CONTRÔLE 1 — POO et récursivité

Exercice 1. **1.** Écrire une fonction python `etoiles` qui étant donné un entier `n` renvoie la chaine de caractères constituée du caractère `*`, répété `n` fois. Toute syntaxe python est acceptée.

Code python

```
def etoiles(n):
    """ int -> str
    Renvoie la chaine de caractère constituée de n caractères * """
    pass
```

Code python

```
print(etoiles(2))
print(etoiles(5))
```

Résultat

```
**
*****
```

Rappel. Si `s1` et `s2` sont deux chaînes de caractères, alors `s1 + s2` renvoie la concaténation de `s1` avec `s2`. Ainsi, `"bonjour" + " ! "` renvoie la chaîne `"bonjour ! "`.

2. On donne le code des fonctions `f`, `g`, `h` et `k` suivantes.

Code python

```
def f(n):
    """ int -> None """
    if n == 1:
        print(etoiles(1))
    else:
        print(etoiles(n))
        f(n - 1)

def g(n):
    """ int -> None """
    if n == 1:
        print(etoiles(1))
    else:
        g(n - 1)
        print(etoiles(n))
```

Code python

```
def h(n):
    """ int -> None """
    if n == 1:
        print(etoiles(1))
    else:
        h(n - 1)
        print(etoiles(n))
        h(n - 1)

def k(n):
    """ int -> None """
    if n == 1:
        print(etoiles(1))
    else:
        print(etoiles(n))
        k(n - 1)
        print(etoiles(n))
```

On donne également, **dans le désordre** les affichages réalisés par les instructions `f(3)`, `g(3)`, `h(3)`, et `k(3)`.

Résultat
```
*
**
*
***
*
**
*
```

Résultat
```
*
**
***
```

Résultat
```
***
**
*
```

Résultat
```
***
**
*
**
***
```

Sans justifier votre réponse, déterminer quelle instruction a provoqué quel affichage.

3. On se propose d'étudier l'affichage réalisé par la fonction `u` suivante.

Code python
```python
def u(n, i):
    if n == i:
        print(etoiles(n))
    else:
        print(etoiles(i))
        u(n, i + 1)
        print(etoiles(n))
        u(n, i + 1)
        print(etoiles(i))
```

a. Dresser l'arbre d'appel de l'instruction `u(3, 1)`. On représentera les appels successifs les uns en dessous des autres, et on indiquera au bon endroit le long de l'arbre les différents événements d'affichage à l'aide de la ligne de code correspondante.

b. En déduire l'affichage réalisé par `u(3, 1)`.

c. Quelle erreur soulève l'instruction `u(1, 3)` ? Expliquer votre réponse.

Exercice 2. Un appartement est constitué (en général) de plusieurs pièces. Dans ces pièces se trouvent un ou plusieurs meubles, qui occupent une surface au sol, diminuant ainsi la surface disponible dans la pièce. L'objectif de cet exercice est de modéliser cette situation à l'aide de classes en python. Les questions de cet exercice forment un ensemble, mais il n'est pas nécessaire d'avoir répondu à une question pour aborder la suivante. En particulier, on pourra utiliser les méthodes des questions précédentes même quand elles n'ont pas été codées.

1. On donne le code de la classe `Meuble`. L'attribut `nom` indique de quel meuble il s'agit (une armoire, un lit, etc.) ; l'attribut `surface_sol` represente la surface qu'occupe le meuble au sol, en mètre carrés.

Code python

```python
class Meuble:
    def __init__(self, n, s):
        """ Meuble, str, float -> None """
        self.nom = n
        self.surface_sol = s
```

Écrire l'instruction python permettant d'instancier une variable `lit` de type `Meuble`, qui représente un lit nommé "Lit Queen Size" (longueur : 2m, largeur 1,6m).

2. On représente une pièce d'un appartement à l'aide de la classe `Piece`. Ses attributs sont les suivants :

- `nom` de type `str` ("chambre 1", "cuisine", par exemple).
  L'attribut est définit par l'utilisateur lors de l'instanciation de l'objet.
- `superficie` de type `float`. Il s'agit de la taille de la pièce en mètre carrés.
  L'attribut est définit par l'utilisateur lors de l'instanciation de l'objet.
- `liste_meubles`, une liste d'objets de type `Meuble`.
  Initialement vide, contient la liste des meubles présents dans la pièce.
- `superficie_occupee`, de type `float`.
  Vaut initialement 0 (la pièce est vide). Doit augmenter lorsque des meubles sont ajoutés à la pièce.

a. Écrire le code python qui permet de définir une classe `Piece` possédant les attributs décrits dans l'énoncé. On écrira de plus la signature de la fonction `__init__`.

b. Écrire une méthode `possede_place` de la classe `Piece` qui renvoie `True` si et seulement si la pièce `self` possède au moins `surface` mètre carrés disponibles.

Code python

```python
def possede_place(self, surface):
    """ Piece, float -> bool
    Détermine si la pièce possède surface m2 disponibles """
    # À compléter
```

c. Écrire une méthode `ajoute` de la classe `Piece`, qui ajoute le `meuble` à la pièce `self`, en modifiant l'attribut `liste_meubles`. Cette méthode mettra également à jour l'attribut `superficie_occupee` en l'incrémentant de la surface au sol occupée par le meuble. Il est cependant possible qu'il n'y ait pas suffisamment d'espace dans la pièce pour accueillir le meuble. Dans ce cas l'attribut `liste_meuble` ne sera pas modifié. La méthode renverra `True` si le meuble a effectivement été ajouté à la pièce, `False` sinon.

Code python

```python
def ajoute(self, meuble):
    """ Pièce, Meuble -> bool """
    # À compléter
```

3. Un appartement contient une ou plusieurs pièces. On donne ci-dessous la définition de la classe `Appartement` :

Code python

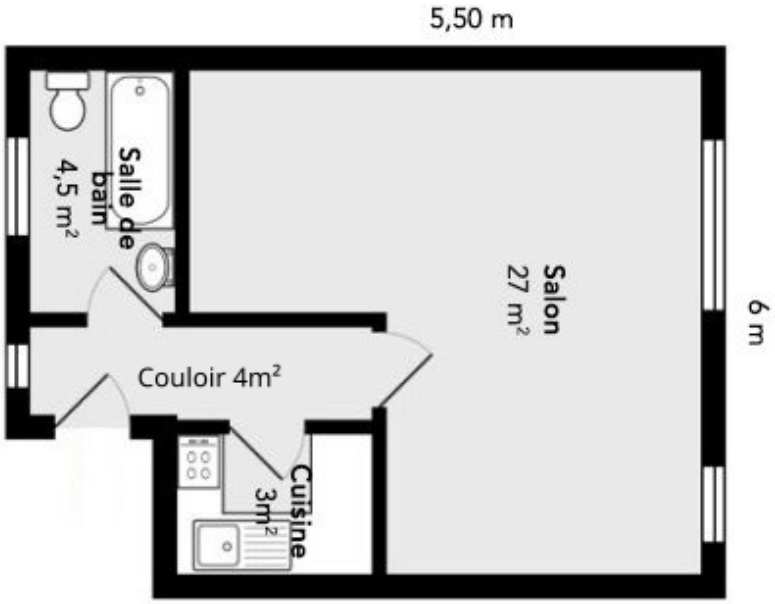
```python
class Appartement:
    def __init__(self, c):
        """ Appartement, [Piece] -> None """
        self.contenu = c

```

a. On suppose que les variables de type `Meuble` suivants ont été correctement définies.

Variable	`lit`	`bain`	`wc`	`lav`	`gaz`	`ev`
Nom	Lit Queen Size	Baignoire	Toilettes	Lavabo (toilettes)	Gazinière	Évier (cuisine)
Surface au sol	3.2	1.2	0.4	0.1	0.25	0.15

Par exemple, la variable `lit` est de type `Meuble`, `lit.nom` est `"Lit Queen Size"` et `lit.surface_sol` vaut `3.2`. À l'aide des variables décrites dans le tableau de l'énoncé, écrire le code python qui permet d'instancier une variable de type `Appartement` qui représente l'appartement dont on donne le plan ci-dessous.



b. Écrire une méthode `prix` de la classe `Appartement`, qui étant donné un prix de vente au mètre carré, renvoie le prix de vente total de l'appartement. Celui-ci sera obtenu en calculant la superficie totale (disponible et occupée) de l'ensemble des pièces de l'appartement, que l'on multipliera par le prix au mètre carré passé en paramètre.

Code python
```python
def prix(self, prix_m2):
    """ Appartement, float -> float
    Calcule le prix de vente de l'appartement. """
    # À compléter
```

c. Écrire une méthode `recherche_place` de la classe `Appartement`, qui étant donné une `superficie` (donnée en mètre carrés) renvoie la liste des pièces où on peut trouver `superficie` mètre carrés disponibles.

Code python
```python
def recherche_place(self, superficie):
    """ Appartement, float -> [Piece]
    Renvoie la listes des pièces de l'appartement avec plus de
    ↪ superficie mètre carrés disponibles. """
    # À compléter
```

4. Écrire une **fonction** `recherche_appart` qui étant donné une liste `collection` d'appartements meublés dont le prix de vente au mètre carré est de 10 000€, renvoie un appartement dont le prix de vente est le minimum.

Code python
```python
def recherche_appart(collection):
    """ [Appartement] -> Appartement """
    # À compléter
```