Tp- Algorithmes Gloutons – Distributeur snack

L'objectif de ce Tp est de découvrir une situation dans laquelle un algorithme de type Glouton est utilisé. Il est aussi de coder en utilisant le paradigme de programmation orienté objet (Poo).

Le distributeur « *snack* » ci-contre accepte les pièces de 5cts, 10cts, 20cts, 50cts, 1€, 2€. Lors d'un paiement, il rend la monnaie.
Les pièces introduites et celles rendues sont recueillies dans des cassettes situées entre la fente d'introduction et la trappe de rendu.

Initialement chacune des cassettes est remplie de 30 pièces. Leur capacité maximale est de 100 pièces.

On se propose dans la suite de créer 2 classes pythons. Une première nommée « *Cassette* », pour modéliser chacune des cassettes contenant les pièces. Une cassette ne contient qu'un même type de pièces.
Une seconde nommée « *Distributeur* », pour modéliser le système de gestion de paiement contenu dans le distributeur. Ce système est composé de 6 cassettes de pièces et des méthodes qui gèrent le paiement d'un article.

1- OBJETS CASSETTES :

Chaque cassette est modélisée par un objet de la classe « *Cassette* ».

a. ATTRIBUTS ET CREATION DE LA METHODE CONSTRUCTEUR :

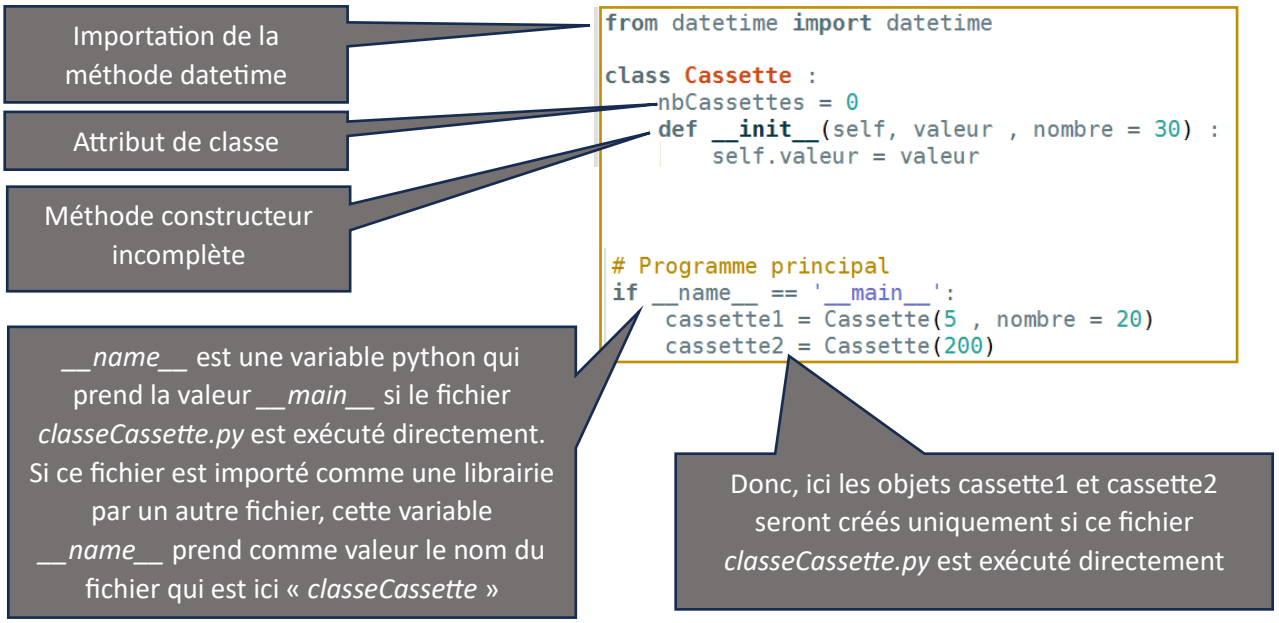
On se propose de créer cette classe nommée *Cassette*. Elle possèdera 1 attribut de classe nommé nbCassettes qui donne le nombre d'instances créées (*int*).

Les attributs d'instances des objets créés seront :

- valeur : donne la valeur en centimes des pièces contenues dans cette cassette (*int*)
- nombre : donne le nombre de pièces contenues dans cette cassette (*int*). La valeur par défaut de cet attribut est égale à 30 car au chargement d'une nouvelle cassette, celle-ci contient 30 pièces.
- nom : donne le nom courant de la pièce (*str*)
- historique : liste dont les éléments sont des tuples du type (date , nombre pièces ajoutées ou retirées) . En python ces tuples seront `(datetime.now(), +nombre)`

⇨ Ouvrir un nouveau fichier et l'enregistrer sous le nom *classeCassette.py*

⇨ On donne ci-dessous, les premières et dernières lignes à écrire dans ce fichier :



⇨ Compléter le script de la méthode constructeur *__init__()* afin de pouvoir retrouver les résultats donnés ci-dessous :

Par exemple, avec les 2 instances suivantes créées dans le programme principal :

```
cassette1 = Cassette(5 , nombre = 20)
cassette2 = Cassette(200)
```

On devra obtenir dans la console :

`>>> cassette1.valeur` `5`	`>>> cassette2.valeur` `200`
`>>> cassette1.nombre` `20`	`>>> cassette2.nombre` `30`
`>>> cassette1.nom` `'5 centimes'`	`>>> cassette2.nom` `'200 centimes'`
`>>> cassette1.historique` `[(datetime.datetime(2024, 10, 28, 21, 48, 59, 924851), 20)]`	`>>> cassette2.historique` `[(datetime.datetime(2024, 10, 28, 21, 48, 59, 924851), 30)]`
`>>> print(cassette1.historique[0][0])` `2024-10-28 21:48:59.924851`	
`>>> Cassette.nbCassettes` `2`	

CORRIGE

```
class Cassette :
    nbCassettes = 0
    def __init__(self, valeur , nombre = 30) :
        self.valeur = valeur
        self.nombre = nombre
        self.nom = str(valeur)+" centimes"
        self.historique = []
        self.historique.append((datetime.now(), +nombre))
        Cassette.nbCassettes +=1
```

b. CREATION DE LA METHODE __STR__() :

⇨ Compléter cette classe *Cassette* en écrivant le code de la méthode *__str__()* .

Vous devrez pouvoir retrouver les résultats donnés ci-contre (exécutions dans la console) :

```
>>> print(cassette1)
Cassette de pièces de 5 centimes. Remplissage 20/100


>>> print(cassette2)
Cassette de pièces de 200 centimes. Remplissage 30/100
```

CORRIGE

```
def __str__(self) :
    return (f"Cassette de pièces de {self.nom}. Remplissage {self.nombre}/100 \n")
```

c. CREATION DE LA METHODE ESTVIDE() :

La méthode *estVide()* permet de savoir si la cassette est vide. Elle renvoie *True* si l'attribut *nombre* est ≤ 0 . Dans le cas contraire elle renvoie *False*.

⇨ Compléter le script de la classe *Cassette* en y écrivant le code de la méthode *estVide ()* . Vous devrez pouvoir retrouver les résultats donnés ci-dessous :

```
>>> print(cassette1)
Cassette de pièces de 5 centimes. Remplissage 20/100


>>> cassette1.estVide()
False

>>> cassette1.nombre = 0

>>> cassette1.estVide()
True
```

CORRIGE

```
def estVide(self) :
    if self.nombre <= 0 : return True
    else : return False
```

d. CREATION DE LA METHODE ESTPLEINE()() :

La méthode *estPleine()* permet de savoir si la cassette est pleine. Elle renvoie *True* si l'attribut *nombre* est ≥ 100 . Dans le cas contraire elle renvoie *False*.

⇨ Compléter le script de la classe *Cassette* en y écrivant le code de la méthode *estVide ()* . Vous devrez pouvoir retrouver les résultats donnés ci-dessous :

```
>>> print(cassette2)
Cassette de pièces de 200 centimes. Remplissage 30/100


>>> cassette2.estPleine()
False

>>> cassette2.nombre = 100

>>> cassette2.estPleine()
True
```

CORRIGE

```
def estPleine(self) :
    if self.nombre >= 100 : return True
    else : return False
```

e. CREATION DE LA METHODE AJOUTER() :

La méthode *ajouter()* permet de rajouter une pièce dans la cassette. Elle renvoie *True* si l'ajout a été réalisé correctement. Si la cassette était déjà pleine (capacité maximale de 100 pièces), l'ajout n'est pas réalisé et cette méthode renvoie *False*.

⇨ Compléter le script de la classe *Cassette* en y écrivant le code de la méthode *ajouter()* . Utiliser la méthode *estPleine()* mise au point précédemment dans le test qui repère si la cassette est pleine. Vous devrez pouvoir retrouver les résultats donnés ci-dessous :

```
>>> print(cassette1)
Cassette de pièces de 5 centimes. Remplissage 20/100


>>> cassette1.historique
[(datetime.datetime(2024, 10, 28, 22, 2, 52, 908626), 20)]

>>> cassette1.ajouter()
True

>>> cassette1.historique
[(datetime.datetime(2024, 10, 28, 22, 2, 52, 908626), 20),
(datetime.datetime(2024, 10, 28, 22, 4, 42, 486479), 1)]

>>> print(cassette1)
Cassette de pièces de 5 centimes. Remplissage 21/100


>>> cassette1.nombre = 100

>>> cassette1.ajouter()
False

>>> print(cassette1)
Cassette de pièces de 5 centimes. Remplissage 100/100
```

CORRIGE

```
def ajouter(self) :
    if self.estPleine() : return False
    self.nombre = self.nombre + 1
    self.historique.append((datetime.now(), 1))
    return True
```

f. CREATION DE LA METHODE RETIRER() :

La méthode *retirer()* permet de retirer une pièce dans la cassette. Elle renvoie *True* si le retrait a été réalisé correctement. Si la cassette était vide aucun retrait n'est réalisé et cette méthode renvoie *False*.

⇨ Compléter le script de la classe *Cassette* en y écrivant le code de la méthode *retirer()* . Utiliser la méthode *estVide()* mise au point précédemment dans le test qui repère si la cassette est vide Vous devrez pouvoir retrouver les résultats donnés ci-contre :

```
>>> print(cassette2)
Cassette de pièces de 200 centimes. Remplissage 30/100


>>> cassette2.historique
[(datetime.datetime(2024, 10, 28, 22, 2, 52, 908626), 30)]

>>> cassette2.retirer()
True

>>> print(cassette2)
Cassette de pièces de 200 centimes. Remplissage 29/100


>>> cassette2.historique
[(datetime.datetime(2024, 10, 28, 22, 2, 52, 908626), 30),
(datetime.datetime(2024, 10, 28, 22, 10, 42, 900285), -1)]

>>> cassette2.nombre = 0

>>> cassette2.retirer()
False

>>> print(cassette2)
Cassette de pièces de 200 centimes. Remplissage 0/100
```

CORRIGE

```
def retirer(self) :
    if self.estVide() : return False
    self.nombre = self.nombre -1
    self.historique.append((datetime.now(), -1))
    return True
```

g. CREATION DE LA METHODE AFFICHERHISTORIQUE() :

La méthode *afficherHistorique()* permet d'afficher (utilisation de la fonction *print()*) proprement la liste contenu dans l'attribut *historique*.

⇨ Compléter le script de la classe *Cassette* en écrivant le code de la méthode *afficherHistorique ()* afin de pouvoir retrouver les résultats donnés ci-dessous :

```
>>> cassette1.afficherHistorique()
Date : 2024-10-28 22:02:52.908626 , 20
Date : 2024-10-28 22:04:42.486479 , 1
```

```
>>> cassette2.afficherHistorique()
Date : 2024-10-28 22:02:52.908626 , 30
Date : 2024-10-28 22:10:42.900285 , -1
```

CORRIGE

```
def afficherHistorique(self) :
    for elt in self.historique :
        print(f"Date : {elt[0]} , {elt[1]}")
```

h. CONCLUSION POUR LA CREATION DE CETTE CLASSE NOMMEE CASSETTE :

On dispose à présent de cette classe nommée *Cassette* et écrite dans le fichier *classeCassette.py* . Elle permet d'encapsuler dans une même entité, toutes les informations et méthodes utiles à l'utilisation des cassettes.
On se propose à présent de refaire un peu la même chose, en créant une classe *Distributeur*, afin de pouvoir gérer le système de paiement du distributeur de « snack ».

****

2- OBJET DISTRIBUTEUR :

Chaque distributeur est modélisé par un objet de la classe « *Distributeur* ». Un distributeur contient 6 cassettes, contenant elles-mêmes respectivement 30 pièces de 5 cts, 10 cts, 20 cts, 50 cts, 100 cts (1€) et 200 cts (2€).

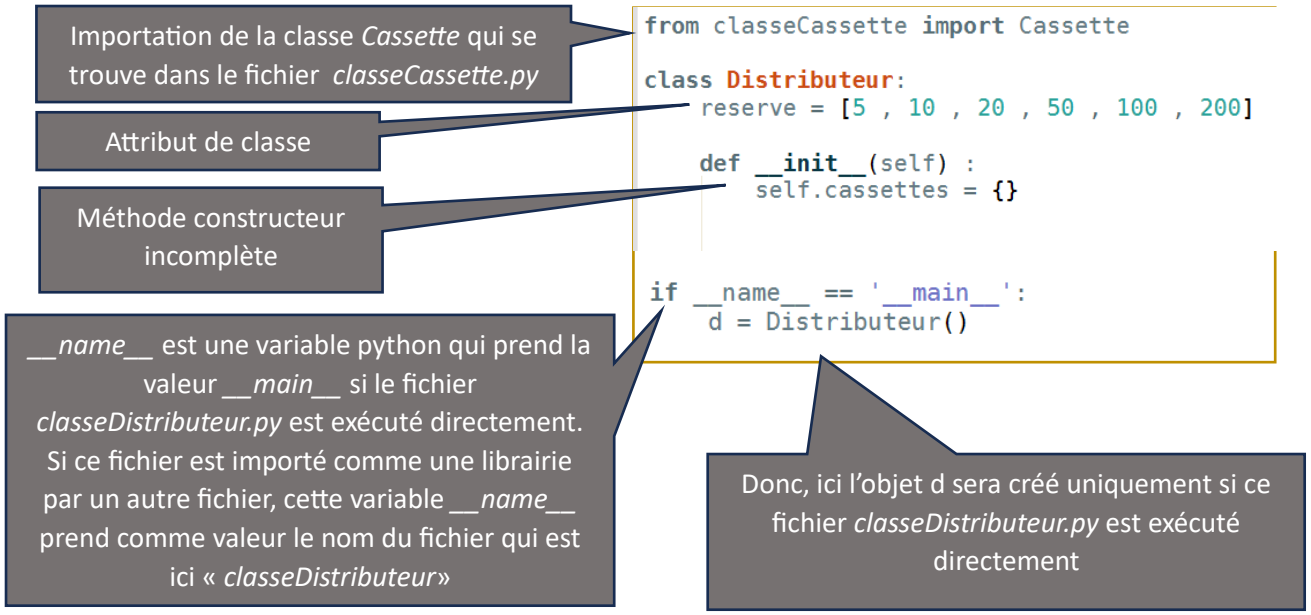
a. ATTRIBUTS ET CREATION DE LA METHODE CONSTRUCTEUR :

On se propose de créer une classe nommée *Distributeur*. Cette classe aura 1 attribut de classe nommé *reserve* dont la valeur est égale à la liste [5 , 10 , 20 , 50 , 100 , 200] qui correspond aux valeurs des pièces contenues dans chacune des 6 cassettes.

Cette classe aura 1 attribut d'instance nommé *cassettes* . Celui-ci contient un dictionnaire de 6 clés égales respectivement à 5, 10, 20, 50, 100 et 200. Les valeurs correspondant à ces clés seront égales à des objets de la classe *Cassette*.

⇨ Ouvrir un nouveau fichier et l'enregistrer sous le nom *classeDistributeur.py*

⇨ On donne ci-dessous, les premières et dernières lignes à écrire dans ce fichier :



⇨ Compléter le script de la méthode constructeur *__init__()* afin de pouvoir retrouver les résultats donnés ci-dessous :

On devra obtenir dans la console, la valeur de l'attribut de classe nommé *reserve* :

```
>>> Distributeur.reserve
[5, 10, 20, 50, 100, 200]
```

Par exemple, avec l'instance suivantes créée dans le programme principal :

```
>>> d = Distributeur()
```

On devra pouvoir obtenir dans la console, la valeur de l'attribut d'instance nommé *cassettes* :

```
>>> d.cassettes
{5: <__main__.Cassette object at 0x00000261AF99BB50>,
10: <__main__.Cassette object at 0x00000261AF99BF70>,
20: <__main__.Cassette object at 0x00000261AF99BBB0>,
50: <__main__.Cassette object at 0x00000261AF99B9D0>,
100: <__main__.Cassette object at 0x00000261AF99B970>,
200: <__main__.Cassette object at 0x00000261AF99B910>}
```

ainsi que des renseignements sur chacune des cassettes du distributeur :

```
>>> type(d.cassettes[5])
<class 'classeCassette.Cassette'>
```

```
>>> print(d.cassettes[20])
Cassette de pièces de 20 centimes. Remplissage 30/100
```

On pourra retrouver le nombre de pièces dans la cassette contenant les pièces de 1 € :

```
>>> d.cassettes[100].nombre
30
```

…. ajouter une pièce dans cette cassette …. ,

puis afficher son historique :

```
>>> d.cassettes[100].ajouter()
True

>>> d.cassettes[100].afficherHistorique()
Date : 2024-10-28 22:02:52.908626 , 30
Date : 2024-10-28 22:43:13.879933 , 1
```

… vérifier qu'une cassette n'est pas vide ou pleine :

```
>>> d.cassettes[10].estVide()
False

>>> d.cassettes[10].estPleine()
False

>>> not d.cassettes[10].estPleine()
True
```

On voit que ce distributeur est notamment composé de 6 cassettes de pièces modélisées ici par des objets de la classe *Cassette*. On va ainsi pouvoir utiliser dans cette nouvelle classe *Distributeur*, les méthodes encapsulées dans la classe *Cassette* qui a été mises au point dans le paragraphe précédent.

```
class Distributeur:
    reserve = [5 , 10 , 20 , 50 , 100 , 200]

    def __init__(self) :                    CORRIGE
        self.cassettes = {}
        for valeur in Distributeur.reserve :
            obj = Cassette(valeur)
            self.cassettes[valeur] = obj
```

b. CREATION DE LA METHODE __STR__() :

⇨ Compléter le script de la classe *Distributeur* en écrivant celui de la méthode *__str__()* .

Une fois cette méthode écrite, on pourra vérifier son bon fonctionnement, en exécutant la ligne suivante dans la console :

```
>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 30/100
Cassette de pièces de 20 centimes. Remplissage 30/100
Cassette de pièces de 50 centimes. Remplissage 30/100
Cassette de pièces de 100 centimes. Remplissage 30/100
Cassette de pièces de 200 centimes. Remplissage 30/100
```

Aide : on pourra utiliser la méthode __str__() de la classe *Cassette* qui retourne la bonne phrase pour chacune des cassettes. On peut par exemple exécuter self.cassettes[10].__str__() pour avoir en retour la phrase :

```
Cassette de pièces de 10 centimes. Remplissage 30 / 100
```

```
def __str__(self) :                                    CORRIGE
    a = "Etat de la réserve : \n"
    for valeur in Distributeur.reserve :
        a = a + self.cassettes[valeur].__str__()
    return a
```

C. CREATION DE LA METHODE PAYER() :

La méthode *payer()* prend en paramètre un nombre entier qui représente le prix de l'achat en centimes. Elle demande au client d'introduire des pièces dans le distributeur. Celle-ci seront ajoutées dans les cassettes. Si une cassette est pleine, la machine accepte tout de même le paiement et dirige la pièce introduite dans un bac auxiliaire. La méthode renvoie un entier qui correspond à la somme en centimes que le distributeur doit rendre au client.

⇨ Compléter le script de la classe *Distributeur* en écrivant le code de la méthode *payer()* afin de pouvoir retrouver les résultats donnés ci-dessous. Pour ajouter une pièce dans une cassette, on utilisera la méthode *ajouter()* de la classe *Cassette* (En Poo, on évite de manipuler directement les attributs d'un objet d'une autre classe)

```
>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 30/100
Cassette de pièces de 20 centimes. Remplissage 30/100
Cassette de pièces de 50 centimes. Remplissage 30/100
Cassette de pièces de 100 centimes. Remplissage 30/100
Cassette de pièces de 200 centimes. Remplissage 30/100


>>> d.payer(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 100
Reste 70 cts - Introduit un pièce : 30
Mauvaise pièce
Reste 70 cts - Introduit un pièce : 50
Reste 20 cts - Introduit un pièce : 50
30
```

Valeurs saisies par le client en exécutant la fonction input()

Détection mauvaise pièce

Somme à rendre

```
>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 30/100
Cassette de pièces de 20 centimes. Remplissage 30/100
Cassette de pièces de 50 centimes. Remplissage 32/100
Cassette de pièces de 100 centimes. Remplissage 31/100
Cassette de pièces de 200 centimes. Remplissage 30/100
```

Quantités modifiées

CORRIGE

```
def payer(self,prix) :
    print(f"\nIntroduire en pièces : {prix} cts")
    s = 0
    while s < prix :
        p = input(f"Reste {prix-s} cts - Introduit un pièce : ")
        p = int(p)
        if p in Distributeur.reserve :
            s = s + p
            retour = self.cassettes[p].ajouter()
        else : print("Mauvaise pièce")
    return s-prix
```

d. CREATION DE LA METHODE GLOUTON() :

La méthode *glouton()* prend en paramètre la liste *Distributeur.reserve* et nombre entier qui représente la somme en centimes, à rendre au client. Elle renvoie une liste contenant les valeurs en centimes des pièces qui composent le rendu. On s'inspirera du code vu en cours et donné ci-contre. L'attribut *nombre* des cassettes dans lesquelles on prélève des pièces pour le rendu doit être mis à jour. Si une cassette est vide, il ne sera pas possible d'y prélever des pièces. Il s'agira de tenir compte de cette situation dans le code.

```
def glouton(l,aRendre):
    rendu = []
    s = 0
    i = len(l)-1
    while i >= 0 and s < aRendre :
        if (s + l[i]) <= aRendre :
            rendu.append(l[i])
            s = s + l[i]
        else :
            i = i - 1
    return rendu
```

⇨ Compléter le script de la classe *Distributeur* en écrivant le code de la méthode *glouton()* afin de pouvoir retrouver les résultats donnés ci-dessous. Pour prélever une pièce dans une cassette, on utilisera la méthode *retirer()* de la classe *Cassette* (En Poo, on évite de manipuler directement les attributs d'un objet d'une autre classe).

```
>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 30/100
Cassette de pièces de 20 centimes. Remplissage 30/100
Cassette de pièces de 50 centimes. Remplissage 32/100
Cassette de pièces de 100 centimes. Remplissage 31/100
Cassette de pièces de 200 centimes. Remplissage 30/100


>>> d.glouton([5 , 10 , 20 , 50 , 100 , 200] , 30)
[20, 10]

>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 29/100
Cassette de pièces de 20 centimes. Remplissage 29/100
Cassette de pièces de 50 centimes. Remplissage 32/100
Cassette de pièces de 100 centimes. Remplissage 31/100
Cassette de pièces de 200 centimes. Remplissage 30/100
```

Rendu d'une pièce de 20 cts et d'une de 10 cts

Quantitées modifiées

Attention, si une des cassettes est vide, elle ne pourra pas délivrer de pièces. Pour vérifier qu'une cassette n'est pas vide, on utilisera la méthode *estVide()* de la classe *Cassette* (En Poo, on évite de manipuler directement les attributs d'un objet d'une autre classe)

```
>>> d.cassettes[10].nombre = 0

>>> d.glouton([5 , 10 , 20 , 50 , 100 , 200] , 30)
[20, 5, 5]

>>> d.cassettes[5].nombre = 0

>>> d.glouton([5 , 10 , 20 , 50 , 100 , 200] , 30)
[20]
```

CORRIGE

```
def glouton(self,l,aRendre):
    rendu = []
    s = 0
    i = len(l)-1
    while i >= 0 and s < aRendre :
        if (s + l[i]) <= aRendre and not self.cassettes[l[i]].estVide():
            rendu.append(l[i])
            retour = self.cassettes[l[i]].retirer()
            s = s + l[i]
        else :
            i = i - 1
    return rendu
```

e. CREATION DE LA METHODE ACHETER() :

La méthode *acheter()* prend en paramètre un nombre entier qui représente la somme en centimes que le client doit payer. Elle renvoie une liste contenant les valeurs en centimes des pièces qui composent le rendu. Si le rendu ne correspond pas à celui attendu pour cause de cassette vide, la valeur renvoyée sera le string "Pas possible de rendre le rendu exact".

Cette méthode fera appel aux méthodes *payer()* et *glouton()* mises au point précédemment.

⇨ Compléter le script de la classe *Distributeur* en écrivant le code de la méthode *acheter()* afin de pouvoir retrouver les résultats donnés ci-dessous :

```
>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 30/100
Cassette de pièces de 20 centimes. Remplissage 30/100
Cassette de pièces de 50 centimes. Remplissage 30/100
Cassette de pièces de 100 centimes. Remplissage 30/100
Cassette de pièces de 200 centimes. Remplissage 30/100


>>> d.acheter(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 100
Reste 70 cts - Introduit un pièce : 100
[20, 10]

>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 29/100
Cassette de pièces de 20 centimes. Remplissage 29/100
Cassette de pièces de 50 centimes. Remplissage 30/100
Cassette de pièces de 100 centimes. Remplissage 32/100
Cassette de pièces de 200 centimes. Remplissage 30/100
```

Si une des cassettes est vide, on peut obtenir :

```
>>> d.cassettes[5].nombre = 0

>>> d.acheter(5)

Introduire en pièces : 5 cts
Reste 5 cts - Introduit un pièce : 10
'Pas possible de rendre le rendu exact'
```

On donne ci-dessous, une autre série d'exécutions qui aident à comprendre ce qui est attendu :

```
>>> d.acheter(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 200
[20, 10]

>>> print(d)
Etat de la réserve :
Cassette de pièces de 5 centimes. Remplissage 30/100
Cassette de pièces de 10 centimes. Remplissage 29/100
Cassette de pièces de 20 centimes. Remplissage 29/100
Cassette de pièces de 50 centimes. Remplissage 30/100
Cassette de pièces de 100 centimes. Remplissage 30/100
Cassette de pièces de 200 centimes. Remplissage 31/100


>>> d.cassettes[20].nombre = 0

>>> d.acheter(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 200
[10, 10, 10]

>>> d.cassettes[10].nombre = 0

>>> d.acheter(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 200
[5, 5, 5, 5, 5, 5]

>>> d.cassettes[5].nombre = 0

>>> d.acheter(170)

Introduire en pièces : 170 cts
Reste 170 cts - Introduit un pièce : 200
'Pas possible de rendre le rendu exact'
```

CORRIGE

```
def acheter(self,prix) :
    aRendre = self.payer(prix)
    rendu = self.glouton(Distributeur.reserve,aRendre)
    total = 0
    for p in rendu :
        total = total + p
    if total != aRendre : return "Pas possible de rendre le rendu exact"
    else : return  rendu
```

3- CONCLUSION :

On dispose à présent de cette classe nommée *Distributeur* écrite dans le fichier *classeDistributeur.py* . Elle permet d'encapsuler dans une même entité, toutes les informations et méthodes utiles à l'utilisation des distributeurs. On a utilisé dans cette classe des objets issus d'une autre classe nommée *Cassette*. Cette façon de procéder permet d'encapsuler dans des objets, des données qui leur sont propres et des méthodes qui permettent de manipuler ces données.

⇨ **N'oubliez pas d'uploader les fichiers *classeCassette.py* et *classeDistributeur.py***