Examen, 2h, avec documents
Corrigé

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.
Solution : Version collant à la définition de la suite :

```
def fibonacci_recursif(n):
    '''
    Remarque : cette version colle à la définition de la suite mais n'est pas
    efficace car on va calculer plusieurs fois les mêmes termes.
    '''
    if n <= 1:
        return n
    else:
        return fibonacci_recursif(n-1) + fibonacci_recursif(n-2)
```

Version qui évite de calculer plusieurs fois les mêmes termes (récursivité terminale).

```
def fibonacci_recursif_terminal(n):

    def fib(n, terme, suivant):
        '''
        Version avec récursivité terminale : la résultat est connu lors du
        dernier appel récursif.
        '''
        if n == 0:
            return terme
        else:
            return fib(n-1, suivant, terme + suivant)

    return fib(n, 0, 1)
```

2. Écrire un sous-programme qui calcule le n^{e} terme de cette suite de manière itérative.
Solution :

```
def fibonacci_iteratif(n):
    terme = 0   # le terme courant de la suite
    suivant = 1 # le terme suivant de la suite
    for i in range(n):
        terme, suivant = suivant, terme + suivant
    return terme
```

Exercice 2 : Multiples de 3

1. Écrire un générateur qui permet d'avoir tous les multiples de 3.
Solution :

```
1 def multiples(nombre=3):
2     '''Générateur produisant les multiples de nombre.'''
3     n = nombre
4     while True:
5         yield n
6         n += nombre
```

2. Écrire un sous-programme qui affiche les 10 premiers multiples de 3 en utilisant le générateur précédent.
Solution :

```
1 def dix_premiers_multiples_de_3():
2     multiples_3 = multiples(3)
3     for i in range(10):
4         print(next(multiples_3))
```

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].

Solution :

```python
def elements_positifs(iterable):
    resultat = []
    for x in iterable:
        if x >= 0:
            resultat.append(x)
    return resultat


def elements_positifs_comprehension(iterable):
    return (x for x in iterable if x >= 0)
```

Remarque : La version en compréhension n'était pas demandée.

2. Écrire un programme de test avec `pytest` de ce sous-programme qui vérifie l'exemple précédent.

Solution :

```python
def executer_test_elt_positifs(elt_positifs):
    assert list(elt_positifs([1, -1, 0, -4, 2.5, -3])) == [1, 0, 2.5]
    assert list(elt_positifs([-1, -4.5, -3])) == []
    assert list(elt_positifs([])) == []

def test_elements_positifs():
    executer_test_elt_positifs(elements_positifs)

def test_elements_positifs_comprehension():
    executer_test_elt_positifs(elements_positifs_comprehension)
```

Remarque : Ici, une seule fonction de test avec les assert était attendue.

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].

Solution :

```python
def entiers_pairs(iterable):
    resultat = []
    for x in iterable:
        if x % 2 == 0:
```

```
5               resultat.append(x)
6       return resultat
7
8   def entiers_pairs_comprehension(iterable):
9       return (x for x in iterable if x % 2 == 0)
```

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'].

Solution :

```
1   def chaines_en_minuscules(iterable):
2       resultat = []
3       for x in iterable:
4           if isinstance(x, str) and x.islower():
5               resultat.append(x)
6       return resultat
7
8   def chaines_en_minuscules_comprehension(iterable):
9       return (x for x in iterable if isinstance(x, str) and x.islower())
```

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 ?

Solution : C'est une fonction qui prend un élément (d'un type quelconque) et retourne un booléen (vrai si le critère est vérifiée, faux sinon).

5.2. Écrire le sous-programme `filtrer`.

Solution :

```
1   def filtrer(iterable, critere):
2       resultat = []
3       for x in iterable:
4           if critere(x):
5               resultat.append(x)
6       return resultat
7
8   def filtrer_comprehension(iterable, critere):
9       return (x for x in iterable if critere(x))
```

```ocaml
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

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

Solution :

```python
def elements_positifs_filtrer(iterable):
    return filtrer(iterable, lambda x : x >= 0)
```

```python
def chaines_en_minuscules_filtrer(iterable):
    return filtrer(iterable, lambda x : isinstance(x, str) and x.islower())
```

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 ?

Solution : On utilise un **`for`**. Ceci fonctionne donc sur un liste mais plus généralement sur un itérable.

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).

```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

1. On veut ajouter la multiplication. Indiquer quelles modifications il faut réaliser sur les listings 1 et 2 et les réaliser.
Solution : En OCaml : On modifie la définition de Expression pour ajouter la possibilité de faire Produit avec deux expressions et on ajoute ce cas dans tous les traitements.

En Python : On ajoute simplement une classe Produit, sur le même modèle que Difference en remplaçant « - » par « * ».

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.
Solution :

En OCaml :

```
let e = Produit(Somme(Constante(2), Constante(3)), Constante(4));;
```

En Python :

```
e = Produit(Somme(Constante(2), Constante(3)), Constante(4))
```

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.
Solution : En OCaml : On ajoute un nouvelle fonction `nombre_traits` qui fait un traitement par induction sur la structure d'un expression.

En python : On ajoute la méthode `nombre_traits` sur toutes les classes concrètes.
Solution :

```
type expression =
    Constante of float
    | Somme of (expression * expression)
    | Difference of (expression * expression)
    | Produit 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) ^ " -"
    | Produit (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)
```

```
    | Produit (e1, e2) -> (valeur e1) *. (valeur e2)
;;

let rec nombre_traits e =
    match e with
    | Somme (e1, e2) -> 2 + (nombre_traits e1) + (nombre_traits e2)
    | Difference (e1, e2) -> 1 + (nombre_traits e1) + (nombre_traits e2)
    | Produit (e1, e2) -> 3 + (nombre_traits e1) + (nombre_traits e2)
    | _ -> 0
;;
```

```
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)

    @property
    def nb_traits(self):
        return 0

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
```

```
33
34      @property
35      def nb_traits(self):
36          return 2 + self.__operande_gauche.nb_traits \
37                   + self.__operande_droite.nb_traits
38
39  class Difference(Expression):
40
41      def __init__(self, operande_gauche, operande_droite):
42          self.__operande_gauche = operande_gauche
43          self.__operande_droite = operande_droite
44
45      @property
46      def valeur(self):
47          return self.__operande_gauche.valeur - self.__operande_droite.valeur
48
49      def __str__(self):
50          return str(self.__operande_gauche) + ' ' \
51                  + str(self.__operande_droite) + ' -'
52
53      @property
54      def nb_traits(self):
55          return 1 + self.__operande_gauche.nb_traits \
56                   + self.__operande_droite.nb_traits
57
58  class Produit(Expression):
59
60      def __init__(self, operande_gauche, operande_droite):
61          self.__operande_gauche = operande_gauche
62          self.__operande_droite = operande_droite
63
64      @property
65      def valeur(self):
66          return self.__operande_gauche.valeur * self.__operande_droite.valeur
67
68      def __str__(self):
69          return str(self.__operande_gauche) + ' ' \
70                  + str(self.__operande_droite) + ' *'
71
72      @property
73      def nb_traits(self):
74          return 3 + self.__operande_gauche.nb_traits \
75                   + self.__operande_droite.nb_traits
```