Introduction à la programmation orientée objet

Comment créer ses propres structures en Python ?

Stéphane Guinard

La POO : kézako ?

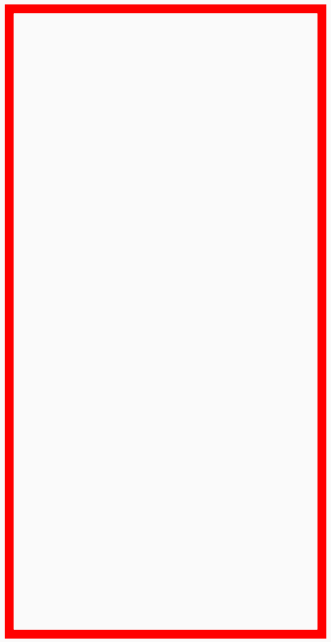
Qu'est-ce que c'est ?

La programmation orientée objet *consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes.*

Wikipédia

Pour quoi faire ?

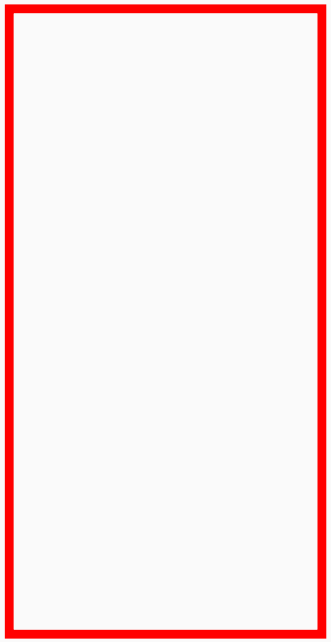
Qu'est-ce qu'un rectangle ?



Pour quoi faire ?

Qu'est-ce qu'un rectangle ?

- Objet dans l'espace
- Possède 4 sommets
- Chaque sommet a ses propres coordonnées `X`, `Y`, `Z`
- Possède 4 côtés
- Côtés sont 2 à 2 égaux
- Côtés perpendiculaires



La POO c'est pas si nouveau

Quand on ajoute des éléments à une liste :

```
>>> l.append("element")
```

Quand on compte le nombre d'éléments (e.g. "a") dans une chaîne de caractères :

```
>>> s.count(a)
```

Quand on ouvre un fichier :

```
>>> fichier = open("fichier.txt","r")
```

Principes de la POO

Il y a 4 principes fondamentaux dans la POO :

- Héritage
- Polymorphisme
- Encapsulation
- Abstraction

Ils reposent tous sur la notion de **classe**.

Notions de classe

Déclaration d'une classe

- Utilisation du mot-clé 'class'

```
class MaClasse(object):
    ...
    ...
```

- Convention :
  - 1ère lettre des mots en majuscule
  - pas d'espace, tiret, underscore...
  - `MonNomDeClasse`

Instanciation d'un objet

L'objet créé est une instance de classe.

On en crée autant que l'on veut.

```
>>> mon_objet1 = MaClasse()
>>> mon_objet2 = MaClasse()
>>> mon_objet3 = MaClasse()

>>> print(mon_objet1 is mon_objet2)
False
```

Les méthodes

Une méthode est une fonction définie à l'intérieur d'une classe.

Pour l'utiliser **on fait précéder le nom de la méthode du nom de l'objet**.

```
>>> class MaClasse(object):
        def une_methode(param):
            print("Une méthode qui fait quelque
                chose")

>>> mon_objet = MaClasse()
>>> mon_objet.une_methode()
Une méthode qui fait quelque chose
```

Les méthodes

Instance appelant la méthode = 1er paramètre de la méthode

- nommé `self` par convention
- spécificité de Python

```
>>> class MaClasse(object):
        def une_methode(self):
            print(self)

>>> mon_objet = MaClasse()
>>> print(mon_objet)
<__main__.DescriptionDeLObjet object at 0
    x059F0190>
>>> mon_objet.une_methode()
<__main__.DescriptionDeLObjet object at 0
    x059F0190>
```

Les attributs

Un attribut est une variable définie à l'intérieur d'une classe. Pour y accéder :

- on fait précéder le nom de l'attribut du nom de l'instance
- ou de `self` si on est dans la classe

```
>>> class MaClasse(object):
        def afficher(self):
            print(self.un_attribut)

>>> mon_objet = MaClasse()
>>> mon_objet.un_attribut = 42
>>> mon_objet.afficher()
42
>>> un_attribut   # n'exsite pas en dehors de la
    classe
NameError: name 'un_attribut' is not defined
```

Le constructeur

- C'est une méthode appelée lors de la création d'objets.
- Elle permet d'initialiser les valeurs des attributs.
- Nommé `__init__`.

```
>>> class MaClasse(object):
        def __init__(self):
            self.un_attribut = "valeur initiale"
            print("Objet créé. Attributs
                initialisés.")

>>> mon_objet = MaClasse()
Objet créé. Attributs initialisés.
>>> mon_objet.un_attribut
"valeur initiale"
```

Le constructeur

Possibilité de passer des paramètres pour personnaliser les objets.

```
>>> class MaClasse(object):
        def __init__(self, valeur):
            self.un_attribut = valeur

>>> mon_objet = MaClasse("valeur personnalisée")
>>> mon_objet.un_attribut
"valeur personnalisée"
```

Le constructeur

Le constructeur peut aussi avoir des valeurs par défaut.

```
>>> class MaClasse(object):
        def __init__(self, valeur="valeur par dé
            faut"):
            self.un_attribut = valeur

>>> mon_objet1 = MaClasse()
>>> mon_objet1.un_attribut
"valeur par défaut"

>>> mon_objet2 = MaClasse("valeur personnalisée")
>>> mon_objet2.un_attribut
"valeur personnalisée"
```

Méthodes spéciales

Nous appelerons méthode spéciale une méthode exécutée sans qu'on ai besoin de l'appeler explicitement.

- `__init__()` en est une.
- Leur nom commence et termine par `__`.
- Utile pour personnaliser le comportement d'un objet.

Méthodes spéciales

Nous appelerons méthode spéciale une méthode exécutée sans qu'on ai besoin de l'appeler explicitement.

- `__init__()` en est une.
- Leur nom commence et termine par `__`.
- Utile pour personnaliser le comportement d'un objet.

Remarque

Pour chaque méthode spéciale, Python a un comportement par défaut. Vous n'avez donc pas besoin de toutes les redéfinir. Il faut se concentrer sur les méthodes spéciales **dont on a besoin** et ou le comportement par défaut de Python ne convient pas.

Méthodes spéciales

```
>>> class MaClasse(object):
        def __init__(self, valeur):
            self.un_attribut = valeur

>>> mon_objet = MaClasse("17")
>>> print(mon_objet)
<__main__.MaClasse object at 0x051D5DF0>
```

Méthodes spéciales

```
>>> class MaClasse(object):
        def __init__(self, valeur):
            self.un_attribut = valeur
        def __str__(self):
                print("Instance de MaClasse (
                    valeur de un_attribut = {})".
                    format(self.un_attribut)

>>> mon_objet = MaClasse("17")
>>> print(mon_objet)
Instance de MaClasse (valeur de un_attribut = 17)
```

Méthodes spéciales

Quelques méthodes spéciales :

- `__str__()` : appelée suite à un : `print(objet)`.
- `__add__()` : pour pouvoir écrire : `objet1 + objet2`.
- `__eq__()` : permet de comparer deux objets : `objet1 == objet2`.
- `__ne__()` : permet de tester si deux objets sont différents : `objet1 != objet2`.
- `__len__()` : pour pouvoir calculer la longueur d'un objet : `len(objet)`.
- `__del__()` : appelé à la suppression de l'objet

Portée des objets

À quel moment la fonction `__del__()` (*destructeur*) est-elle appelée ?

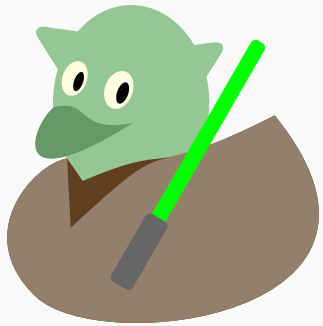
Portée des objets

À quel moment la fonction `__del__()` (*destructeur*) est-elle appelée ?

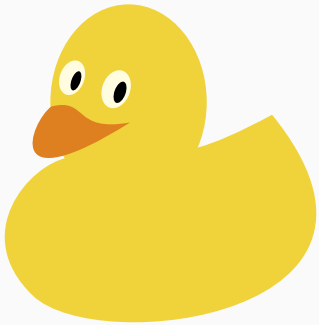
- quand on l'appelle explicitement
- lorsque le contexte local est détruit (par exemple à la fin d'une fonction)
- à la fin du programme

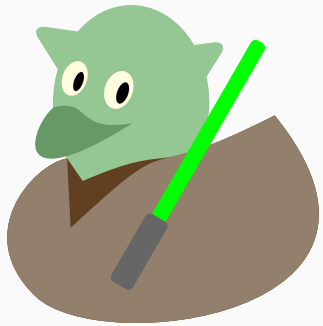
L’héritage

L'héritage

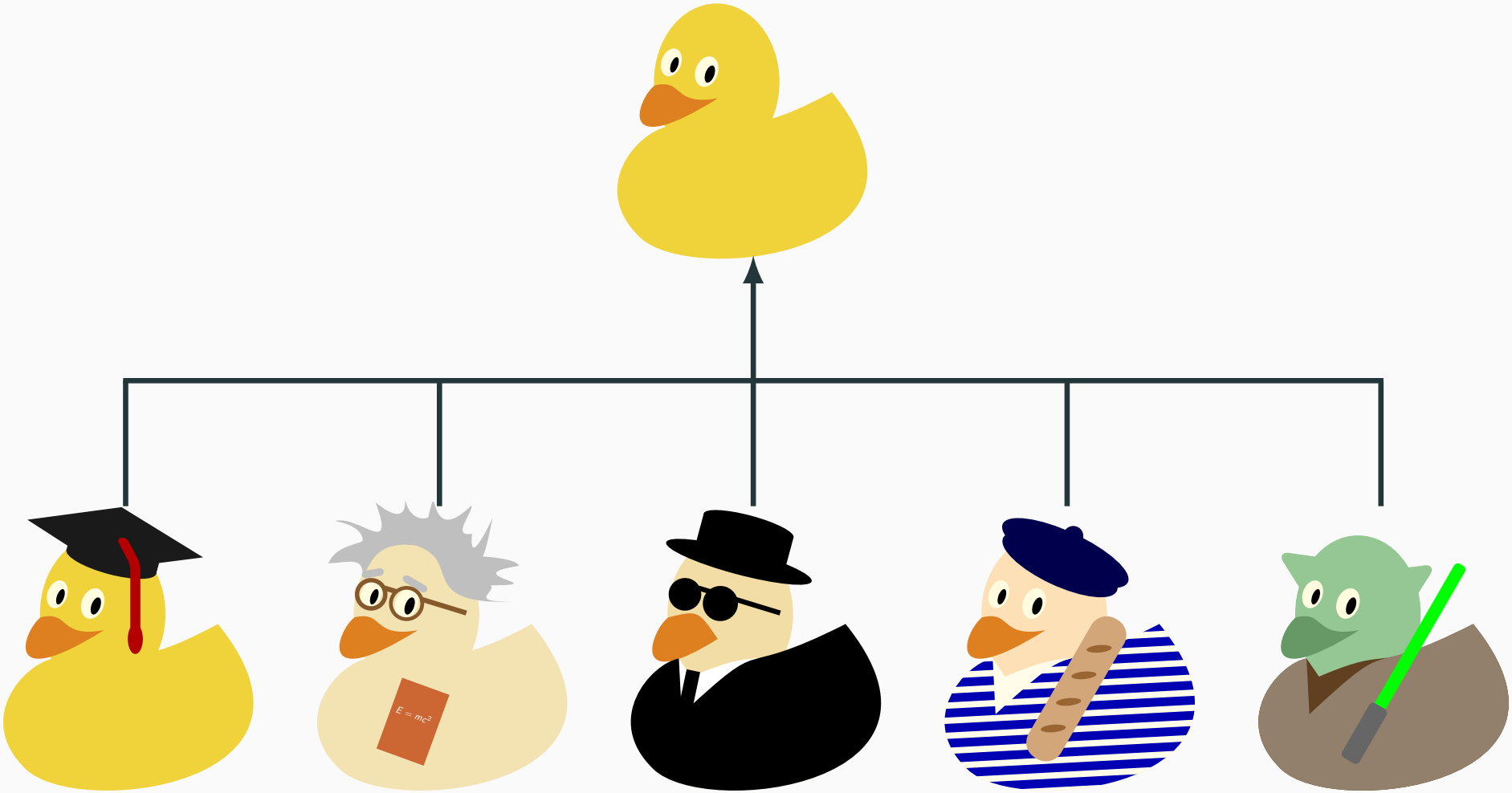


L'héritage





L'héritage



L'héritage

- Principe permettant de créer une classe à partir d'une autre.
- Nom de la classe mère entre paranthèses lors de la définition.
- Méthodes et attributs de la classe mère transmis à la classe fille.

```
class ClasseMere(object):
    ...
    ...

class ClasseFille(ClasseMere):
    ...
    ...
```

Remarque : toutes les classes héritent d'une classe `object`.

L'héritage

```
>>> class ClasseMere(object):
        def une_methode(self):
            print("Je suis dans la classe mère")

>>> class ClasseFille(ClasseMere):
        pass

>>> ClasseFille().une_methode()
Je suis dans la classe mère
```

L'héritage

Un objet de la classe fille peut appeler une méthode ou un attribut de la classe mère :

```
>>> class Canard(object):
        def __init__(self):
            self.couleur = "jaune"

>>> class CanardYoda(Canard):
        def __init__(self, couleur):
            Canard.__init__(self)
            self.couleur = couleur  # Appelle l'attribut
                couleur défini dans la classe mère

>>> Canard.couleur
'jaune'
>>> CanardYoda("vert").couleur
'vert'
```

Le polymorphisme

Le polymorphisme

Principe permettant de donner plusieurs définitions à une méthode.

En programmation orientée objet, cela se traduit par :

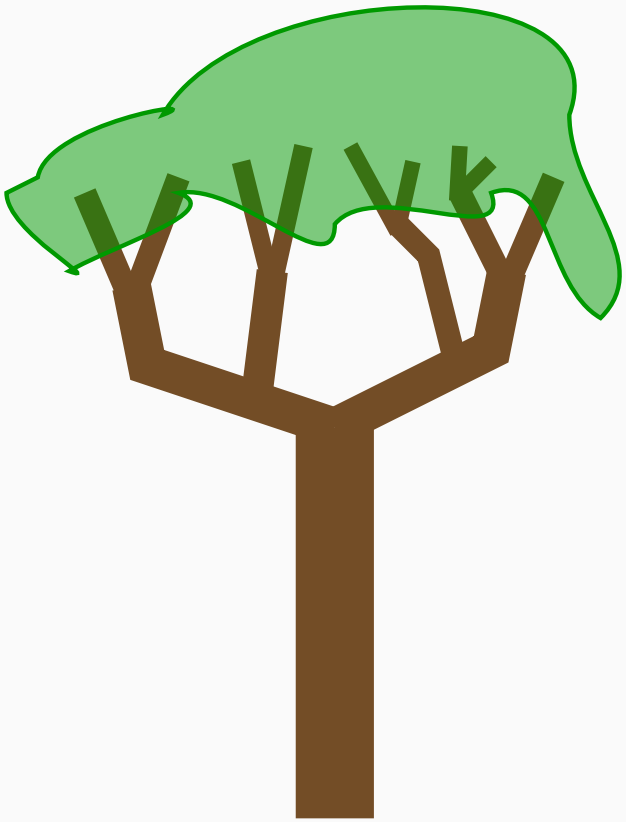
- la possibilité de redéfinir une méthode d'une classe mère dans une classe fille
- le fait que Python se charge de trouver la bonne méthode à utiliser

Le polymorphisme

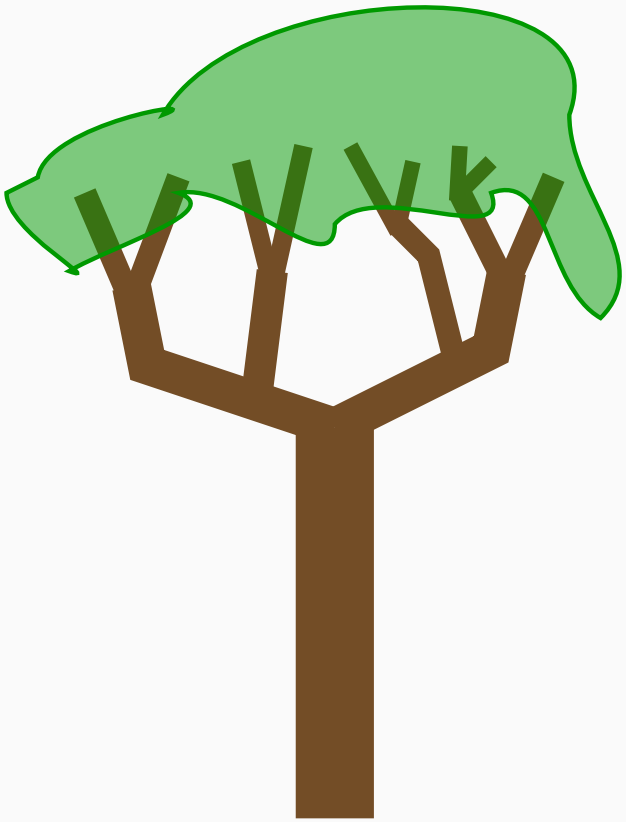
```
>>> class ClasseMere(object):
        def une_methode(self):
            print("Je suis dans la classe mère")

>>> class ClasseFille(ClasseMere):
        def une_methode(self):
            print("Je suis dans la classe fille")

>>> ClasseMere().une_methode()
Je suis dans la classe mère
>>> ClasseFille().une_methode()
Je suis dans la classe fille
```

L'encapsulation

L'encapsulation



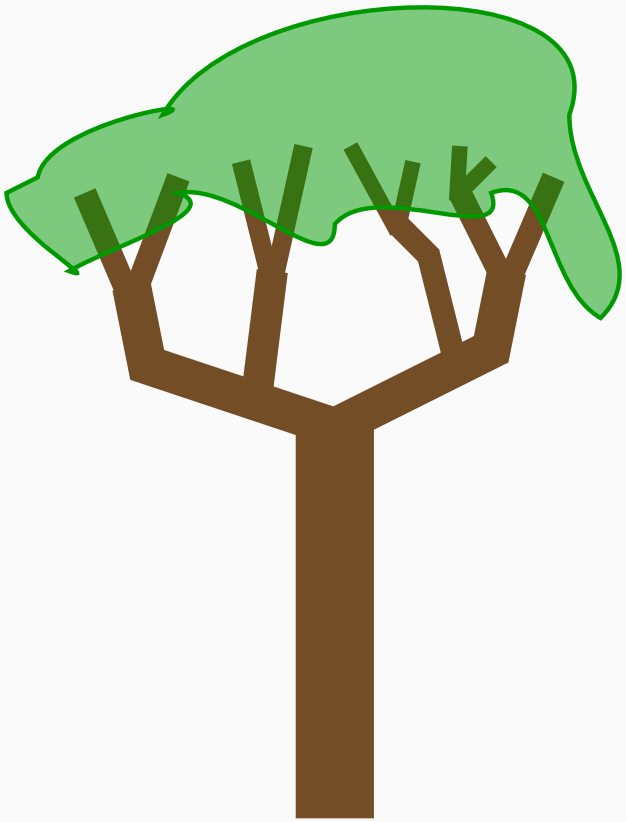
L'encapsulation



Un arbre est composé de :

- un tronc
- des branches
- des brindilles
- des feuilles

L'encapsulation



Un arbre est composé de :

- un tronc
- des branches
- des brindilles
- des feuilles

Quand on parle d'un arbre, on sait qu'il contient toutes ces parties mais on ne le détaille pas.

L'encapsulation

Principe visant à cacher les détails de l'implémentation à l'utilisateur : **les attributs de chaque classe doivent être privés**, c'est-à-dire inaccessibles en dehors de la classe.

En Python cette notion n'existe pas : **tout est public**. Par convention on ajoutera un " _ " avant chaque attribut pour faire comme s'il était privé.

L'encapsulation

```
>>> class MaClasse(object):
        def __init__(self, valeur1, valeur2):
            self.attribut_public = valeur1
            self._attribut_prive = valeur2
```

Lecture/écriture d'attributs

Comment lire/écrire les valeurs des attributs privés ?

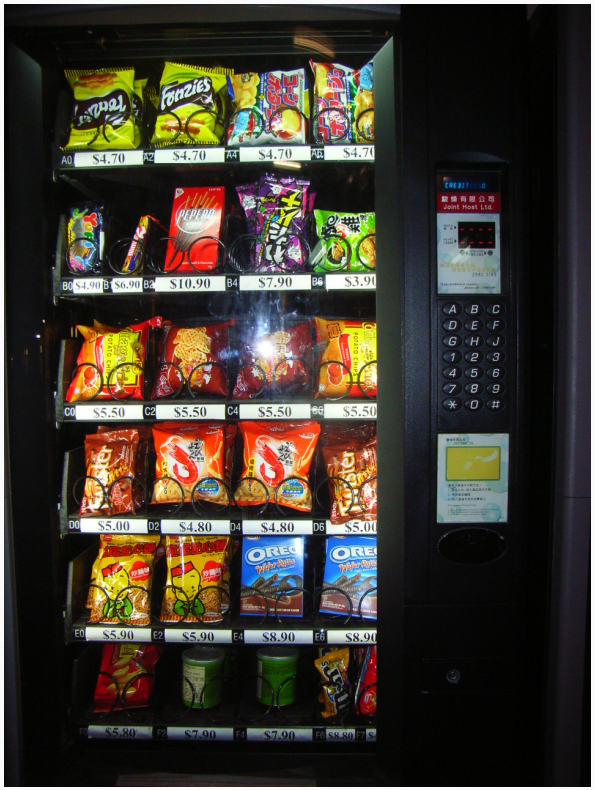
On utilise des méthodes spécifiques. On les appellent des **getter** (pour lire une valeur) et des **setter** (pour changer une valeur).

Lecture/écriture d'attributs

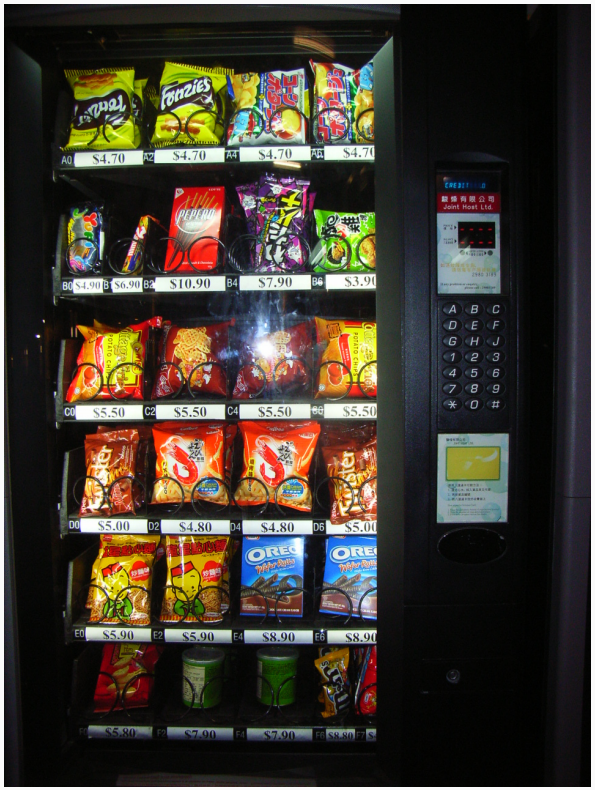
```
>>> class Canard(object):
        def __init__(self, couleur, taille):
            self._couleur = couleur
            self._taille = taille

        def get_couleur(self):
            return self._couleur

        def set_couleur(self, couleur):
            self._couleur = couleur

>>> canard = Canard("jaune", 5)
>>> canard.set_couleur("vert")
>>> canard.get_couleur()
'vert'
```

L’abstraction

Idée générale



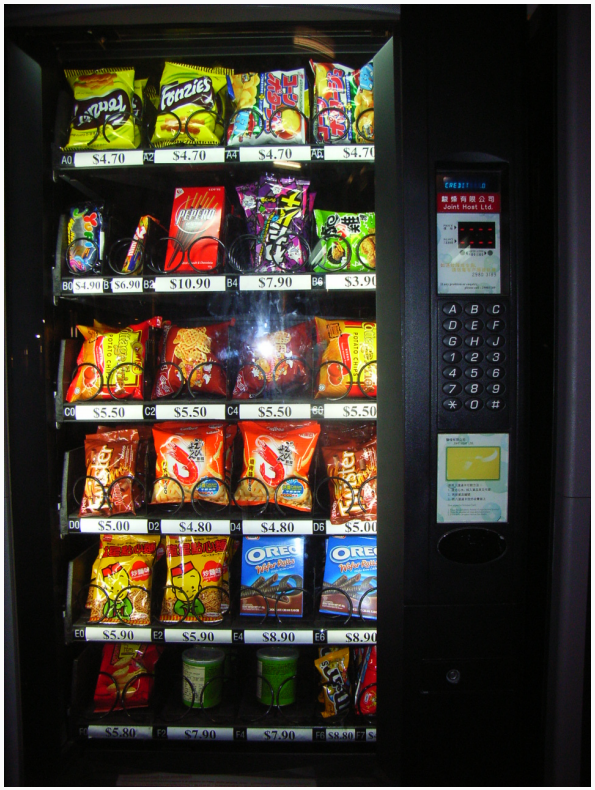
Idée générale



Ce que veut l'utilisateur :

- Appuyer sur un bouton
- Mettre de l'argent
- Récupérer son soda/snack

Idée générale



Ce que veut l'utilisateur :

- Appuyer sur un bouton
- Mettre de l'argent
- Récupérer son soda/snack

Ce que l'utilisateur ne voit pas :

- Interprétation de la commande
- Vérification du paiement
- Donner le soda/snack

Les classes abstraites

Une classe abstraite est une classe qui ne peut être instanciée.

C'est une notion qui est difficile à mettre en œuvre en Python.

Remarque : on peut lever une exception dans le constructeur.

Les classes abstraites

```
class MaClasseAbstraite(object):
    def __init__(self, param1, param2...):
        raise NotImplementedError

    def methode1(self, param...):
        # methode1 est une méthode abstraite
        raise NotImplementedError

    def methode2(self, param...):
        # methode2 est une méthode concrète : on
            l'implémente ici
```

Les classes abstraites

```
class Animal(object):
    def __init__(self):
        raise NotImplementedError

    def manger():
        # methode abstraite
        raise NotImplementedError

class Canard(Animal):
    def __init__(self, couleur):
        self._couleur = couleur

    def manger(ingredient):
        # methode concrète
        ...
```

Différences abstraction/encapsulation

Quelles sont les différences entre abstraction et encapsulation ?

Différences abstraction/encapsulation

Quelles sont les différences entre abstraction et encapsulation ?

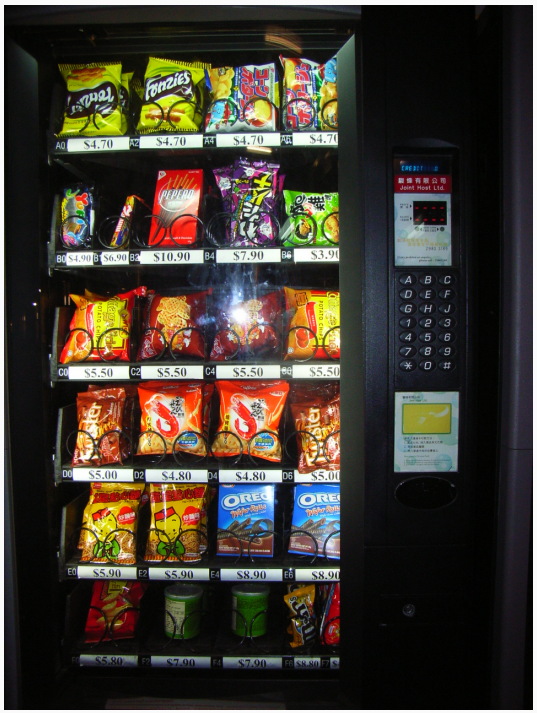


Figure 1: Abstraction

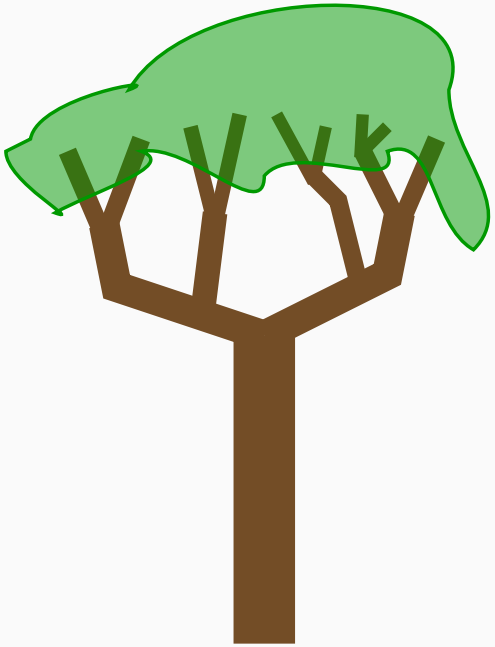


Figure 2: Encapsulation

Différences abstraction/encapsulation

Abstraction	Encapsulation
• Généralisation	• Structuration
• Cache les données / structures pour mettre en avant les idées	• Cache les détails de l'implémentation
• Se concentrer sur ce que fait un objet et pas sur comment il le fait	• cache les détails / méchanismes utilisés par l'objet
• Design	• Implémentation