Python

Orienté objet

Table des matières

Table des matières

Introduction

Paradigme orienté objet

Le paradigme orienté objet a pour objectif de définir des briques logicielles appelées des objets. Ces derniers peuvent représenter des entités physiques ou encore des concepts ou des idées.

Chaque objet possède une structure interne et un comportement.

Le concept d'objet

Il s'agit du point central du paradigme orienté objet.

Un objet est donc une entité modélisant une idée, un concept ou toutes entités physiques du monde réel.

Un objet est caractérisé par :

- Un état
- Un comportement

Coder dans plusieurs fichiers

Pour développer de manière structurée, nous allons coder dans plusieurs fichiers.

Pour que cela soit possible, il sera nécessaire de réaliser un import entre les fichiers.

```
from Package.Fichier import ma_fonction
```

- from → Fichier contenant le code à importer.
  Chemin complet : Répertoire en package + Nom du fichier.
- import → Nom de l'élément à rendre disponible dans le code.
  Il est possible d'importer une fonction, une variable ou une classe.

Exemple d'import

FichierPrincipal.py

```
from FichierExterne import puissance

nombre = 5
exposant = 3

resultat = puissance(nombre, exposant)
print(resultat)
```

FichierExterne.py

```
def puissance(nb, exp):
    resultat = 1
    count = 0
    while count < exp:
        resultat *= nb
        count += 1
    return resultat
```

Limiter l'exécution du code

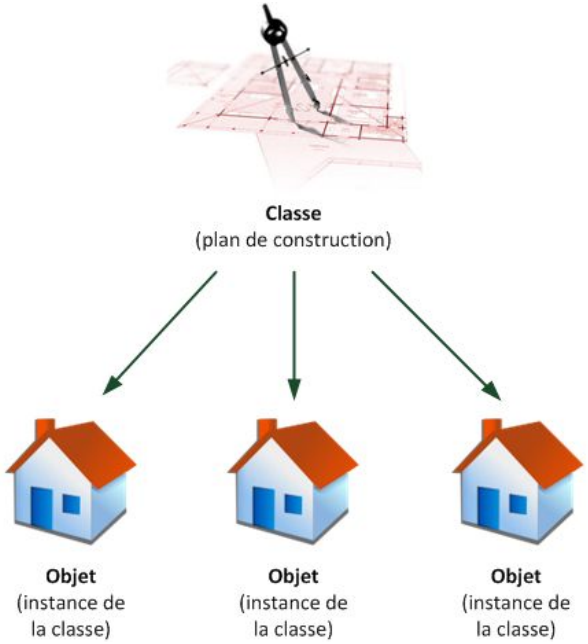
Lors de l'import d'un fichier, nous ne souhaitons pas exécuter le code de celui-ci.

Il faut donc limiter l'exécution du code au sein du fichier.
Pour cela, il faut utiliser le nom du scope accessible à l'aide de la variable "*__name__*".
La valeur de celle-ci vaut "*__main__*" lorsque le script est le point d'entrée.

```
if __name__ == "__main__":
    # execute only if run as a script
    print("Hello World")
```

Les classes

Définition d'une classe



Une classe définit un type d'objet, leurs états (champs) et leurs comportements (méthodes).

Une classe peut être vue comme un plan de construction ou un moule permettant de créer des objets du type défini par la classe.

Un objet appartenant à une classe est une instance de cette classe. On peut créer autant d'objets que l'on désire avec une classe .

Une classe en python

Le mot clef « class » permet de créer la structure d'une classe.

```
# Définition de la classe Voiture
class Voiture:
    pass


# Création d'un objet de type Voiture
v1 = Voiture()
```

La convention de nommage pour les classes est également le « PascalCase »

Les attributs

Les attributs de classe permettent de stocker des informations au niveau de la classe.

La méthode « __init__ » permet de définir des attributs de classe. Celle-ci est automatique appelée lors de la création d'un objet.

Le langage python permet également d'ajouter dynamiquement des attributs sur un objet.

```python
# Définition de la classe Personne
class Personne:

    def __init__(self):
        self.prenom = "John"
        self.nom = "Smith"


# Création d'un objet Personne
p = Personne()

# Ajout d'un attribut sur l'objet "p"
p.age = 42

# Affichage...
print(p.prenom, p.nom, p.age)
```

Utilisation de la méthode « __init__ »

La méthode « __init__ » permet de définir et d'initialiser les attributs.

La méthode peut être définie avec des paramètres, pour recevoir des données lors de l'instanciation d'un objet.

```python
# Définition de la classe Personne
class Personne:

    # Méthode d'initialisation
    def __init__(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom


# Création d'un objet avec ses paramètres
p = Personne("Balthazar", "Picsou")

# Affichage...
print(p.prenom, p.nom)
```

Les méthodes

Les méthodes sont des procédures et des fonctions au sein de la classe.

Celles-ci permettent de définir les comportements de la classe.

Le premier paramètre (obligatoire) récupère l'instance de l'objet.
Par convention, celui-ci aura comme nom « self ».

```python
# Définition de la classe Voiture
class Voiture:

    def __init__(self):
        self.vitesse = 0

    def accelerer(self):
        self.vitesse += 50

    def freiner(self):
        self.vitesse -= 50


# Création d'un objet Voiture
p = Voiture()

# Utilisation d'une méthode de la classe
p.accelerer()
```

Encapsulation

Le but de l'encapsulation

L'encapsulation est un concept primordial de l'orienté objet.

Il s'agit d'une règle consistant à cacher les données internes d'un objet, en autorisant l'accès à ces dernières uniquement avec des appels de méthodes.

L'encapsulation apporte de nombreux avantages :

- Elle permet un couplage faible des classes
- Elle permet de garantir l'intégrité des données internes d'une classe

Mise en place

Deux éléments nous permettent de mettre en place l'encapsulation :

- Les modificateurs de visibilité
- Les accesseurs et les mutateurs

Le langage python ne supportant pas les modificateurs de visibilité. Celui-ci utilise une convention d'écriture à l'aide de préfixe d'underscore, pour définir la visibilité "privé"...

- Un underscore : accessible directement via son nom.
- Deux underscores : accessible via un nom généré sous la forme *_Class__variable*

Si vous utilisez un IDE, un "Warning" est ajouté lorsque vous utilisez un élément "privé"

Les propriétés

Une propriété est un ensemble de méthodes qui permettent de contrôler les accès aux attributs de la classe.

Une propriété est composée de :

- Un accesseur (Getter) permet de consulter la valeur d'un champ
- Un mutateur (Setter) permet de modifier la valeur d'un champ

```python
class Voiture:

    def __init__(self):
        self.__roues = 4

    # Accesseur
    def _get_roues(self):
        return self.__roues

    # Mutateur
    def _set_roues(self, value):
        self.__roues = value

    # Definition de la propriété
    roues = property(_get_roues, _set_roues)


ma_voiture = Voiture()
ma_voiture.roues = 5
```

Syntaxe à l'aide des décorateurs

Le python dispose d'une syntaxe basée sur 2 décorateurs, qui permet de créer une propriété de manière simple.

Ceux-ci sont :

- *@property*
  Permet de définir la propriété, il se place sur l'accesseur

- *@<NomProp>.setter*
  Permet de définir le mutateur de la propriété.

```python
class Voiture:

    def __init__(self):
        self.__roues = 4

    # Propriété & Accesseur
    @property
    def roues(self):
        return self.__roues

    # Mutateur
    @roues.setter
    def roues(self, value):
        self.__roues = value


ma_voiture = Voiture()
ma_voiture.roues = 5
print(ma_voiture.roues)
```

Data model

Les attributs spéciaux

Les attributs spéciaux (entourés de double underscore) sont accessibles depuis le type d'une classe ou l'instance d'une classe. Ceux-ci sont en lecture seule.

Nom	Accessible sur	Objectif
__name__	classe	Renvoie le nom de la classe
__bases__	classe	Renvoie un tuple des classes parentes de la classe.
__class__	objet	Permet de connaître à partir de quelle classe l'objet a été créé
__dict__	objet	Renvoie un dictionnaire des méthodes et attributs disponibles
__doc__	classe et objet	Renvoie la "docstring" de la classe interrogée

Les méthodes spéciales

Les méthodes spéciales (entourés de double underscore) permettent de définir le comportement d'une classe. Celles-ci peuvent être redéfinies par la classe.

Nom	Rôle
__init__(self[, ...])	Permet l'initialisation des champs corrects lors de la création de l'objet.
__repr__(self)	Envoie la représentation "officielle" en chaîne de caractères d'un objet. Cette fonction est principalement utilisée à fins de débogage.
__str__(self)	Envoie la représentation "informelle" en chaîne de caractères d'un objet. Elle est appelée nativement par les fonctions "str(...)", "format()" et "print()".

https://docs.python.org/fr/3.7/reference/datamodel.html#basic-customization

Les méthodes spéciales

Elles permettent également de redéfinir les opérateurs de comparaison.

Nom	Opérateur
__eq__(self, other)	==
__ne__(self, other)	!=

Nom	Opérateur
__lt__(self, other)	<
__le__(self, other)	<=

Nom	Opérateur
__gt__(self, other)	>
__ge__(self, other)	>=

Par défaut, il existe une relation implicite entre les opérateurs d'égalité stricte.

L'opérateur "__ne__()" délègue à "__eq__()" et renvoie le résultat inverse.

Héritage

Concept de l'héritage

Une classe peut hériter d'une autre classe. Cela permet de créer de nouvelles classes, mais avec une base existante.

- Une classe qui hérite d'une autre classe est appelée sous-classe, classe enfant, classe fille ou encore classe dérivée.
- Une classe dont hérite(nt) une ou plusieurs classes est appelée super-classe ou classe mère.

L'objectif de cette hiérarchie logique est de permettre à la classe enfant de récupérer toutes les méthodes et les attributs de la classe mère.

L'héritage en Python

Pour définir un héritage, il est nécessaire de le définir sur la classe enfant.

```
class ClasseMere:
    pass

class ClasseEnfant(ClasseMere):
    pass
```

La méthode « issubclass » permet de tester la hiérarchie d'héritage entre les types.

```
test = issubclass(ClasseEnfant, ClasseMere)
```

La méthode « super() »

La méthode « super() » permet d'utiliser les propriétés et les méthodes de la classe mère au sein de la classe enfant.

Elle permet en autre de faire référence à la méthode « __init__ » de la classe mère au sein de l'enfant.

```
class ClasseMere:

    def __init__(self, msg):
        self.__msg = msg


class ClasseEnfant(ClasseMere):

    def __init__(self, msg, val):
        super().__init__(msg)
        self.__val = val


o1 = ClasseMere("Hello")
o2 = ClasseEnfant("Hi", 42)
```

Redéfinition de méthodes

Les méthodes de la classe mère peuvent être redéfinies par les enfants de la classe.

Cela permet de définir un comportement spécialisé pour une classe enfant.

```
class ClasseMere:

    def ma_methode(self):
        print("Méthode de la classe mère")


class ClasseEnfant(ClasseMere):

    def ma_methode(self):
        print("Méthode redéfinie dans la classe fille")
```

Exemple de mise en place (1/2)

La classe mère

```
class Animal:

    def __init__(self, nom):
        self.__nom = nom

    @property
    def nom(self):
        return self.__nom

    @nom.setter
    def nom(self, value):
        self.__nom = value

    def dormir(self):
        print(self.nom, "dort!")
```

La classe enfant

```
class Chat(Animal):

    def __init__(self, nom, couleur):
        super().__init__(nom)
        self.__couleur = couleur

    @property
    def couleur(self):
        return self.__couleur

    @couleur.setter
    def couleur(self, value):
        self.__couleur = value

    def ronronner(self):
        print(self.nom, "ronronne!")
```

Exemple de mise en place (2/2)

```
# Création d'un object "Chat"
c = Chat("Le chat", "Noir")

# Utilisation des propriétés
msg = c.nom + " est de couleur " + c.couleur
print(msg)

# Utilisation des méthodes
c.ronronner()
c.dormir()
```

Le polymorphisme

Le concept du polymorphisme signifie qu'un objet peut avoir plusieurs types.

La méthode « isinstance » permet de tester les types d'un objet.

```
# Création d'un object "Chat"
c = Chat("Le chat", "Noir")

# Possede le type "Chat"
test1 = isinstance(c, Chat)          # True

# Possede le type "Animal"
test2 = isinstance(c, Animal)        # True

# Ne possede pas le type "Voiture"
test3 = isinstance(c, Voiture)       # False
```

L'héritage multiple

Le langage python permet la mise en place d'héritage multiple (une classe héritant de plusieurs classes).

```
class MereA:
    pass

class MereB:
    pass


class Enfant(MereA, MereB):
    pass
```

L'héritage multiple

Cependant, cela peut poser un problème :

Lorsque plusieurs classes mères définissent la même méthode :

- Quelle méthode sera utilisée par défaut (sans utiliser la redéfinition) ?
- Comment cibler la méthode d'un des parents au sein de la classe Enfant ?

C'est d'ailleurs pour ces raisons que peu de langages permettent la mise en place d'héritage multiple. Par exemple, les langages "C#" et "Java" ne le permettent pas.

L'héritage multiple

Pour éviter ses problèmes, en cas de conflit, Python redéfini les méthodes.

L'attribut de classe « __mro__ » permet d'obtenir un tuple avec l'ordre de redéfinition.

```
(<class '__main__.Enfant'>, <class '__main__.MereA'>, <class '__main__.MereB'>, <class 'object'>)
```

La méthode « super() » accède aux méthodes de la classe parente sur base de cet ordre.

Pour cibler la méthode d'une classe particulière, il est possible d'utiliser le nom de la classe à la place de la méthode « super() ».

Exemple d'héritage multiple

```python
class MereA:

    def __init__(self, val_a):
        self.val_a = val_a

    def methode(self):
        print("Elem A")


class MereB:

    def __init__(self, val_b):
        self.val_b = val_b

    def methode(self):
        print("Elem B")
```

```python
class Enfant(MereA, MereB):

    def __init__(self, val_a, val_b):
        MereA.__init__(self, val_a)
        MereB.__init__(self, val_b)

    def methode_1(self):
        super().methode()    # Elem A

    def methode_2(self):
        MereA.methode(self) # Elem A

    def methode_3(self):
        MereB.methode(self) # Elem B
```

Les classes abstraites

L'objectif

Lors de la mise en place de l'héritage, il est possible que certaines fonctionnalités de la classe parente ne soit pas suffisamment complète pour permettre de l'implémenter.

Le concept de méthode abstraite permet d'ajouter la signature de la méthode à la classe parente et de déléguer l'implémentation aux classes enfants.

Attention, une classe ne peut pas être instanciée si elle possède une méthode abstraite.

En python, le module « ABC » permet de fournir l'infrastructure nécessaire pour mettre en place des méthodes abstraites au sein de la classe.

Le module « ABC »

Le module « ABC » (Abstract Base Class) permet de marquer les méthodes ou les propriétés comme abstraites à l'aide du décorateur « @abc.abstractmethod »

```
from abc import ABC, abstractmethod

class ClasseAbstraite(ABC):

    @abstractmethod
    def methode_abstraite(self, msg):
        pass

class ClasseEnfant(ClasseAbstraite):

    def methode_abstraite(self, msg):
        print("Afficher : " + msg)
```

Exemple de propriétés abstraites

Propriété à l'aide des méthodes

```
from abc import ABC, abstractmethod

class ClasseAbstraite(ABC):

    @abstractmethod
    def _get_x(self):
        pass

    @abstractmethod
    def _set_x(self, value):
        pass

    x = property(_get_x, _set_x)
```

Propriété à l'aide des décorateurs

```
from abc import ABC, abstractmethod

class ClasseAbstraite(ABC):

    @property
    @abstractmethod
    def msg(self):
        pass

    @msg.setter
    @abstractmethod
    def msg(self, value):
        pass
```

Les membres statiques

L'objectif

Les membres statiques d'une classe sont des éléments qui appartiennent au type lui-même plutôt qu'à une instance spécifique.

Nous accédons aux membres statiques en utilisant le nom du type lui-même.

⚠️ Contrairement à d'autres langages (telle que le C# et le Java), le python permet d'accéder aux membres statiques depuis une instance de la classe !

Attributs statiques

Pour définir un attribut statique, il faut l'ajouter à la classe et l'initialiser.

Particularité du langage python :

- L'attribut statique est accessible depuis l'instance de la classe
- L'instance peut également dissimuler la valeur statique à l'aide d'une valeur d'instance.

```
class MaClasse:
    x = 1337


print(MaClasse.x)       # 1337

MaClasse.x = 42
print(MaClasse.x)       # 42

c = MaClasse()
print(c.x)              # 42
print(c.__dict__)       # {}

c.x = 13
print(c.x)              # 13
print(c.__dict__)       # {'x': 13}

print(MaClasse.x)       # 42
```

Méthodes statiques

Pour définir une méthode statique, il faut lui ajouter le décorateur « *@staticmethod* ».

Contrairement aux méthodes d'instance, celles-ci ne possèdent pas l'argument « *self* ».

```
class MaClasse:

    @staticmethod
    def methode_static():
        print("Hello World !")


MaClasse.methode_static()
```

Les interfaces

L'objectif

L'objectif des interfaces est de décrire un ensemble de fonctionnalités liées pouvant appartenir à n'importe quelle classe.

C'est un « contrat » qu'on peut demander à une classe de respecter.

Le langage python ne supportant nativement pas les interfaces.
Il est possible d'utiliser 2 solutions à la place :

- Le Duck Typing
- Simuler leurs fonctionnements

Le Duck Typing

Le nom de ce système de typage est une référence au test du canard :

> *« Si je vois un oiseau qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard »*

Concrètement : si un objet définit les méthodes d'un type, c'est qu'il est de se type.

Exemple :
Un objet est une « collection » s'il définit les méthodes : __len__, __contains__, __iter__.

Simuler les interfaces

En python, il est possible de simuler des interfaces à l'aide de plusieurs mécanismes :

- Les classes abstraites
- L'héritage multiple
- Le polymorphisme

Simuler les interfaces

Pour mettre en place une *interface* en python, il faut :

- Créer une "*interface*" à l'aide d'une classe abstraite, celle-ci devra uniquement posséder les signatures des méthodes.
- Mettre en place l'héritage entre "*l'interface*" et notre classe.
- Implémenter les méthodes de "*l'interface*" au sein de notre classe.

Grâce au polymorphisme, il est possible de vérifier si un objet implémente une interface à l'aide de la méthode « isinstance(mon_objet, mon_interface) »

Merci pour votre attention.