

L'héritage

Table des matières

I. Notion de base en héritage	3
II. Exercice : Quiz	5
III. Création de hiérarchies de classes	6
IV. Exercice : Quiz	10
V. Essentiel	12
VI. Auto-évaluation	12
A. Exercice	12
B. Test	12
Solutions des exercices	13

I. Notion de base en héritage

Durée : 1 h

Environnement de travail : PC connecté à internet

Contexte

En programmation, l'héritage est un aspect important du paradigme orienté objet. Il permet de réutiliser du code déjà écrit pour éviter les répétitions de ce même code dans votre programme. De manière générale, l'héritage est le mécanisme de dérivation d'une nouvelle classe à partir d'une classe existante. Ce faisant, nous obtenons une hiérarchie des classes.

Dans la plupart des langages orientés objet basés sur des classes, un objet créé par héritage (**objet enfant**) acquiert toutes les propriétés et tous les comportements de l'**objet parent**. Cela facilite la maintenance du code et augmente la productivité. De plus, la structuration du programme engendrée par l'utilisation du mécanisme d'héritage facilite grandement la compréhension.

Définition

L'héritage modélise ce que l'on appelle une relation. Cela signifie que lorsque vous avez une classe dérivée qui hérite d'une classe de base, vous avez créé une relation où la classe dérivée est une version spécialisée de la classe de base. L'héritage est représenté à l'aide du langage de modélisation unifié ou UML de la manière suivante :

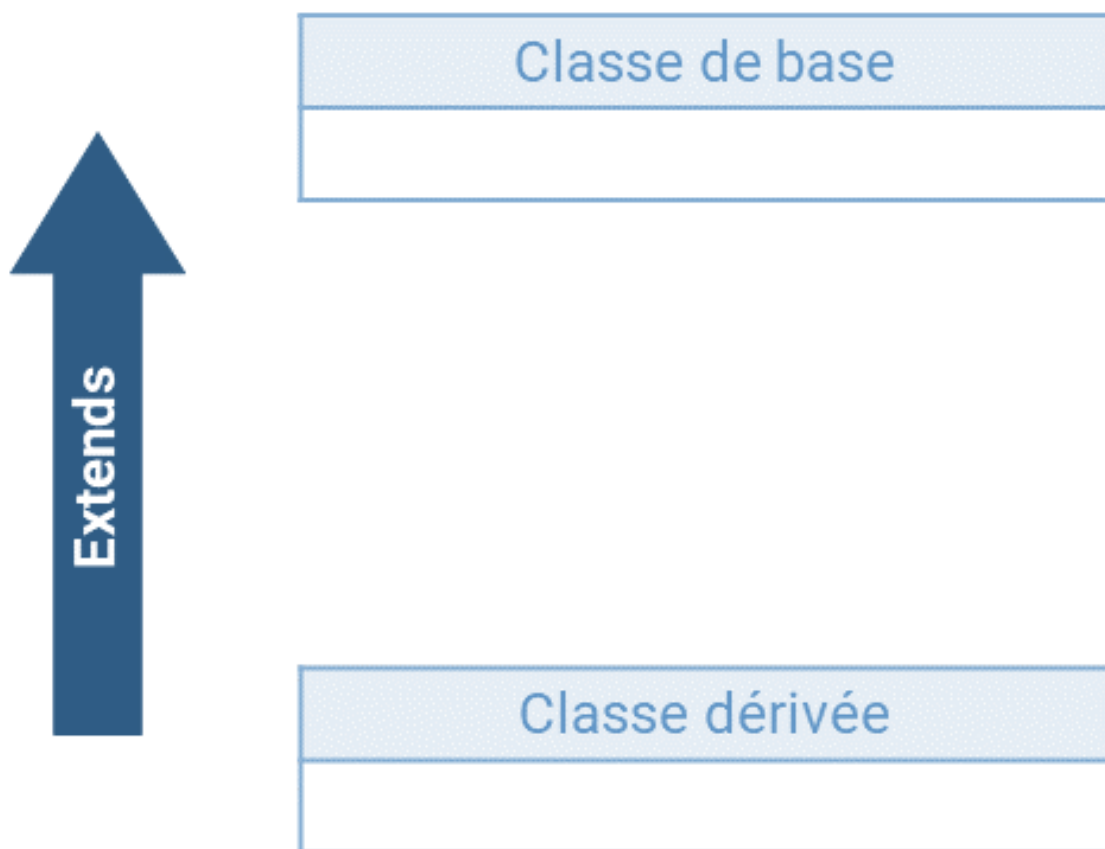


Illustration de l'héritage en programmation

Les classes sont représentées par des boîtes avec le nom de la classe en haut. La relation d'héritage est représentée par une flèche partant de la classe dérivée et pointant vers la classe de base. Le mot **extends** est généralement ajouté à la flèche.

Remarque

Les classes qui héritent d'une autre sont appelées **classes dérivées**, sous-classes ou sous-types. Les classes dont d'autres classes sont dérivées sont appelées classes de base ou super classes. On dit d'une classe dérivée qu'elle dérive, hérite ou étend une classe de base.

Disons que vous avez une classe de base Animal et que vous en dérivez pour créer une classe Cheval. La relation d'héritage stipule qu'un Cheval est un Animal. Cela signifie que Cheval hérite de l'interface et de l'implémentation de Animal, et que les objets Cheval peuvent être utilisés pour remplacer les objets Animal dans l'application.

C'est ce qu'on appelle le principe de substitution de Liskov. Ce principe stipule que « *dans un programme informatique, si S est un sous-type de T, alors les objets de type T peuvent être remplacés par des objets de type S sans altérer aucune des propriétés souhaitées du programme* ». Vous verrez dans cet article pourquoi vous devez toujours suivre le principe de substitution de Liskov lorsque vous créez vos hiérarchies de classes, et les problèmes que vous rencontrerez si vous ne le faites pas.

Tout en Python est un objet. Les modules sont des objets, les définitions de classes et les fonctions sont des objets, et bien sûr, les objets créés à partir de classes sont aussi des objets. L'héritage est une caractéristique requise de tout langage de programmation orienté objet. Cela signifie que Python prend en charge l'héritage et, comme vous le verrez plus tard, c'est l'un des rares langages qui prend en charge l'héritage multiple. Lorsque vous écrivez du code Python à l'aide de classes, vous utilisez l'héritage même si vous ne le savez pas. Voyons ce que cela signifie.

Exemple

La façon la plus simple de voir l'héritage en Python est de se mettre dans le shell interactif Python et d'écrire un peu de code. Vous commencerez par écrire la classe la plus simple possible :

```
1 >>> class MyClass:
2 ...     pass
3 ...
```

Vous avez déclaré une classe MyClass qui ne fait pas grand-chose, mais elle illustre les concepts d'héritage les plus élémentaires. Maintenant que vous avez déclaré la classe, vous pouvez utiliser la fonction `dir()` pour lister ses membres :

```
1 >>> c = MyClass()
2 >>> dir(c)
3 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
4  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
5  '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
6  '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
7  '__str__', '__subclasshook__', '__weakref__']
```

`dir()` renvoie une liste de tous les membres de l'objet spécifié. Vous n'avez déclaré aucun membre dans `MyClass`, alors d'où vient la liste ? Vous pouvez le découvrir à l'aide de l'interpréteur interactif :

```
1 >>> o = object()
2 >>> dir(o)
3 ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
4  '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
5  '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
6  '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
7  '__subclasshook__']
```

Comme vous pouvez le voir, les deux listes sont presque identiques. Il y a quelques membres supplémentaires dans `MyClass` comme `__dict__` et `__weakref__`, mais chaque membre de la classe `Object` est également présent dans `MyClass`.

En effet, chaque classe que vous créez en Python dérive implicitement de **Object**. Vous pourriez être plus explicite et écrire `class MyClass (object):`, mais c'est redondant et inutile. Notez que dans Python 2, vous devez explicitement dériver de `Object` pour des raisons dépassant le cadre de ce cours, mais vous pouvez en savoir plus en faisant des recherches approfondies sur le sujet.

Par ailleurs, il faut savoir qu'il existe une exception au fait que chaque classe en python dérive implicitement de **Object**. L'exception à cette règle concerne les classes utilisées pour indiquer des erreurs en levant une exception. Ces classes doivent forcément dériver de la classe **BaseException**. Pour ce faire, on dérive les classes d'erreur personnalisée de la classe parent **Exception**, qui à leur tour dérivent de `BaseException`.

Exercice : Quiz

[solution n°1 p.15]

Question 1

Lors de l'héritage d'une classe par une autre, à quelles méthodes de classes pouvez-vous accéder ?

- ☐ Aucune
- ☐ Cela dépend de la classe
- ☐ Toutes

Question 2

Comment hériter d'une autre classe ?

- ☐ En l'appelant en tant que module sur la première ligne de votre nouvelle classe
- ☐ En passant le nom de l'autre classe en paramètre de la nouvelle classe
- ☐ En la nommant

Question 3

Lors de l'instanciation d'une nouvelle classe, quels paramètres de l'ancienne classe devez-vous passer ?

- ☐ Seuls ceux qui seront utiles dans la nouvelle classe
- ☐ Cela dépend de la classe
- ☐ Tous

Question 4

Lequel des énoncés suivants décrit le mieux l'héritage ?

- ☐ Capacité d'une classe à dériver des membres d'une autre classe dans le cadre de sa propre définition
- ☐ Moyens de regrouper les variables d'instance et les méthodes afin de restreindre l'accès à certains membres de la classe
- ☐ Se concentre sur les variables et le passage des variables aux fonctions
- ☐ Permet la mise en œuvre de logiciels élégants, bien conçus et faciles à modifier

Question 5

Laquelle des affirmations suivantes est erronée à propos de l'héritage ?

- ☐ Les membres protégés d'une classe peuvent être hérités
- ☐ La classe héritière est appelée une sous-classe
- ☐ Les membres privés d'une classe peuvent être hérités et accessibles
- ☐ L'héritage est l'une des caractéristiques de la POO

III. Création de hiérarchies de classes

Héritage et classe dérivée

L'héritage est le mécanisme que vous utiliserez pour créer des hiérarchies de classes liées. Ces classes apparentées partageront une interface commune qui sera définie dans les classes de base. Les classes dérivées peuvent spécialiser l'interface en fournissant une implémentation particulière le cas échéant.

Exemple

Dans cette section, vous commencerez à modéliser un système RH. L'exemple démontre l'utilisation de l'héritage et comment les classes dérivées peuvent fournir une implémentation concrète de l'interface de classe de base. Le système RH doit traiter la paie des employés de l'entreprise, mais il existe différents types d'employés en fonction de la façon dont leur masse salariale est calculée.

Vous commencez par implémenter une classe `PaieSysteme` qui traite de la paie des employés :

```
1 # In hr.py
2
3 class PaieSysteme:
4     def calcule_paie(self, employees):
5         print('Calculating Paie')
6         print('=====')
7         for employee in employees:
8             print(f'Paie for: {employee.id} - {employee.name}')
9             print(f'- Check amount: {employee.calcule_paie()}')
10        print('')
```

Le `PaieSysteme` implémente une méthode `.calcule_paie()` qui prend une collection d'employés et imprime leur **id**, **name** et le montant du chèque en utilisant la méthode `.calcule_paie()` exposée sur chaque objet employé. Maintenant, vous implémentez une classe de base **Employee** qui gère l'interface commune pour chaque type d'employé :

```
1 # In hr.py
2
3 class Employee:
4     def __init__(self, id, name):
5         self.id = id
6         self.name = name
```

Employee est la classe de base pour tous les types d'employés. Il est construit avec un id et un name. Cela signifie que chaque **Employee** doit avoir une affectation id ainsi qu'un nom.

Le système RH exige que chaque **Employee** traité fournisse une interface `.calcule_paie()` qui renvoie le salaire hebdomadaire de l'employé. L'implémentation de cette interface diffère selon le type d'**Employee**. Par exemple, les employés administratifs ont un salaire fixe, donc chaque semaine, ils reçoivent le même montant :

```
1 # In hr.py
2
3 class SalaireEmployee(Employee):
4     def __init__(self, id, name, SalaireSemaine):
5         super().__init__(id, name)
6         self.SalaireSemaine = SalaireSemaine
7
8     def calcule_paie(self):
9         return self.SalaireSemaine
```

Vous créez une classe dérivée **SalaireEmployee** qui hérite de **Employee**. La classe est initialisée avec le id et name requis par la classe de base, et vous utilisez `super()` pour initialiser les membres de la classe de base.

SalaireEmployee nécessite également un paramètre **SalaireSemaine** d'initialisation qui représente le montant que l'employé gagne par semaine. La classe fournit la méthode `.calcule_paie()` requise utilisée par le système RH. L'implémentation renvoie simplement le montant stocké dans **SalaireSemaine**. L'entreprise emploie également des ouvriers de fabrication qui sont payés à l'heure, vous ajoutez donc un **EmployeHoraire** au système RH :

```
1 # In hr.py
2
3 class EmployeHoraire(Employee):
4     def __init__(self, id, name, horaire_travail, taux_horaire):
5         super().__init__(id, name)
6         self.horaire_travail = horaire_travail
7         self.taux_horaire = taux_horaire
8
9     def calcule_paie(self):
10        return self.horaire_travail * self.taux_horaire
```

La classe **EmployeHoraire** est initialisée avec id et name, comme la classe de base, ainsi que les **horaire_travail** et les **taux_horaire** nécessaires au calcul de la masse salariale. La méthode `.calcule_paie()` est mise en œuvre en retournant les heures travaillées multipliées par le taux horaire.

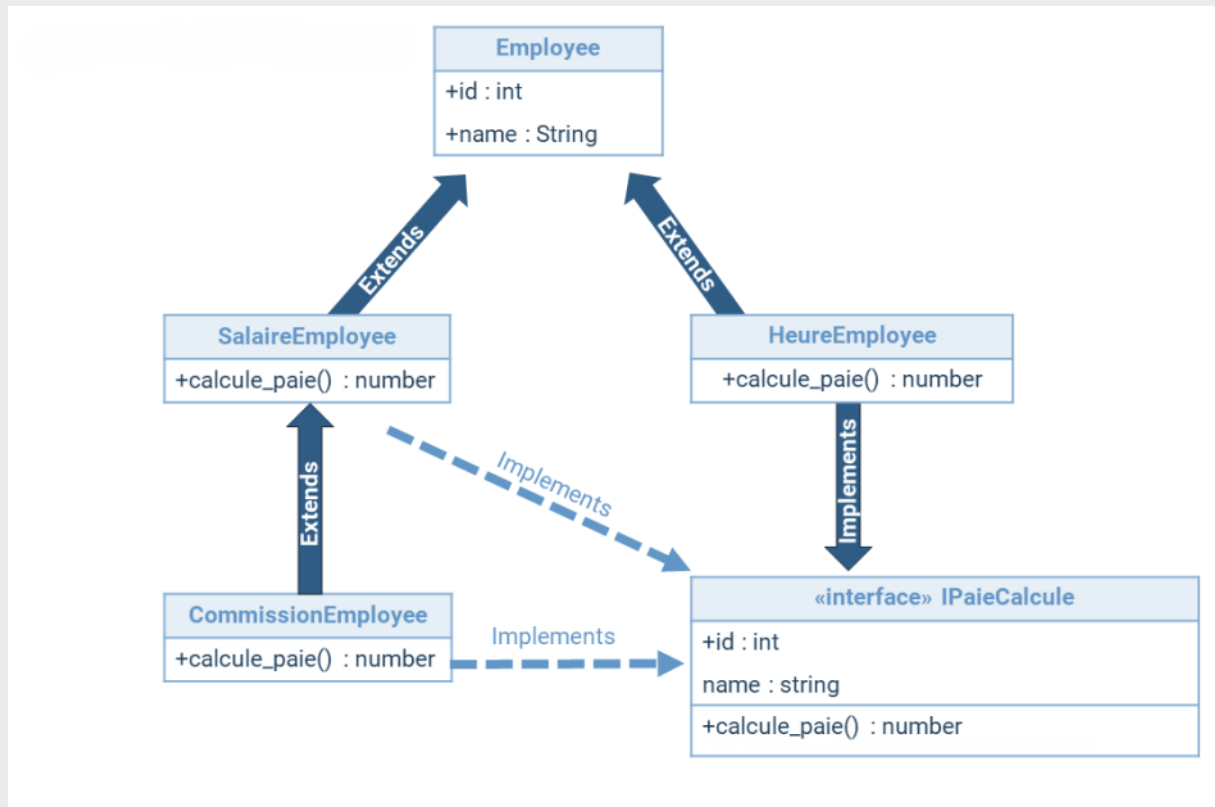
Enfin, l'entreprise emploie des vendeurs qui sont rémunérés par un salaire fixe plus une commission basée sur leurs ventes, vous créez donc une classe **CommissionEmployee** :

```
1 # In hr.py
2
3 class CommissionEmployee(SalaireEmployee):
4     def __init__(self, id, name, SalaireSemaine, commission):
5         super().__init__(id, name, SalaireSemaine)
6         self.commission = commission
7
8     def calcule_paie(self):
9         fixed = super().calcule_paie()
10        return fixed + self.commission
```

Vous dérivez **CommissionEmployee** de **SalaireEmployee** parce que les deux classes ont un **SalaireSemaine** à considérer. En même temps, **CommissionEmployee** est initialisée avec une valeur de la commission basée sur les ventes de l'employé. Par ailleurs, `.calcule_paie()` tire parti de la mise en œuvre de la classe de base pour récupérer le salaire fixé et ajouté la valeur de la commission.

Puisque **CommissionEmployee** dérive de **SalaireEmployee**, vous avez accès à **SalaireSemaine** directement à la propriété et vous auriez pu implémenter **.calculer_paie()** en utilisant la valeur de cette propriété. Le problème avec l'accès direct à la propriété est que si l'implémentation de **SalaireEmployee.calculer_paie()** change, vous devrez également modifier l'implémentation de **CommissionEmployee.calculer_paie()**. Il est préférable de s'appuyer sur la méthode déjà implémentée dans la classe de base et d'étendre les fonctionnalités si nécessaire.

Vous avez créé votre première hiérarchie de classes pour le système. Le diagramme UML des classes ressemble à ceci :



Exemple d'héritage avec plusieurs classes dérivées Employee

Le diagramme montre la hiérarchie d'héritage des classes. Les classes dérivées implémentent l'interface **IPaieCalcule** requise par le **PaieSysteme**. L'implémentation **PaieSysteme.calculer_paie()** nécessite que les **employeeobjets** passés contiennent une implémentation **id**, **name** et **calculer_paie()**. Les interfaces sont représentées de la même manière que les classes avec le mot interface au-dessus du nom de l'interface. Les noms d'interface sont généralement précédés d'une majuscule I. L'application crée ses employés et les transmet au système de paie pour traiter la paie :

```

1 # In program.py
2
3 import hr
4
5 salary_employee = hr.SalaireEmployee(1, 'John Smith', 1500)
6 hourly_employee = hr.EmloyeeHoraire(2, 'Jane Doe', 40, 15)
7 commission_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)
8 Paie_system = hr.PaieSysteme()
9 Paie_system.calculer_paie([
10     salary_employee,
11     hourly_employee,
12     commission_employee

```



```
13 ])
```

Vous pouvez exécuter le programme dans la ligne de commande et voir les résultats :

```
1 $ python program.py
2
3 Calculating Paie
4 =====
5 Paie for: 1 - John Smith
6 - Check amount: 1500
7
8 Paie for: 2 - Jane Doe
9 - Check amount: 600
10
11 Paie for: 3 - Kevin Bacon
12 - Check amount: 1250
```

Le programme crée trois objets employés, un pour chacune des classes dérivées. Ensuite, il crée le système de paie et transmet une liste des employés à sa méthode `.calcule_paie()`, qui calcule la paie de chaque employé et imprime les résultats.

Remarquez que la classe **Employeeclasse** de base ne définit pas de méthode `.calcule_paie()`. Cela signifie que si vous deviez créer un **Employeeobjet** simple et le transmettre au **PaieSysteme**, vous obtiendrez une erreur. Vous pouvez l'essayer dans l'interpréteur interactif Python :

```
1 >>> import hr
2 >>> employee = hr.Employee(1, 'Invalid')
3 >>> Paie_system = hr.PaieSysteme()
4 >>> Paie_system.calcule_paie([employee])
5
6 Paie for: 1 - Invalid
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   File "/hr.py", line 39, in calcule_paie
10     print(f'- Check amount: {employee.calcule_paie()}')
11 AttributeError: 'Employee' object has no attribute 'calcule_paie'
```

Bien que vous puissiez instancier un **Employeeobjet**, l'objet ne peut pas être utilisé par le **PaieSysteme**. Pourquoi ? Parce que ce n'est pas possible de faire `.calcule_paie()` pour un **Employee**.

Pour répondre aux exigences de **PaieSysteme**, vous souhaitez convertir la **Employeeclasse**, qui est actuellement une classe concrète, en une classe abstraite. De cette façon, aucun employé n'est jamais juste un **Employee**, mais un qui implémente `.calcule_paie()`.

Classes de base abstraites en Python

La **Employeeclasse** dans l'exemple ci-dessus est ce qu'on appelle une classe de base abstraite. Les classes de base abstraites existent pour être héritées, mais jamais instanciées. Python fournit le module `abc` pour définir des classes de bases abstraites.

Vous pouvez utiliser des traits de soulignement de début dans le nom de votre classe pour indiquer que les objets de cette classe ne doivent pas être créés. Les traits de soulignement offrent un moyen simple d'empêcher l'utilisation abusive de votre code, mais ils n'empêchent pas les utilisateurs impatients de créer des instances de cette classe. Le module `abc` de la bibliothèque standard Python fournit des fonctionnalités pour empêcher la création d'objets à partir de classes de bases abstraites.

Exemple

En Python, vous pouvez modifier l'implémentation d'une classe abstraite pour qu'elle ne puisse pas être instanciée :

```
1 # In hr.py
2
3 from abc import ABC, abstractmethod
4
5 class Employee(ABC):
6     def __init__(self, id, name):
7         self.id = id
8         self.name = name
9
10    @abstractmethod
11    def calculate_payroll(self):
12        pass
```

Vous décrivez **Employee** de ABC, ce qui en fait une classe de base abstraite. Ensuite, vous décorez la **méthode .calculate_payroll()** avec le décorateur `@abstractmethod`. Ce changement a deux effets secondaires intéressants :

- Vous dites aux utilisateurs du module que les objets de type **Employee** ne peuvent pas être créés.
- Vous dites aux autres développeurs travaillant sur le module hr que s'ils dérivent de **Employee**, ils doivent remplacer la méthode **.calculate_payroll()** abstraite.

Vous pouvez voir que les objets de type **Employee** ne peuvent pas être créés à l'aide de l'interpréteur interactif :

```
1 >>> import hr
2 >>> employee = hr.Employee(1, 'abstract')
3
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 TypeError: Can't instantiate abstract class Employee with abstract methods
7 calculate_payroll
```

La sortie montre que la classe ne peut pas être instanciée car elle contient une méthode abstraite **calculate_payroll()**. Les classes dérivées doivent remplacer la méthode pour permettre la création d'objets de leur type.

Pour résumer, les classes abstraites sont utilisées pour créer un plan de nos classes car elles ne contiennent pas l'implémentation de la méthode. Il s'agit d'une fonctionnalité très utile, en particulier dans les situations où les classes enfants doivent fournir leur implémentation distincte. De plus, dans les projets complexes impliquant de grandes équipes et une énorme base de code, il est assez difficile de se souvenir de tous les noms de classe.

Par ailleurs, l'importance d'utiliser la classe abstraite en Python est que si les sous-classes ou classes dérivées ne suivent pas un plan strict, Python donnera une erreur. Ainsi, nous pouvons nous assurer que nos classes suivent la structure donnée et implémentent toutes les méthodes abstraites définies dans notre classe abstraite.

Exercice : Quiz

[solution n°2 p.16]

Question 1

Quelle sera la sortie du code Python suivant ?

```
1 class A():
2     def disp(self):
3         print("A disp()")
4 class B(A):
5     pass
6 obj = B()
7 obj.disp()
```

- ☐ Syntaxe non valide pour l'héritage
- ☐ Erreur car lors de la création de l'objet, l'argument doit être passé
- ☐ Rien n'est imprimé
- ☐ A disp()

Question 2

Quelle sera la sortie du code Python suivant ?

```
1 class Test:
2     def __init__(self):
3         self.x = 0
4 class Derived_Test(Test):
5     def __init__(self):
6         self.y = 1
7 def main():
8     b = Derived_Test()
9     print(b.x,b.y)
10 main()
```

- ☐ 0 1
- ☐ 0 0
- ☐ Erreur car la classe B hérite de A mais la variable x n'est pas héritée
- ☐ Erreur car lorsque l'objet est créé, l'argument doit être passé comme Derived_Test (1)

Question 3

Si une classe est dérivée de deux classes différentes, on parle :

- ☐ D'héritage multilevel
- ☐ D'héritage multiple
- ☐ D'héritage hiérarchique
- ☐ D'héritage Python

Question 4

Quelle affirmation se rapproche le plus de la définition de la succession ?

- ☐ C'est un mécanisme qui permet aux objets de changer de type
- ☐ Il s'agit d'une relation de spécialisation entre une classe et ses sous-classes
- ☐ Il est synonyme du terme « *instanciation* »
- ☐ Il permet à un objet de connaître la valeur d'un attribut d'un autre objet

Question 5

Lorsqu'une classe B hérite d'une classe A, la classe B ne peut plus contenir d'autres méthodes en dehors de celles de la classe A dont elle a hérité.

- ☐ Vrai
- ☐ Faux

V. Essentiel

L'héritage est l'un des concepts les plus importants de la POO. Il permet la réutilisation du code, sa lisibilité et la transition des propriétés, ce qui contribue à l'optimisation et à l'efficacité de la construction du code.

Le langage de programmation Python est chargé de concepts tels que l'héritage. L'ampleur des applications Python nécessite un nombre croissant de programmeurs Python sur le marché actuel. Pour maîtriser vos compétences et donner un coup de fouet à votre apprentissage, vous devez bien maîtriser ce concept.

VI. Auto-évaluation

A. Exercice

Python est l'un des langages de programmation les plus populaires. Malgré une transition pleine de hauts et de bas de la version Python 2 à Python 3, le langage de programmation orienté objet a connu un énorme bond en popularité. Comme dit précédemment, la relation d'héritage définit les classes qui héritent d'autres classes en tant que classes dérivées, sous-classes ou sous-types.

Question 1

[solution n°3 p.17]

Donner de manière très précise ce que l'héritage implique pour la classe mère et la classe enfant.

Question 2

[solution n°4 p.17]

Expliquer clairement l'héritage multi-niveau en python

B. Test

Exercice 1 : Quiz

[solution n°5 p.18]

Question 1

L'héritage permet d'étendre des classes mères dans des sous-classes.

- ☐ Vrai
- ☐ Faux

Question 2

La classe mère ne contient que des attributs réutilisables.

- ☐ Vrai
- ☐ Faux

Question 3

Le code `class C (A, B):` signifie que la classe C hérite de la classe B et de la classe A.

- ☐ Vrai
- ☐ Faux

Exercice

L'héritage multiple suppose que plusieurs sous-classes héritent d'une même classe mère.

- ☐ Vrai
- ☐ Faux

Question 5


La sous-classe d'une classe enfant hérite aussi de la classe mère.

- ☐ Vrai
- ☐ Faux

Solutions des exercices


Exercice p. 5 Solution n°1**Question 1**

Lors de l'héritage d'une classe par une autre, à quelles méthodes de classes pouvez-vous accéder ?

- ☐ Aucune
- ☐ Cela dépend de la classe
- ☒ Toutes
-  Toutes les méthodes qui n'ont pas été déclarées privées sont accessibles.


Question 2

Comment hériter d'une autre classe ?

- ☐ En l'appelant en tant que module sur la première ligne de votre nouvelle classe
- ☒ En passant le nom de l'autre classe en paramètre de la nouvelle classe
- ☐ En la nommant
-  Pour faire hériter la nouvelle classe de l'ancienne, il faut un paramètre de la classe à hériter.


Question 3

Lors de l'instanciation d'une nouvelle classe, quels paramètres de l'ancienne classe devez-vous passer ?

- ☐ Seuls ceux qui seront utiles dans la nouvelle classe
- ☐ Cela dépend de la classe
- ☒ Tous
-  L'instanciation d'une classe suppose que vous passez les paramètres de la classe mère dans la nouvelle.


Question 4

Lequel des énoncés suivants décrit le mieux l'héritage ?

- ☒ Capacité d'une classe à dériver des membres d'une autre classe dans le cadre de sa propre définition
- ☐ Moyens de regrouper les variables d'instance et les méthodes afin de restreindre l'accès à certains membres de la classe
- ☐ Se concentre sur les variables et le passage des variables aux fonctions
- ☐ Permet la mise en œuvre de logiciels élégants, bien conçus et faciles à modifier
-  L'héritage en Python fonctionne comme un passage de propriétés entre une classe et une sous-classe.

Question 5

Laquelle des affirmations suivantes est erronée à propos de l'héritage ?


- ☐ Les membres protégés d'une classe peuvent être hérités
- ☐ La classe héritière est appelée une sous-classe
- ☒ Les membres privés d'une classe peuvent être hérités et accessibles
- ☐ L'héritage est l'une des caractéristiques de la POO
-  Lorsqu'une classe a des attributs privés, ceux-ci ne peuvent être hérités ou être modifiés par les sous-classes.

Exercice p. 10 Solution n°2

Question 1

Quelle sera la sortie du code Python suivant ?


```
1 class A():
2     def disp(self):
3         print("A disp()")
4 class B(A):
5     pass
6 obj = B()
7 obj.disp()
```

- ☐ Syntaxe non valide pour l'héritage
- ☐ Erreur car lors de la création de l'objet, l'argument doit être passé
- ☐ Rien n'est imprimé
- ☒ A disp()
-  La fonction disp est héritée par la classe B.

Question 2


Quelle sera la sortie du code Python suivant ?

```
1 class Test:
2     def __init__(self):
3         self.x = 0
4 class Derived_Test(Test):
5     def __init__(self):
6         self.y = 1
7 def main():
8     b = Derived_Test()
9     print(b.x,b.y)
10 main()
```

- ☐ 0 1
- ☐ 0 0
- ☒ Erreur car la classe B hérite de A mais la variable x n'est pas héritée
- ☐ Erreur car lorsque l'objet est créé, l'argument doit être passé comme Derived_Test (1)
-  Si dans une sous-classe, une variable n'est pas héritée, elle ne peut être affichée.


Question 3

Si une classe est dérivée de deux classes différentes, on parle :

- ☐ D'héritage multilevel
- ☒ D'héritage multiple
- ☐ D'héritage hiérarchique
- ☐ D'héritage Python
-  En Python, il y a la possibilité pour une sous-classe d'hériter de deux classes mères. On appelle ça l'héritage multiple.


Question 4

Quelle affirmation se rapproche le plus de la définition de la succession ?

- ☐ C'est un mécanisme qui permet aux objets de changer de type
- ☒ Il s'agit d'une relation de spécialisation entre une classe et ses sous-classes
- ☐ Il est synonyme du terme « *instanciation* »
- ☐ Il permet à un objet de connaître la valeur d'un attribut d'un autre objet
-  Lors de la succession, les sous-classes interviennent pour étendre les fonctionnalités de la classe mère.

Question 5

Lorsqu'une classe B hérite d'une classe A, la classe B ne peut plus contenir d'autres méthodes en dehors de celles de la classe A dont elle a hérité.

- ☐ Vrai
- ☒ Faux
-  Non seulement elle peut hériter de tous les attributs et méthodes disponibles dans la classe A, mais elle peut aussi contenir de nouvelles méthodes pour effectuer des actions pour cette classe enfant

p. 12 Solution n°3

Une utilisation courante de l'héritage est la classification hiérarchique des objets. Supposons qu'il y ait une superclasse qui a plusieurs sous-classes. Étant donné que toutes les sous-classes ont les mêmes opérations que leur superclasse, elles peuvent être utilisées partout où la superclasse est attendue. La superclasse spécifie alors ce qui est commun à toutes les sous-classes, et les sous-classes peuvent indiquer en quoi elles diffèrent des caractéristiques communes. La superclasse spécifie le type général de chose, et les sous-classes spécifient les variations. Cela correspond à notre façon de définir les choses.

p. 12 Solution n°4

Python est composé de plusieurs objets, et avec l'héritage multi-niveaux, les possibilités de réutilisation des fonctionnalités de la classe sont infinies. L'héritage à plusieurs niveaux est défini chaque fois qu'une classe dérivée hérite d'une autre classe dérivée. Il n'y a pas de limite au nombre de classes dérivées pouvant hériter des fonctionnalités, et c'est pourquoi l'héritage à plusieurs niveaux contribue à améliorer la réutilisabilité en Python.


Exercice p. 12 Solution n°5

Question 1

L'héritage permet d'étendre des classes mères dans des sous-classes.

☒ Vrai

☐ Faux


 L'héritage est le concept POO qui permet de créer des sous-classes pour utiliser et ajouter des fonctions aux classes mères.

Question 2

La classe mère ne contient que des attributs réutilisables.

☐ Vrai

☒ Faux


 Une classe mère contient un mélange d'attributs, qu'ils soient privés ou non. Les attributs privés ne peuvent être hérités.

Question 3

Le code `class C (A, B)`: signifie que la classe C hérite de la classe B et de la classe A.

☒ Vrai

☐ Faux


 Quand une classe hérite de deux classes mères, on met les deux classes mères en attributs séparés par une virgule.

Exercice

L'héritage multiple suppose que plusieurs sous-classes héritent d'une même classe mère.

☐ Vrai

☒ Faux


 On parle d'héritage multiple lorsqu'une sous-classe peut hériter de plusieurs classes mères différentes.

Question 5

La sous-classe d'une classe enfant hérite aussi de la classe mère.

☒ Vrai

☐ Faux

 Si une classe enfant hérite des propriétés de la classe parent, la sous-classe de la classe enfant hérite automatiquement des propriétés de la classe parent.