Numéro d'anonymat (à remplir par un surveillant)

Examen Session 2 2015 – 2016
Éléments de Programmation I – 1I001
Durée : 2h

Documents autorisés : Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Python.

Le sujet comporte 22 pages. Ne pas désagrafer les feuilles.

Répondre directement sur le sujet, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le quadrillage permet de bien aligner les indentations.

Le barème indiqué pour chaque question n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : 66 points.

La clarté des réponses et la présentation des programmes seront appréciées. Les exercices peuvent être résolus dans un ordre quelconque. Il est possible (et souvent utile), pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.

Remarque : Pour les réponses de cet examen, on pourra considérer que la bibliothèque de fonctions mathématiques a été importée avant les fonctions à écrire. Sauf mention contraire explicite, seules les primitives Python présentes sur la carte de référence peuvent être utilisées.

Important : sauf exception (ex.: fonction mystère) **il n'est pas demandé de donner la description en français dans les spécifications des fonctions ni de proposer un jeu de tests**. Lisez bien la question : demande-t-on la définition avec signature et hypothèse(s) éventuelle(s) ? ou demande-t-on une définition seule *sans* signature ni hypothèse ? Ou alors demande-t-on uniquement la signature et hypothèse(s) éventuelle(s) ?

L'examen est composé de quattre exercices indépendants.

Exercice 1 : Polynômes

Le but de cet exercice est de définir un ensemble de fonctions capables de manipuler des polynômes.

Un polynôme à une variable est un objet mathématique que l'on définit comme étant une somme de termes : $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Chaque terme du polynôme est de la forme a_kx^k dans lequel x est la variable du polynôme, k le *degré* du terme et a_k son coefficient.

Ainsi, le polynôme $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ est complètement défini par la liste des coefficients $[a_0, a_1, \ldots, a_n]$. Par exemple, le polynôme $P_1(x) = 5 - 2x + x^2$ est déterminé par la liste de ses coefficients `[5, -2, 1]`. De même, le polynôme $P_2(x) = 1 + x^4$ est représenté par `[1, 0, 0, 0, 1]` et le polynôme $P_3(x) = -x^4$ est représenté par `[0, 0, 0, 0, -1]`. Par ailleurs, on décide de représenter le polynôme nul par la liste `[0]`.

On s'intéresse, dans cet exercice, aux polynômes à coefficients entiers et nous pourrons utiliser l'alias de type `Poly` pour `list[int]`.

On supposera également dans la suite que les polynômes donnés en exemple ci-dessus ont été définis par les variables suivantes :

```
>>> poly1 = [5, -2, 1]
>>> poly2 = [1, 0 , 0, 0, 1]
>>> poly3 = [0, 0, 0, 0, -1]
>>> polynul = [0]
```

Remarque : Dans tout ce qui suit, nous pourrons supposer, sans le rajouter en hypothèse dans les spécifications, que tout polynôme contient au moins 1 coefficient, c'est à dire que la liste qui le représente est non vide.

Question 1.1 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `poly_applique` qui, étant donné un polynôme `P` et un nombre `x`, renvoie la valeur du polynôme `P` en `x`.

Par exemple :

```
>>> poly_applique(poly1, 0)
5.0                          # P1(0) = 5 - 2*0 + 0^2 = 0
>>> poly_applique(poly1, 1)
4.0                          # P1(1) = 5 - 2*1 + 1^2 = 4
>>> poly_applique(poly1, 2)
5.0                          # P1(2) = 5 - 2*2 + 2^2 = 5
>>> poly_applique(poly3, 2)
-16.0                        # P3(2) = -2^4 = -16
>>> poly_applique(polynul, 2)
0.0
```

Réponse

```
def poly_applique(P,x):
    """ Poly * Number -> float"""

    #r: float
    r=0.0

    #i: int
    for i in range(0,len(P)):
        r = r + P[i]*(x**i)

    return r
```

Question 1.2 : [3/66]

On rappelle que l'addition de deux polynômes P et P' est donné par le polynôme P'' dont chacun des coefficients correspond à l'addition des coefficients des termes de même degré dans P et P'.

Attention, les degrés des deux polynômes ne sont pas forcément égaux. Si $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ et $P'(x) = b_0 + b_1x + b_2x^2 + \ldots + b_px^p$ avec $p > n$, alors l'addition de P et P' donne le polynôme $P''(x) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \ldots + (a_n + b_n)x^n + b_{n+1}x^{n+1} + \ldots + b_px^p$

Ainsi l'addition de P_1 et P_2 de l'exercice précédent donne le polynôme P suivant :

$$\begin{array}{cc|ccccc} & P_1 & 5 & -2 & 1 & & \\ + & P_2 & 1 & 0 & 0 & 0 & 1 \\ \hline & P & 6 & -2 & 1 & 0 & 1 \end{array}$$

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `poly_add` qui renvoie l'addition de deux polynômes. Par exemple :

```
>>> poly_add(poly1, poly2)
[6, -2, 1, 0, 1]
>>> poly_add(poly1, polynul)
[5, -2, 1]
>>> poly_add(poly2, poly3)
[1, 0, 0, 0, 0]
```

Réponse

```
def poly_add(P1,P2):
    """ Poly * Poly -> Poly"""

    #Poly
    PR = []

    #i: int
    i = 0

    while i < len(P1) and i < len(P2):
        PR.append(P1[i] + P2[i])
        i = i + 1

    if i < len(P1):
        PR = PR + P1[i:len(P1)]

    if i < len(P2):
        PR = PR + P2[i:len(P2)]

    return PR
```

Le résultat de l'addition de `poly2` et `poly3` dans l'exemple précédent, bien que corrrect, n'est pas canonique car les coefficients les plus élevés sont nuls. La bonne représentation du résultat devrait être le polynôme constant `[1]`.

Ainsi, on dit que P est sous forme *canonique* si le coefficient de plus fort degré de P est non nul. Tout polynôme a une forme canonique et les questions suivantes ont justement pour but d'écrire une fonction mettant un polynôme sous sa forme canonique.

Question 1.3 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `dernier_nonnul` qui, étant donné un polynôme `P`, renvoie l'indice du dernier coefficient non nul du polynôme `P` ou -1 si tous les coefficients sont nuls.

Par exemple :

```
>>> dernier_nonnul(polynul)
-1
>>> dernier_nonnul(poly1)
2
>>> dernier_nonnul(poly2)
4
>>> dernier_nonnul(poly_add(poly2,poly3))
0
```

Réponse

```
def dernier_nonnul(P):
    """ Poly -> int"""

    #i: int
    i = len(P) - 1

    while i >=0 and P[i] == 0:
        i = i - 1

    return i
```

Question 1.4 : [2/66]

En déduire une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `normalise` qui, étant donné un polynôme `P`, renvoie la forme canonique de `P`.

Par exemple :

```
>>> normalise([1, 0, 1, 0, 1])
[1, 0, 1, 0, 1]
>>> normalise([1, 0, 1, 0])
[1, 0, 1]
>>> normalise(polynul)
[0]
>>> normalise(poly1)
[5, -2, 1]
>>> normalise(poly_add(poly2,poly3))
[1]
```

Réponse

```
def normalise(P):
    """ Poly -> Poly"""

    #indice: int
    indice = dernier_nonnul(P)

    if indice == -1:
        return [0]
    else:
        return P[0:indice+1]
```

Question 1.5 : [3/66]

On rappelle que, pour tout polynôme $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ de degré n non nul, le polynôme dérivé de P est défini par $P'(x) = a_1 + 2*a_2x + 3*a_3x^2 + \ldots + i*a_ix^{i-1} \ldots + n*a_nx^{n-1}$. Ainsi, la dérivé de P_1 est par exemple le polynôme $-2 + 2x$, celle de P_2 est $4x^3$ et celle de P_3 est $-4x^3$. On rappelle également que la dérivée de tout polynôme constant (y compris nul) est le polynôme nul.

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `derive` qui, étant donné un polynôme `P`, renvoie le polynôme dérivé de `P`.

Par exemple

```
>>> derive(polynul)
[0]
>>> derive([3])
[0]
>>> derive(poly1)
[-2, 2]
>>> derive(poly2)
[0, 0, 0, 4]
>>> derive(poly3)
[0, 0, 0, -4]
```

Réponse

```
def derive(P):
    """ Poly -> Poly"""

    #PR: Poly
    PR = []

    if len(P) == 1:
        return [0]

    else:
        #i: int
        for i in range(1,len(P)):
            PR.append(i*P[i])

        return PR
```

Exercice 2 : Codage HTML des couleurs

Dans cet exercice tous les nombres sont positifs ou nuls.

Système hexadécimal

Le système hexadécimal est un système de numération en base 16. Il nécessite l'utilisation de 16 chiffres hexadécimaux, représentant les 16 premiers entiers naturels ; ce sont les 16 caractères : 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ; A ; B ; C ; D ; E ; F.

Remarque : dans cet exercice, les lettres correspondant à des chiffres hexadécimaux sont toujours des majuscules.

Un nombre hexadécimal est écrit comme une chaîne de caractères utilisant ces 16 chiffres hexadé-

cimaux. Sa valeur en base 10 vaut la somme des valeurs décimales des chiffres affectées de poids correspondant aux puissances successives du nombre 16 (les poids augmentant de la droite vers la gauche). Par exemple :

– l'entier hexadécimal 104 vaut $1 * 16^2 + 0 * 16 + 4 = 260$ en base 10,

– l'entier hexadécimal 4D5 vaut $5 * 16^0 + 13 * 16^1 + 4 * 16^2 = 1237$ en base 10.

Question 2.1 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `conv_chf` qui, étant donné un chiffre hexadécimal c, renvoie la valeur de c en base 10. Par exemple :

```
>>> conv_chf('3')
3
>>> conv_chf('A')
10
>>> conv_chf('F')
15
```

Attention : les chiffres hexadécimaux 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 sont des **caractères** et leurs valeurs en base 10 sont des **nombres**.

Indication : on peut utiliser la fonction `ord`, dont la spécification est donnée dans la carte de référence.

Remarque : les numéros de code des lettres majuscules de `'A'` à `'Z'` sont consécutifs, les numéros de code des chiffres de `'0'` à `'9'` sont consécutifs. Par exemple :

```
>>> ord('A')
65
>>> ord('B')
66
>>> ord('0')
48
>>> ord('1')
49
>>>
```

Réponse

```
def conv_chf(c):
    """ str -> int
    Hypothèse : c est un chiffre hexadécimal
    Retourne la valeur en base 10 du chiffre hexadécimal c."""

    if 'A' <= c and c <= 'F':
        return 10 + ord(c) - ord('A')
    else:
        return ord(c) - ord('0')
```

Question 2.2 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `hexa_vers_dec` qui, étant donnée une chaîne de caractères s représentant un entier hexadécimal, renvoie la valeur de s en base 10. Par exemple :

```
>>> hexa_vers_dec('4D5')
```

```
1237
>>> hexa_vers_dec('0')
0
>>> hexa_vers_dec('FF')
255
```

Réponse

```
def hexa_vers_dec(s):
    """ str -> int
    Hypothèse : s représente un entier hexadécimal
    Retourne la valeur en base 10 de l'entier hexadécimal s."""

    # res : int (l'entier décimal calculé)
    res = 0
    for i in range(len(s)):
        res = res * 16 + conv_chf(s[i])
    return res
```

Codage couleur HTML

Le codage HTML d'une couleur est représenté par une chaîne de 7 caractères : le premier caractère est un # et les six suivants sont des chiffres hexadécimaux (par exemple : #F9429E). Le bloc de 6 chiffres hexadécimaux est composé de 3 blocs de 2 chiffres hexadécimaux : le premier bloc correspond à la quantité de rouge, le deuxième à la quantité de vert et le troisième à la quantité de bleu, chaque quantité étant exprimée par un nombre entre 0 et 255 écrit en hexadécimal.

Ainsi, dans la couleur #F9429E :
- la quantité de rouge est F9 en hexadécimal, soit 249 en décimal ;
- la quantité de vert est 42 en hexadécimal, soit 66 en décimal ;
- la quantité de bleu est 9E en hexadécimal, soit 158 en décimal.

(Cela donne la couleur *rose bonbon*.)

Le codage HTML du rouge pur est #FF0000, celui du vert pur est #00FF00, celui du bleu pur est #0000FF.

Question 2.3 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `est_code_HTML` qui, étant donnée une chaîne de caractères s, renvoie la valeur `True` si s est un code couleur HTML, et `False` sinon.

Par exemple :

```
>>> est_code_HTML('#F9429E')
True
>>> est_code_HTML('#F9429G')
False
>>> est_code_HTML('#F9429E9')
False
>>> est_code_HTML('F9429E9')
False
```

Indication : on pourra écrire une fonction qui teste si une chaîne de caractères est un chiffre hexadécimal.

Réponse

```
def est_chf_hexa(c):
    """ str -> bool
    Retourne True si c est un chiffre hexadécimal, et False sinon."""
    return ('A' <= c and c <= 'F') or ('0' <= c and c <= '9')

def est_code_HTML(s):
    """ str -> bool
    Retourne True si s est un code couleur HTML, et False sinon."""

    if len(s) != 7:
        return False
    if s[0] != '#':
        return False
    for i in range(1, 7):
        if not(est_chf_hexa(s[i])):
            return False
    return True
```

Question 2.4 : [2/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `HTML_vers_RGB` qui, étant donné un code couleur HTML s, renvoie le codage RGB de cette couleur, c'est-à-dire le triplet formé de trois entiers : la quantité de rouge, la quantité de vert et la quantité de bleu calculées en base 10.

Par exemple :

```
>>> HTML_vers_RGB('#FF0000')
(255, 0, 0)
>>> HTML_vers_RGB('#00FF00')
(0, 255, 0)
>>> HTML_vers_RGB('#0000FF')
(0, 0, 255)
>>> HTML_vers_RGB('#F9429E')
(249, 66, 158)
```

Réponse

```
def HTML_vers_RGB(s):
    """ str -> tuple[int]
    Hypothèse : s représente le codage HTML d'une couleur
    Retourne le codage RGB de s."""

    return (hexa_vers_dec(s[1:3]), hexa_vers_dec(s[3:5]), \
            hexa_vers_dec(s[5:7]))
```

Exercice 3 : Base de données de films

Dans cet exercice, on s'intéresse à la définition de fonctions permettant de manipuler une base de données décrivant le casting de films. Dans une telle base de données, chaque film est associé aux

acteurs et actrices qui ont tourné dans ce film. À titre d'exemple, on considère la base de données suivante contenant des films du réalisateur Jean-Luc Godard :

Film	Acteurs
A bout de souffle	Belmondo, Seberg, Hanin
Pierrot le fou	Belmondo, Karina
Made in USA	Léaud, Karina
Bande a part	Karina, Brasseur, Frey
Une femme est une femme	Brialy, Karina, Belmondo

Une base de données de casting est donc représentée en Python par un dictionnaire de type `dict[str:set[str]]` avec :

— les noms des films, de type `str`, comme clés
— l'ensemble des acteurs, de type `set[str]`, comme valeurs associées.

Ainsi, l'exemple précédent donnerait le dictionnaire `Godard` suivant :

```
>>> Godard = {
       'a bout de souffle' : {'belmondo', 'seberg', 'hanin'},
       'pierrot le fou' : {'belmondo', 'karina'},
       'made in USA' : {'leaud', 'karina'},
       'bande a part' : {'karina', 'brasseur', 'frey'},
       'une femme est une femme' : {'brialy', 'karina', 'belmondo'}}
```

Dans la suite de cet exercice, on pourra utiliser l'alias de type `Casting` pour `dict[str:set[str]]`.

Question 3.1 : [2/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `ens_films` qui, étant donnée une base de données de casting, renvoie l'ensemble des films apparaissant dans cette base.

Par exemple :

```
>>> ens_films(Godard)
{'a bout de souffle',  'pierrot le fou', 'une femme est une femme',
  'made in USA', 'bande a part'}
```

Réponse

```
def ens_films(C):
    """ Casting -> set[str]"""

    #SR: set[str]
    SR = set()

    #film : str
    for film in C:
        SR.add(film)

    return SR
```

Variante avec compréhension :

```
def ens_films_comp(C):
    """ Casting -> set[str]"""

    return {film for film in C}
```

Question 3.2 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `films_avec` qui, étant donnés une base de données de casting C et le nom d'un acteur a, renvoie l'ensemble des films dans lesquels a a joué.

Par exemple :

```
>>> films_avec(Godard, 'belmondo')
{'a bout de souffle',  'pierrot le fou', 'une femme est une femme'}
>>> films_avec(Godard, 'karina')
{'pierrot le fou', 'une femme est une femme', 'bande a part', 'made in USA'}
>>> films_avec(Godard, 'dujardin')
{}
```

Réponse

```
def films_avec(C, a):
    """ Casting * str -> set[str]"""

    # SR : set[str]
    SR = set()

    # film : str
    for film in C:
        if a in C[film]:
            SR.add(film)

    return SR
```

Variante avec compréhension :

```
def films_avec_comp(C, a):
    """ Casting * str -> set[str]"""

    return {film for film in C if a in C[film]}
```

Question 3.3 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `tous_acteurs` qui, étant donné une base de données de casting C, renvoie l'ensemble de tous les acteurs ayant joué dans au moins un film de C.

Par exemple :

```
>>> tous_acteurs(Godard)
{'brialy', 'karina', 'belmondo', 'seberg', 'hanin', 'frey', 'leaud', 'brasseur'}
```

Réponse

```
def tous_acteurs(C):
    """ Casting -> set[str]"""

    # SR : set[str]
    SR = set()

    # film : str
    for film in C:
        # a : str
        for a in C[film]:
            SR.add(a)

    return SR
```

Variante avec compréhension :

```
def tous_acteurs_comp(C):
    """ Casting -> set[str]"""

    return {a for film in C for a in C[film]}
```

Question 3.4 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `partenaires` qui, étant donné une base de casting C et un acteur a, renvoie l'ensemble des acteurs ayant tourné au moins un film avec a.

Par exemple :

```
>>> partenaires(Godard,'leaud')
{'karina'}
>>> partenaires(Godard,'belmondo')
{'brialy', 'karina', 'seberg', 'hanin'}
>>> partenaires(Godard,'karina')
{'brialy', 'belmondo', 'brasseur', 'frey', 'leaud', 'brasseur'}
```

Réponse

```
def partenaires(C, a):
    """ Casting * str -> set[str]"""

    #SR : set[str]
    SR = set()

    #film : str
    for film in films_avec(C,a):
        SR = SR | (C[film] - {a})

    return SR
```

Variante avec compréhension :

```
def partenaires_comp(C, a):
    """ Casting * str -> set[str]"""

    return {act for film in films_avec(C,a) for act in C[film] if act != a}
```

Question 3.5 : [4/66]

En déduire une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `acteur_max` qui, étant donné une base de casting C, renvoie le nom de l'acteur ayant tourné avec le plus grand nombre d'acteurs différents.

Par exemple :

```
>>> acteur_max(Godard)
'karina'
```

Réponse

```
def acteur_max(C):
    """ Casting -> str"""

    #res : str
    res = ''

    #nb_max: int
    nb_max = 0

    #acteurs: set[str]
    acteurs = tous_acteurs(C)

    #a: str
    for a in acteurs:
        nb_part = len(partenaires(C,a))

        if nb_part > nb_max:
            res = a
            nb_max = nb_part

    return res
```

Question 3.6 : [4/66]

La base de données manipulée jusqu'ici facilite la recherche d'information à partir du nom d'un film. On aimerait disposer également d'une base de données permettant de rechercher des informations à partir du nom d'un acteur. Une telle base de données, appelée *base de données de filmographie*, est représentée en Python par le type `dict[str:set[str]]` dans lequel une clé est un nom d'acteur ou d'actrice dont la valeur associée est l'ensemble des films dans lesquels il (ou elle) a joué.

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `filmographie` qui, étant donné une base de données de casting C, renvoie la base de données de filmographie associée.

Par exemple :

```
>>> filmographie(Godard)
{'brasseur': {'bande a part'},
  'hanin': {'a bout de souffle'},
  'karina': {'pierrot le fou', 'made in USA', 'bande a part',
             'une femme est une femme'},
  'brialy': {'une femme est une femme'},
  'seberg': {'a bout de souffle'},
  'frey': {'bande a part'},
  'belmondo': {'pierrot le fou', 'a bout de souffle', 'une femme est une femme'},
  'leaud': {'made in USA'}}
```

Réponse

```
def filmographie(C):
    """ Casting -> dict[set:set[str]]"""

    # DR : dict[str : set[str]}
    DR = dict()

    # acteurs : set[str]
    acteurs = tous_acteurs(C)

    # act : str
    for act in acteurs:
        DR[act] = films_avec(C,act)

    return DR
```

Variante avec compréhension :

```
def filmographie_comp(C):
    """ Casting -> dict[set:set[str]]"""

    return { act:films_avec(C, act) for act in tous_acteurs(C) }
```

Catégories de films

On suppose à partir de maintenant que l'on dispose d'une seconde base de données qui décrit non pas le casting des films mais la catégorie à laquelle ils appartiennent. Par exemple :

Catégorie	Films
Policier	A bout de souffle, Pierrot le fou, Made in USA
Drame	A bout de souffle, Pierrot le fou, Bande a part
Romance	A bout de souffle
Comédie	Pierrot le fou, Une femme est une femme

Cette base de données de catégories est obtenue en Python avec la définition suivante :

```
>>> Godard_CAT = {
  'policier': {'a bout de souffle', 'pierrot le fou', 'made in USA'},
  'drame': {'a bout de souffle', 'pierrot le fou', 'bande a part'},
  'romance': {'a bout de souffle'},
  'comedie': {'pierrot le fou', 'une femme est une femme'}}
```

Question 3.7 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `acteur_par_cat` qui, étant donnés une base de données de casting C, une base de données de catégorie CAT et un nom de catégorie x, renvoie l'ensemble des acteurs ayant joué dans au moins un film de catégorie x.

Par exemple :

```
>>> acteur_par_cat(Godard, Godard_CAT, 'romance')
{'seberg', 'belmondo', 'hanin'}
>>> acteur_par_cat(Godard, Godard_CAT, 'policier')
```

```
{'karina', 'belmondo', 'seberg', 'hanin', 'leaud'}
```

Réponse

```
def acteur_par_cat(C, CAT, x):
    """Casting * dict[str:set[str]] * str -> set[str]"""

    #SR : set[str]
    SR = set()

    #films : str
    for films in CAT[x]:
        SR = SR | C[films]

    return SR
```

Variante avec compréhension :

```
def acteur_par_cat_comp(C, CAT, x):
    """Casting * dict[str:set[str]] * str -> set[str]"""

    return {a for films in CAT[x] for a in C[films]}
```

Question 3.8 : [5/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `freq_cat` qui, étant donnés une base de données de casting C, une base de données de catégorie CAT et un nom d'acteur a, renvoie un dictionnaire de fréquence des catégories pour cet acteur, c'est-à-dire un dictionnaire dont les clefs sont les catégories et la valeur associée, le nombre de films de cette catégorie dans lesquels a a joué.

Par exemple :

```
>>> freq_cat(Godard, Godard_CAT, 'leaud')
{'romance': 0, 'policier': 1, 'drame': 0, 'comedie': 0}
>>> freq_cat(Godard, Godard_CAT, 'hanin')
{'romance': 1, 'policier': 1, 'drame': 1, 'comedie': 0}
>>> freq_cat(Godard, Godard_CAT, 'belmondo')
{'romance': 1, 'policier': 2, 'drame': 2, 'comedie': 2}
```

Réponse

```
def freq_cat(C, CAT, a):
    """Casting * dict[str:set[str]] * str -> dict[str:int]"""

    #DR:dict[str:int]
    DR = {}

    #F: dict[str:set[str]]
    F = filmographie(C)

    #x:str
    for x in CAT:
        DR[x] = len(CAT[x] & F[a])

    return DR
```

Question 3.9 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `palette_acteur` qui, étant donné une base de données de casting C, une base de données de catégorie CAT et un nom d'acteur a, renvoie l'ensemble des catégories contenant au moins un film dans lequel a a joué.

Par exemple :

```
>>> palette_acteur(Godard, Godard_CAT, 'leaud')
{'policier'}
>>> palette_acteur(Godard, Godard_CAT, 'hanin')
{'romance', 'policier', 'drame'}
>>> palette_acteur(Godard, Godard_CAT, 'belmondo')
{'romance', 'policier', 'drame', 'comedie'}
```

Réponse

```
def palette_acteur(C, CAT, a):
    """Casting * dict[str:set[str]] * str -> set[str]"""

    #SR:set[str]
    SR = set()

    #freq: dict[str:int]
    freq = freq_cat(C,CAT,a)

    #x:str
    for x in freq:
        if freq[x] >=1:
            SR.add(x)

    return SR
```

Variante avec compréhension :

```
def palette_acteur_comp(C, CAT, a):
    """Casting * dict[str:set[str]] * str -> set[str]"""

    #freq: dict[str:int]
    freq = freq_cat(C,CAT,a)


    return {x for x in freq if freq[x]>=1}
```

Exercice 4 : Intervalles compatibles

Dans cet exercice, un intervalle ouvert est représenté par un couple d'entiers (a, b) tel que $a < b$ et on appelle `Intervalle` le type des intervalles.

L'*intérieur* d'un intervalle (a, b) est l'ensemble des nombres réels x tels que $a < x < b$ (sa notation mathématique est $]a, b[$).

On dit que deux intervalles sont *compatibles* si leurs intérieurs sont disjoints.

Par exemple :

- les intervalles $(3, 5)$ et $(5, 12)$ sont compatibles car leurs intérieurs $]3, 5[$ et $]5, 12[$ sont disjoints ;
- les intervalles $(3, 5)$ et $(0, 2)$ sont compatibles car leurs intérieurs $]3, 5[$ et $]0, 2[$ sont disjoints ;
- les intervalles $(3, 5)$ et $(4, 12)$ ne sont pas compatibles car leurs intérieurs $]3, 5[$ et $]4, 12[$ ne sont pas disjoints : par exemple, le réel 4.5 appartient à $]3, 5[$ et à $]4, 12[$.

Question 4.1 : [2/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `compatibles` qui, étant donnés deux intervalles `I1` et `I2`, renvoie la valeur `True` si les intervalles `I1` et `I2` sont compatibles, et `False` sinon.

Par exemple :

```
>>> compatibles((3, 5), (0, 2))
```

```
True
>>> compatibles((3, 5), (8, 12))
True
>>> compatibles((3, 5), (4, 12))
False
>>> compatibles((3, 5), (5, 12))
True
>>> compatibles((3, 5), (0, 12))
False
```

Réponse

```
def compatibles(I1, I2):
    """ Intervalle * Intervalle -> bool
    Retourne True si  I1 et I2 sont compatibles, False sinon.
    """

    d1, f1 = I1
    d2, f2 = I2
    return (f2 <= d1) or (f1 <= d2)
```

Question 4.2 : [3/66]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `compatible_avec` qui, étant donnés un intervalle `I` et une liste d'intervalles `L`, renvoie la valeur `True` si l'intervalle `I` est compatible avec tous les intervalles de `L`, et False sinon.

Par exemple :

```
>>> compatible_avec((3, 5), [(0, 2), (8, 12), (5,12)])
True
>>> compatible_avec((3, 5), [(0, 2), (4, 12), (5,12)])
False
>>> compatible_avec((3, 5), [])
True
```

Réponse

```
def compatible_avec(I, L):
    """ Intervalle * Liste[Intervalle] -> bool
    Retourne True si I est compatible avec tous les intervalles de L,
    False sinon.
    """

    # J : Intervalle
    for J in L:
        if not(compatibles(I, J)):
            return False
    return True
```

Question 4.3 : [2/66]

Compléter dans les cadres prévus à cet effet les informations de typage de la fonction mystère ci-dessous.

Attention : il n'est demandé ici que de donner la signature de la fonction et les déclarations des types des variables.

```
def myst(L):
    """
        
    ... """

    # x :

    x = []

    # y :

    for y in L:
        if compatible_avec(y, x):
            x.append(y)

    return x
```

Réponse

```
def myst(L):
    """
      Liste[Intervalle] -> Liste[Intervalle]
    """

    # x : Liste[Intervalle]

    x = []

    # y : Intervalle

    for y in L:
        if compatible_avec(y, x):
            x.append(y)

    return x
```

Question 4.4 : [2/66]

Compléter la table de simulation ci-dessous pour l'application :

`myst([(0, 2), (8, 12), (5,12), (3, 5), (4, 6)])`

Remarque : la table comporte plus de lignes que nécessaire.

Tour	Variable `y`	Variable `x`
entrée		
1er		

Réponse

Simulation pour : `L=[(0, 2), (8, 12), (5,12), (3, 5), (4, 6)]`

Tour	Variable `y`	Variable `x`
entrée	–	`[]`
1er	`(0, 2)`	`[(0, 2)]`
2ème	`(8, 12)`	`[(0, 2), (8, 12)]`
3ème	`(5, 12)`	`[(0, 2), (8, 12)]`
4ème	`(3, 5)`	`[(0, 2), (8, 12), (3, 5)]`
5ème	`(4, 6)`	`[(0, 2), (8, 12), (3, 5)]`
sortie	–	`[(0, 2), (8, 12), (3, 5)]`

Question 4.5 : [2/66]

On considère la liste `L0` définie par la compréhension :

```
>>> L0 = [(i, i + 5) for i in range(0,1000)]
```

Exprimer le résultat rendu par `myst(L0)` sous la forme d'une expression avec compréhension.

Réponse

On peut définir `myst(L0))` par la compréhension :

```
[(i, i + 5) for i in range(0,1000,5)]
```

ou bien :

```
[(i, i + 5) for i in range(0,1000) if i % 5 == 0]
```

ou encore :

```
[(5 * i, 5 * (i + 1)) for i in range(0, 200)]
```