

let rec to_postfixe e =
    match e with
    | Constante v -> string_of_float (v)
    | Somme (e1, e2) -> (to_postfixe e1) ^ " " ^ (to_postfixe e2) ^ " +"
    | Difference (e1, e2) -> (to_postfixe e1) ^ " " ^ (to_postfixe e2) ^ " -"
;;

let rec valeur e =
    match e with
    | Constante v -> v
    | Somme (e1, e2) -> (valeur e1) +. (valeur e2)
    | Difference (e1, e2) -> (valeur e1) -. (valeur e2)
;;
```

Listing 1 – Les expressions en OCaml

```python
class Expression:
    pass

class Constante(Expression):

    def __init__(self, valeur):
        self.__valeur = float(valeur)

    @property
    def valeur(self):
        return self.__valeur

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

class Somme(Expression):

    def __init__(self, operande_gauche, operande_droite):
        self.__operande_gauche = operande_gauche
        self.__operande_droite = operande_droite

    def __str__(self):
        return str(self.__operande_gauche) + ' ' \
                + str(self.__operande_droite) + ' +'

    @property
    def valeur(self):
        return self.__operande_gauche.valeur + self.__operande_droite.valeur

class Difference(Expression):

    def __init__(self, operande_gauche, operande_droite):
        self.__operande_gauche = operande_gauche
        self.__operande_droite = operande_droite

    @property
    def valeur(self):
        return self.__operande_gauche.valeur - self.__operande_droite.valeur

    def __str__(self):
        return str(self.__operande_gauche) + ' ' \
                + str(self.__operande_droite) + ' -'
```

Listing 2 – Les expressions en Python