

Constructeur et mot-clé self

Table des matières

I. Notion de constructeur	3
II. Exercice : Quiz	6
III. Mot clé « self »	7
IV. Exercice : Quiz	9
V. Essentiel	10
VI. Auto-évaluation	10
A. Exercice	10
B. Test	10
Solutions des exercices	12

I. Notion de constructeur

Durée : 1 h

Environnement de travail : PC connecté à Internet

Contexte

Comme vous le savez bien, la notion d'objet est plus qu'essentielle en Python. En Python, on peut dire que « *tout est objet* ». Mais comme tous les objets ne sont pas le fruit du hasard, il faut les créer et c'est là que rentrent en jeu les classes que vous connaissez bien maintenant. Les classes sont comme des recettes de cuisine qui permettent d'élaborer un gâteau. Elles se composent d'un état qui fait référence aux données traitées et d'un comportement qui, lui, correspond aux méthodes (ce que va faire le programme).

Mais vous vous demandez maintenant comment construire un objet. Eh bien, vous allez justement apprendre à utiliser une méthode : le constructeur. Un constructeur se définit dans une classe comme une fonction et permet de créer un objet grâce à la classe à laquelle il appartient. Il nous sera utile pour qu'une méthode puisse toujours afficher un résultat, sans qu'il soit nécessaire d'effectuer une manipulation sur l'objet. La méthode est exécutée automatiquement lorsque l'on instancie un nouvel objet à partir d'une classe.

Nous allons donc voir dans ce cours comment utiliser le constructeur `__init__` puis comment utiliser le mot clef « *self* ». Nous allons faire le point sur les concepts d'instance et d'instanciation, et cela sur deux types d'attributs : les attributs de classe et les attributs d'instance. Après ce cours, vous saurez créer un objet à partir d'une classe.

Les objectifs de cette partie sont les suivants :

- Comprendre la notion de constructeur
- Comprendre l'instanciation avec `__init__`

Définition

• Constructeur

Un constructeur est une méthode qui va permettre la création d'objets à partir d'une classe. Dans ce cours, nous allons apprendre à utiliser la méthode `__init__`.

• Instance

Une instance représente une occurrence d'une classe, c'est-à-dire l'ensemble des attributs et méthodes qui caractérisent un objet créé à partir de cette classe.

• Instanciation

L'instanciation, c'est lorsqu'on va concrètement créer un objet avec une classe. C'est un peu comme si on appliquait une recette de cuisine pour cuisiner un gâteau.

Pour créer un objet, il va nous falloir tout d'abord créer une classe. Admettons que nous voulons créer une classe qui fabrique un vélo :

```
1 class MonVelo:
2     roues = 2
3     freins = "disques"
```

Pour l'instant, nous avons juste une classe, mais qui ne sert pas à grand-chose. Comment créer un objet ?

```
1 class MonVelo:
2     def __init__(self, roues, freins):
3         self.roues = roues
4         self.freins = freins
```

Remarque

Vous vous demandez peut-être à quoi sert le mot « *self* ». Nous verrons ce point en détail dans la seconde partie, mais pour que vous vous y retrouviez, le mot « *self* » représente l'instance de la classe. Pour l'instant, dites-vous que le mot « *self* » veut dire « *soi-même* » : si vous instanciez un objet, il fera donc référence aux attributs de cet objet instancié.

Concrètement, qu'est-ce qui se passe ? Dans la classe « *Monvelo* », on utilise le constructeur car on veut créer un objet qui correspond à la classe « *Monvelo* », c'est-à-dire qui soit en quelque sorte « *coulé dans le moule* » de la classe « *Monvelo* ». Mais là, on ne crée rien, il n'y a aucune instanciation. Comment fabriquer un vélo à 2 roues et avec des freins à disques, par exemple ? Il va nous falloir instancier la classe et affecter notre objet à une variable afin de pouvoir l'utiliser :

```
1 class Monvelo:
2     def __init__(self, roues, freins):
3         self.roues = roues
4         self.freins = freins
5
6 velo1 = Monvelo (2, "disques")
```

On peut voir ici qu'on affecte à la variable « *velo1* » l'objet instancié par la classe « *Monvelo* » avec roues = « 2 » et freins = « "disques" ». La variable est donc une référence vers cet objet. On peut essayer d'afficher la valeur d'un des attributs de l'objet. Par exemple, pour afficher le type de freins de l'objet :

```
1 class Monvelo:
2     def __init__(self, roues, freins):
3         self.roues = roues
4         self.freins = freins
5
6 velo1 = Monvelo (2, "disques")
7
8 print(velo1.freins)
```

Le terminal renvoie :

```
1 disques
```

Je peux aussi décider d'apporter une modification à mon objet « *velo1* ». Par exemple :

```
1 velo1.roues = 1
```

Ici je change l'attribut « *roues* » de mon objet « *velo1* » et il sera désormais égal à 1. Nous venons de créer un monocycle !

Remarque

En réalité la méthode `__init__` ne construira un nouvel objet qu'à partir du moment où il sera instancié.

Exemple

Maintenant, essayons de voir un exemple de programme faisant appel à ce que nous venons d'apprendre et en allant un peu plus loin. Nous allons gérer les dépenses quotidiennes d'un consommateur. Il va donc falloir créer une classe « *Consumer* » qui va contenir :

- Un attribut « *wealth* » qui stocke la richesse du consommateur.
- Une méthode « *earn(y)* » qui augmente la richesse du consommateur de *y*.
- Une méthode « *spend(x)* » qui diminue la richesse de *x* ou renvoie une erreur si les fonds sont insuffisants.

```

1 class Consumer:
2     def __init__(self, w):
3         self.wealth = w
4     def earn(self, y):
5         self.wealth += y
6     def spend(self, x):
7         new_wealth = self.wealth - x
8         if new_wealth < 0:
9             print("Fonds insuffisants")
10        else:
11            self.wealth = new_wealth

```

Donc concrètement, on comprend qu'avec `__init__`, l'objet qu'on crée va correspondre à un consommateur. Les méthodes `earn` et `spend` vont permettre de faire évoluer l'argument `wealth`, donc le solde du consommateur.

Voici un exemple d'utilisation de la classe « *Consumer* ». Nous commençons par créer une instance d'un consommateur nommé `C1`. Après l'avoir créé, il sera doté de 10 €, nous appliquerons la méthode aux dépenses.

```

1 c1 = Consumer(10) # Créer une instance avec 10€
2 c1.spend(5)
3 print(c1.wealth)

```

Après avoir créé l'instance, nous modifions l'objet en lui appliquant la méthode `spend` avec pour paramètre « 5 ». On retire donc 5 au solde de l'objet, puis on affiche l'argument `wealth`.

Le terminal renvoie donc :

```
1 5
```

Maintenant, admettons qu'on veut rajouter 15 au solde et dépenser 100. On a donc finalement le code :

```

1 class Consumer:
2     def __init__(self, w):
3         self.wealth = w
4     def earn(self, y):
5         self.wealth += y
6     def spend(self, x):
7         new_wealth = self.wealth - x
8         if new_wealth < 0:
9             print("Fonds insuffisants")
10        else:
11            self.wealth = new_wealth
12 c1 = Consumer(10) # Créer une instance avec 10€
13 c1.spend(5)
14 c1.earn(15)
15 c1.spend(100)
16 print(c1.wealth)

```

Qu'est-ce que le terminal va renvoyer ? L'objet a initialement un argument `wealth` égal à 10, il passe à 5, puis on lui rajoute 15, donc il passe à 20 et enfin on veut lui retirer 100. Avec la condition inscrite dans la méthode `spend`, l'opération ne sera pas effectuée étant donné que le solde est insuffisant. Le terminal renvoie donc :

```

1 Fonds insuffisants
2 20

```

Nous pouvons aussi créer plusieurs instances. C'est-à-dire plusieurs consommateurs, chacun avec son propre nom et ses propres données.

```

1 class Consumer:
2     def __init__(self, w):
3         self.wealth = w
4     def earn(self, y):
5         self.wealth += y

```

```

6     def spend(self, x):
7         new_wealth = self.wealth - x
8         if new_wealth < 0:
9             print("Fond insuffisants")
10        else:
11            self.wealth = new_wealth
12    c1 = Consumer(10)
13    c2 = Consumer(12)
14    c2.spend(4)
15    print(c1.wealth)
16    print(c2.wealth)

```

Le terminal va alors renvoyer :

```

1 10
2 8

```

Exercice : Quiz

[solution n°1 p.13]

Question 1

Un(e) _____ est utilisé pour créer un objet.

- ☐ Classe
- ☐ Constructeur
- ☐ Fonction

Question 2

Pour créer, un objet appartenant à une classe, on utilise :

- ☐ __inite__
- ☐ __class__
- ☐ __init__

Question 3

Dans le code suivant, qu'affichera la commande print(voiture1.cv) ?

```

1 class Voiture:
2     def __init__(self, marque, cv):
3         self.marque = marque
4         self.cv = cv
5
6 voiture1 = Voiture ("Bugatti", 400)
7 voiture2 = Voiture ("Chevrolet", 350)

```

- ☐ 400
- ☐ 350
- ☐ Error
- ☐ « cv »

Question 4

L'instanciation, c'est :

- ☐ Lorsqu'on crée effectivement un objet appartenant à une classe
- ☐ Lorsqu'on écrit : `__init__`
- ☐ Un morceau de code qui ne peut exister qu'une fois dans un script

Question 5

Quelle sera la sortie de l'extrait de code suivant ?

```
1 class Sales:
2     def __init__(self, id):
3         self.ide = id
4         id = 100
5 val = Sales(123)
6 print (val.ide)
```

- ☐ SyntaxError
- ☐ 100
- ☐ 123

III. Mot clé « self »

Les objectifs de cette partie sont les suivants :

- Comprendre l'intérêt du mot clé `self` en Programmation Orientée Objet (POO)
- Maîtriser davantage le concept d'instanciation de classe

Le mot clé « `self` » est un concept clé de la Programmation Orientée Objet (POO). Pour reprendre notre exemple avec un script qui crée des vélos, on peut maintenant créer un programme plus poussé que tout à l'heure pour fabriquer des vélos (fictifs).

```
1 class Velo:
2     roues = 2
3     def __init__(self, marque, prix, poids):
4         self.marque = marque
5         self.prix = prix
6         self.poids = poids
7 velo = Velo(marque="Peugeot", prix=500, poids=100)
```

Définition **Attribut de classe ≠ Attribut d'instance (ou Variable de classe ≠ Variable d'instance)**

Un attribut de classe est une variable déclarée sous une classe qui est partagée par toutes les instances de la classe. On la définit à l'intérieur de la classe mais en dehors des autres méthodes. C'est le cas de la variable « `roues` » dans notre exemple, car pour toute instanciation, l'objet créé aura un attribut `roues = 2`. Un attribut d'instance, quant à lui, est déclaré sous un objet. C'est le cas des variables « `self.marque` », « `self.prix` » et « `self.poids` », donc les valeurs seront indépendantes pour chaque objet créé. C'est justement le mot clé « `self` » qui permet de créer un attribut d'instance.

Les variables d'instance sont définies à l'intérieur de la méthode `__init__`.

Comme vous le savez bien maintenant, en Python, cette méthode spéciale est exécutée automatiquement à chaque instantiation d'un nouvel objet à partir de la classe. Si vous essayez de créer deux instances de la classe « *Velo* » et que vous essayez d'accéder à ces variables vous aurez :

```
1 class Velo:
2     roues = 2
3     def __init__(self, marque, prix, poids):
4         self.marque = marque
5         self.prix = prix
6         self.poids = poids
7 velo_01 = Velo("btwin", 250, 15)
8 velo_02 = Velo("rockrider", 170, 12)
```

Toutes les instances ont accès aux attributs de classe, c'est pourquoi les deux vélos ont tous les deux 2 roues car la variable `roues` est un attribut de classe. À l'inverse, chaque vélo possède sa propre marque car il y a deux instances différentes et la variable `marque` est un attribut d'instance :

```
1 class Velo:
2     roues = 2
3     def __init__(self, marque, prix, poids):
4         self.marque = marque
5         self.prix = prix
6         self.poids = poids
7 velo_01 = Velo("btwin", 250, 15)
8 velo_02 = Velo("rockrider", 170, 12)
9 print(velo_01.roues) # 2
10 print(velo_02.roues) # 2
11 print(velo_01.marque) # btwin
12 print(velo_02.marque) # rockrider
```

À quoi sert le `self` ? Lorsque nous créons la méthode `__init__`, en plus des paramètres, nous avons ajouté le mot `self`. Il est impératif de l'ajouter dans les paramètres en première position. Nous allons l'utiliser pour définir les variables d'instance. Mais pourquoi ? `Self` est utilisé pour représenter l'instance de la classe, il permet d'accéder aux attributs et aux méthodes de la classe.

Nous allons maintenant créer la méthode « *rouler* » dans la classe « *Velo* » et avec comme paramètre « *self* » :

```
1 class Velo:
2     roues = 2
3     def __init__(self, marque, prix, poids):
4         self.marque = marque
5         self.prix = prix
6         self.poids = poids
7     def rouler(self):
8         print("Wouh, ça roule mieux avec un vélo {} !".format(self.marque))
9
10 velo_01 = Velo("btwin", 250, 15)
11 velo_02 = Velo("rockrider", 170, 12)
12 print(velo_01.roues) # 2
13 print(velo_02.roues) # 2
14 print(velo_01.marque) # btwin
15 print(velo_02.marque) # rockrider
```

Vous pouvez appeler cette méthode « *rouler* » sur toutes les instances créées. En Python, le mot « *self* » est utilisé par convention. Lorsque vous définissez une méthode d'instance, « *self* » doit toujours être placé en première position. On ajoute à la fin du code :

```
1 velo_01.rouler()
2 velo_02.rouler()
```


Nous avons appelé la méthode « *rouler* » sans passer d'argument. Comment la méthode « *rouler* » fait pour savoir qu'elle doit utiliser les attributs de cette instance et pas d'une autre ? C'est le rôle de « *self* ».

```
1 velo_01.rouler()
```

En coulisse, grâce à la bonne utilisation du mot clé « *self* » dans le code, il se passe ceci.

```
1 Velo.rouler(velo_01)
```

Python utilise la classe « *Velo* » pour exécuter la méthode « *rouler* », et passe l'instance en premier argument. Un argument est bien envoyé à la méthode « *rouler* », et Python s'en charge. Il est obligatoire de mettre « *self* » en premier paramètre dans la définition de la méthode « *rouler* » pour pouvoir l'utiliser avec l'instance « *velo_01* ».

Exercice : Quiz

[solution n°2 p.14]

Question 1

Le mot clé « *self* » sert à :

- ☐ Faire appel à une fonction modulo
- ☐ Créer un attribut d'instance
- ☐ Rien

Question 2

Une variable de classe :

- ☐ Sera différente pour chaque objet créé
- ☐ Sera la même pour chaque objet créé

Question 3

La bonne méthode pour instancier la classe « *Dog* » est :

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

- ☐ Dog()
- ☐ Dog.create("Rufus", 3)
- ☐ Dog("Rufus", 3)
- ☐ Dog.__init__("Rufus", 3)

Question 4

Dans le code suivant :

```
1 class Gateau:
2     oeuf = 4
3     def __init__(self, farine):
4         self.farine = farine
```

- ☐ Chaque gâteau aura un nombre différent d'œufs
- ☐ Chaque gâteau aura la même quantité de farine
- ☐ Chaque gâteau aura un nombre identique d'œufs

Question 5

Que va renvoyer le programme suivant ?

```
1 class Voiture:
2     def __init__(marque, catégorie):
3         self.marque = marque
4         self.catégorie = catégorie
5
6 v1 = Voiture ("Peugeot", "Citadine")
7 print (v1.marque)
```

- ☐ Peugeot
- ☐ Catégorie
- ☐ Une erreur

V. Essentiel

Dans le cadre de la Programmation Orientée Objet en Python, la notion de constructeur est plus qu'essentielle. On peut créer ce qu'on appelle des classes, c'est-à-dire des modèles à partir desquels on va pouvoir fabriquer des objets. Les constructeurs vont permettre de fabriquer effectivement un objet. Pour bien comprendre les termes de vocabulaire, c'est un peu comme si la classe était une recette de cuisine d'un gâteau, l'instance l'ensemble des caractéristiques et actions dans la réalisation d'un gâteau précis, et que l'instanciation était la préparation effective de ce gâteau.

La méthode `__init__` permet de construire un objet lors de l'instanciation. Chaque objet va avoir plusieurs attributs, parmi lesquels figurent notamment les attributs d'instance et les attributs de classe. Les attributs d'instance sont propres à chaque objet créé et les attributs de classe sont communs à tous les objets créés à partir de la classe. Grâce à ces deux types d'attributs, on peut créer des caractéristiques qui seront communes à tous les objets et des caractéristiques qui seront propres à chacun.

Lorsqu'on instancie une classe, il faut affecter l'objet créé à une variable pour pouvoir l'utiliser. La variable est alors une référence vers cet objet. On peut donc afficher et modifier directement les attributs de l'objet en utilisant cette variable. Nous pouvons instancier une classe un nombre de fois illimité, le nombre d'instances est donc infini.

Le mot clé « *self* » fait référence à l'instance, c'est-à-dire à l'objet créé. C'est d'ailleurs le mot « *self* » qui permet de définir des variables d'instance. Il doit être inscrit comme premier argument de la méthode - `init`- pour pouvoir être utilisé.

VI. Auto-évaluation

A. Exercice

Vous êtes un marchand de voitures et vous désirez vendre plusieurs voitures ayant toutes le même nombre de portes (3) et appartenant à la même catégorie (citadine). Cependant, ces voitures ont une marque, une puissance (nombre de cv) et un prix différent.

Question 1

[solution n°3 p.15]

Créez une classe permettant de fabriquer autant de voitures que vous voulez, en tenant compte de ces caractéristiques.

Question 2

[solution n°4 p.16]

Instanciez 2 voitures différentes et affichez pour chacune leur marque, leur catégorie et leur prix.

B. Test

Exercice 1 : Quiz

[solution n°5 p.16]

Question 1

Qu'est-ce qu'un constructeur ? :

- ☐ Une méthode qui permet de créer une classe
- ☐ Une méthode qui permet de créer un objet
- ☐ Une méthode permettant de construire une maison

Question 2

Quelle sera la sortie de l'extrait de code suivant ?

```
1 class Test:
2     def __init__(self,a="Hello World"):
3         self.a = a
4     def display(self):
5         print(self.a)
6 obj=Test()
7 obj.display()
```

- ☐ Le programme imprime une erreur car le constructeur ne peut pas avoir d'argument par défaut
- ☐ Rien ne s'affiche
- ☐ « Hello World » s'affiche
- ☐ Erreur de syntaxe

Question 3

Quel type de variable permet de définir le mot clé « self » ?

- ☐ Une variable de classe
- ☐ Toutes les variables
- ☐ Une variable d'instance
- ☐ Une variable péremptoire

Question 4

Quel type de variable est « cv » ?

```
1 class Voiture:
2     cv=90
3     def __init__(self, marque, catégorie, portes, prix):
4         self.marque = marque
5         self.catégorie = catégorie
6         self.portes = portes
7         self.prix = prix
8
9 v1 = Voiture ("Peugeot", "Citadine", 5, 15000)
10 v2 = Voiture ("Citroën", "Berline", 3, 14000)
```

- ☐ Une variable de classe
- ☐ Une variable d'instance
- ☐ Une variable absolue

Question 5

Que va renvoyer le terminal ?

```

1 class Velo:
2     def __init__(self, marque, prix):
3         self.marque = marque
4         self.categorie = categorie
5         self.prix = prix
6
7 c1 = Velo ("Rockrider", "VTT", 500)
8 c2 = Velo ("Btwin", "VTC", 300)
9 print (c1.marque)

```


- ☐ Une erreur
- ☐ Rockrider
- ☐ Btwin

Solutions des exercices

Exercice p. 6 Solution n°1**Question 1**

Un(e) _____ est utilisé pour créer un objet.


- ☐ Classe
- ☒ Constructeur
- ☐ Fonction

 Un constructeur est une méthode qui va permettre de créer un objet.

Question 2

Pour créer, un objet appartenant à une classe, on utilise :

- ☐ __inite__
- ☐ __class__
- ☒ __init__


 La méthode __init__ permet de créer un objet appartenant à une classe, quand la classe est instanciée.

Question 3

Dans le code suivant, qu'affichera la commande `print(voiture1.cv)` ?

```
1 class Voiture:
2     def __init__(self, marque, cv):
3         self.marque = marque
4         self.cv = cv
5
6 voiture1 = Voiture ("Bugatti", 400)
7 voiture2 = Voiture ("Chevrolet", 350)
```

- ☒ 400
- ☐ 350
- ☐ Error
- ☐ « cv »

 Lors de la première instanciation, on affecte au paramètre `marque` la valeur « *Bugatti* » et au paramètre « `cv` » la valeur « 400 ». Puis on dit dans l'init que la marque de notre objet sera égale à la marque et que le nombre de cv de notre objet sera égal à cv. Pour la première voiture, le nombre de cv est donc 400. La commande `print(voiture1.cv)` affiche donc 400.

Question 4

L'instanciation, c'est :

- ☒ Lorsqu'on crée effectivement un objet appartenant à une classe
- ☐ Lorsqu'on écrit : `__init__`
- ☐ Un morceau de code qui ne peut exister qu'une fois dans un script

Q L'instanciation, c'est lorsqu'on crée effectivement notre objet à partir d'une classe. C'est à partir de ce moment que notre objet existe. On peut instancier autant d'objets qu'on veut, qui correspondront alors tous au modèle de leur classe.

Question 5

Quelle sera la sortie de l'extrait de code suivant ?

```
1 class Sales:
2     def __init__(self, id):
3         self.ide = id
4         id = 100
5 val = Sales(123)
6 print (val.ide)
```

- ☐ SyntaxError
- ☐ 100
- ☒ 123

Q Lorsqu'on instancie la classe « *Sales* », on utilise l'argument « *123* ». Dans la première ligne de code appartenant au `__init__`, on dit que lors de l'instanciation, la valeur de la variable « *ide* » sera égale à « *id* » donc à 123. Ce n'est qu'après que la variable « *id* » change de valeur.

Exercice p. 9 Solution n°2

Question 1

Le mot clé « *self* » sert à :

- ☐ Faire appel à une fonction modulo
- ☒ Créer un attribut d'instance
- ☐ Rien

Q Le mot clé « *self* » sert à créer un attribut d'instance car il fait référence à l'instance de la classe. Il va donc viser directement les attributs de l'objet créé.

Question 2

Une variable de classe :

- ☐ Sera différente pour chaque objet créé
- ☒ Sera la même pour chaque objet créé

Q Une variable de classe sera la même pour chaque objet créé. Chaque objet aura cette variable comme attribut.

Question 3

La bonne méthode pour instancier la classe « *Dog* » est :

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

- ☐ Dog()
- ☐ Dog.create("Rufus", 3)
- ☒ Dog("Rufus", 3)
- ☐ Dog.__init__("Rufus", 3)


 Lorsqu'on instancie la classe « *Dog* » avec « *Dog("Rufus", 3)* », le premier paramètre est « *Rufus* », ce qui veut dire qu'on aura « *name=Rufus* », et le deuxième paramètre est « *3* », ce qui veut dire qu'on aura « *age=3* ».

Question 4

Dans le code suivant :

```
1 class Gateau:
2     oeuf = 4
3     def __init__(self, farine):
4         self.farine = farine
```

- ☐ Chaque gâteau aura un nombre différent d'œufs
- ☐ Chaque gâteau aura la même quantité de farine
- ☒ Chaque gâteau aura un nombre identique d'œufs


 « *œuf* » est une variable de classe. Cette variable sera un attribut identique à chaque objet instancié. À l'inverse, « *self.farine* » est une variable d'instance, elle sera donc propre à chaque objet créé.

Question 5

Que va renvoyer le programme suivant ?

```
1 class Voiture:
2     def __init__(marque, catégorie):
3         self.marque = marque
4         self.catégorie = catégorie
5
6 v1 = Voiture ("Peugeot", "Citadine")
7 print (v1.marque)
```

- ☐ Peugeot
- ☐ Catégorie
- ☒ Une erreur

 Le paramètre « *self* » n'est pas mentionné dans les paramètres du `__init__`. Python indique donc que le nombre d'arguments indiqués est de 2 alors que 3 arguments (ou paramètres) sont en réalité donnés : `TypeError: __init__() takes 2 positional arguments but 3 were given`

p. 10 Solution n°3

Voici une solution possible :

```
1 class Voiture:
2     catégorie = "citadine"
3     nbportes = 3
4     def __init__(self, marque, cv, prix) :
5         self.marque = marque
6         self.cv = cv
7         self.prix = prix
```

On peut distinguer deux types de variables différentes : « *catégorie* » et « *nbportes* » sont des variables de classe et « *self.marque* », « *self.cv* » et « *self.prix* » sont des variables d'instance.

p. 10 Solution n°4

Voici une solution possible :

```
1 class Voiture:
2     catégorie = "citadine"
3     nbportes = 3
4     def __init__(self, marque, cv, prix) :
5         self.marque = marque
6         self.cv = cv
7         self.prix = prix
8
9 v1 = Voiture ("Peugeot", 80, 15000)
10 v2 = Voiture ("Citroën", 75, 14000)
11
12 print (v1.marque)
13 print (v1.catégorie)
14 print (v1.prix)
15
16 print (v2.marque)
17 print (v2.catégorie)
18 print (v2.prix)
```

On constate que les marques et les prix sont différents mais que la catégorie est la même. C'est dû à la différence entre les variables d'instance et de classe.

Exercice p. 10 Solution n°5

Question 1


Qu'est-ce qu'un constructeur ? :

- ☐ Une méthode qui permet de créer une classe
- ☒ Une méthode qui permet de créer un objet
- ☐ Une méthode permettant de construire une maison
- ☒ Un constructeur est une méthode qui permet de créer un objet à partir d'une classe.

Question 2


Quelle sera la sortie de l'extrait de code suivant ?

```
1 class Test:
2     def __init__(self,a="Hello World"):
3         self.a = a
4     def display(self):
5         print(self.a)
6 obj=Test()
7 obj.display()
```


- ☐ Le programme imprime une erreur car le constructeur ne peut pas avoir d'argument par défaut
 - ☐ Rien ne s'affiche
 - ☒ « Hello World » s'affiche
 - ☐ Erreur de syntaxe
-  Grâce au `__init__`, la valeur de l'argument « *a* » (inscrit dans les parenthèses et qui est égal à "Hello World") est affectée à la variable d'instance « *self.a* ». La classe « *Test* » est instanciée et l'objet est affecté à la variable « *obj* ». Lorsqu'on appelle la fonction « *display* » (avec « *self* » en argument, qui fait donc référence à l'instance), la fonction imprime la variable d'instance « *self.a* » qui est donc égale à « "Hello World" ».

Question 3


Quel type de variable permet de définir le mot clé « *self* » ?

- ☐ Une variable de classe
 - ☐ Toutes les variables
 - ☒ Une variable d'instance
 - ☐ Une variable péremptoire
-  Lorsqu'on utilise le mot clé « *self* », on fait référence à l'instance de la classe. On peut, grâce à « *self* », définir des variables d'instances, qui seront donc propres à chaque objet créé.

Question 4

Quel type de variable est « *cv* » ?

```
1 class Voiture:
2     cv=90
3     def __init__(self, marque, catégorie, portes, prix):
4         self.marque = marque
5         self.catégorie = catégorie
6         self.portes = portes
7         self.prix = prix
8
9 v1 = Voiture ("Peugeot", "Citadine", 5, 15000)
10 v2 = Voiture ("Citroën", "Berline", 3, 14000)
```

- ☒ Une variable de classe
 - ☐ Une variable d'instance
 - ☐ Une variable absolue
-  « *cv* » est une variable de classe. En effet, tous les objets créés auront le même attribut « *cv* » tel que « *cv=90* ». Sémantiquement, ça veut dire que chaque voiture créée aura le même nombre de chevaux.

Question 5

Que va renvoyer le terminal ?


```
1 class Velo:
2     def __init__(self, marque, prix):
3         self.marque = marque
4         self.categorie = categorie
5         self.prix = prix
6
7 c1 = Velo ("Rockrider", "VTT", 500)
```

```
8 c2 = Velo ("Btwin", "VTC", 300)
9 print (c1.marque)
```

☒ Une erreur

☐ Rockrider

☐ Btwin

 La variable catégorie n'est pas renseignée dans les arguments de la méthode `__init__`. Le terminal renvoie donc : « *TypeError: __init__() takes 3 positional arguments but 4 were given* ». Cette erreur signifie qu'il n'y a que 3 arguments renseignés alors que 4 sont utilisés.