

Programmation 3

CM séance 1 : Introduction

Pacôme Perrotin, repris de Manon Scholivet, adapté des cours de Florian Bridoux (et d'une partie de ceux d'Arnaud Labourel)

6 septembre 2021

Description de l'UE

Les Objets

Héritage et Composition

Comment gérer un projet

Que savez-vous ?/Quelques Rappels...

UE SMI5U05 - "Informatique S5" : Programmation 3

Responsable : Pacôme Perrotin

Mail : pacome.perrotin@lis-lab.fr

Objectifs

- Approfondir la POO (Programmation Orientée Objet) en Python 3
- Algorithmique
- Apprendre à gérer un projet

MCC : DM = projet en binômes, ET = examen terminal 2h,
NF = note finale

- Session 1 : $NF = \text{Max}(0.5*DM + 0.5*ET, ET)$
- Session 2 : $NF = ET$

Description de l'UE

Les Objets

Un objet c'est quoi ?

Les objets en python

Héritage et Composition

Comment gérer un projet

Que savez-vous ?/Quelques Rappels...

Un objet c'est ..

Un cahier, un stylo, ...

Un objet est défini par :

- Ses **attributs/propriétés** (le stylo a de l'encre noir, il a un bouchon, il est à plume ou non, ...)
- Les actions qu'il peut réaliser, ce sont ses **méthodes** (un cahier peut se remplir de texte, tourner une page, ...)

Un objet est un **concept**

On parle d'**instance** lorsqu'on affecte une vraie valeur a un objet

Instances possibles de stylo



Instances possibles de stylo



Instances possibles de stylo



Les objets en python

Le mot clef *class* permet de déclarer des objets avec des attributs et des méthodes.

```
class NomDeLaClasse :  
    def __init__(self, paramètres_création) :  
        corps du constructeur  
  
    def __del__(self) :  
        corps du destructeur optionnel  
  
    def autre_methode(self, autres_paramètres) :  
        corps méthode
```

Règles de bon usage (PEP 8) :

- NomDeLaClasse : en CamelCase, première lettre en majuscule
- instances, attributs et méthodes : en snake_case, première lettre minuscule

Les objets en python

```
class NomDeLaClasse :  
    def __init__(self, paramètres_création) :  
        corps du constructeur  
  
    def __del__(self) :  
        corps du destructeur optionnel  
  
    def autre_methode(self, autres_paramètres) :  
        corps méthode
```

Le paramètre self est passé en premier à chaque méthode, il désigne l'instance courante ("this" en java et en C++).

Création d'une instance :

```
instance = NomDeLaClasse(paramètres_creation)
```

La fonction __init__() est appelée lors de la création de l'instance.

Appel d'une méthode : instance.autre_methode(autres_paramètres)

- Quels sont les objets nécessaires à la résolution du problème ?
⇒ décomposition du problème en objets
- À quels modèles des objets correspondent-il ?
⇒ Quelles sont les classes ?
- Quels sont les fonctionnalités/opérations dont on doit/veut pouvoir disposer sur ces objets ?
⇒ Quelles sont les méthodes des classes ?
- Quelle est la structure des données de l'objet ?
⇒ Quelles sont les attributs des classes ?

Exemple de problèmes

- Un catalogue regroupe des articles, il permet de trouver un article à partir de sa référence.
- Un article est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- Une commande est créée pour un client et un catalogue donnés, on peut ajouter des articles à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- Un client peut créer une commande pour un catalogue et commander dans cette commande des articles à partir de leurs références

Toutes les variables ne doivent pas toujours être accessible partout. Certains attributs et certaines méthodes vont rester internes à la classe. Ce sont des **attributs/méthodes privé(e)s**. Une variable est privée si son nom est préfixé par un `"_"`.

En POO, cette notion de publique/privée est essentielle pour coder proprement. Python est un des rares langages qui ne force pas les méthodes privées à rester internes à la classe, mais qui indique seulement aux autres utilisateurs qu'il faut éviter de modifier cette variable depuis l'extérieur.

```
class Point :
    def __init__(self, px, py) :
        self.x = px
        self.y = py
    def _is_origin(self):
        if(self.x == 0 and self.y == 0):
            return True
    def afficher(self) :
        if(self._is_origin()):
            print("This point is the origin")
        else:
            print(self.x, self.y)

p1 = Point(3,5)
p2 = Point(0,0)

p1.afficher()
p2.afficher()
```

Description de l'UE

Les Objets

Héritage et Composition

Comment gérer un projet

Que savez-vous ?/Quelques Rappels...

Composons !

Une classe peut être un attribut d'une autre classe. C'est un **lien de composition**.

```
class Volant :
    def __init__(self, diametre, marque):
        self.diametre = diametre
        self.marque = marque
    def __str__(self):
        return "Volant de diametre " + str(self.diametre)
        + " de marque " + str(self.marque)

class Voiture :
    def __init__(self, volant, couleur):
        self.volant = volant
        self.couleur = couleur
    def __str__(self):
        return "Voiture de couleur " + self.couleur + "
        utilisant un " + str(self.volant)
```

Volant est une classe **agrégée**, Voiture est la classe **composite**. La classe agrégée dépend de la classe composite : lorsqu'une instance de Voiture est détruite, l'instance de son Volant l'est aussi. L'instance de la classe Volant est utilisée exclusivement pour l'instance de Voiture courante.

Et l'Héritage, c'est quoi ? (quand on n'est pas chez le notaire)

Consiste à fabriquer une nouvelle classe (la fille, ou classe dérivée) à partir d'une classe existante (la mère) :

- la fille "raffine" ou "spécialise" la mère
- la mère "généralise" la fille.

La classe fille hérite les méthodes et les attributs de la classe mère.

⇒ Le constructeur étant une méthode, il est également hérité. Si la classe fille a son propre constructeur, celui de la mère n'est plus appelé ; on peut l'appeler explicitement avec `super()`

Exemple d'héritage

Pour qu'une classe hérite d'une autre, au moment de la déclaration de la classe, on donne en argument la classe mère :

`class Carré(Rectangle) :`

```
class Vehicule :
    def __init__(self, couleur, marque, vitesse):
        self.couleur = couleur
        self.marque = marque
        self.vitesse = vitesse

class Voiture(Vehicule) :
    def __init__(self, couleur, marque, vitesse):
        super().__init__(couleur, marque, vitesse)
    def __str__(self):
        return "Voiture de couleur " + self.couleur + " de marque " +
self.marque + " roulant a une vitesse de " + str(self.vitesse) + "km/h"

class Bateau(Vehicule) :
    def __init__(self, couleur, marque, vitesse):
        super().__init__(couleur, marque, vitesse)
    def __str__(self):
        return "Bateau de couleur " + self.couleur + " de marque " +
self.marque + " naviguant a une vitesse de " + str(self.vitesse) + " noeuds"
```

Surcharge

Certaines méthodes ou certains attributs de la classe mère vont avoir besoin d'être redéfinie dans la classe fille : on appelle ça la **surcharge**.

```
class Rectangle :
    def __init__(self, longueur, largeur):
        self.long = longueur
        self.larg = largeur
    def aire(self) :
        return self.long * self.larg
    def __str__(self):
        return "Rectangle de taille " + str(self.long) + " par " + str(self.larg)

class Carré(Rectangle) :
    def __init__(self, côté, couleur):
        super().__init__(côté, côté)
        self.couleur = couleur
        self.c = côté
    def __str__(self):
        return "Carré " + self.couleur + " de côté " + str(self.c)
```

Composition vs Héritage

Héritage : Une classe héritée **est** une autre classe plus générale (un Carré *est* un Rectangle)

Composition : Une classe composite **a/utilise** une autre classe (une classe Voiture *a* un Volant)

Description de l'UE

Les Objets

Héritage et Composition

Comment gérer un projet

Tests unitaires

Gestionnaires de version

Que savez-vous ?/Quelques Rappels...

Règle

Un code non-testé n'a aucune valeur

Corollaire

Tout code doit être testé

Différents types de tests

- **Tests unitaires** : Tester les différentes parties d'un programme indépendamment les unes des autres
- **Tests de non régression** : vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être

Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
 - cas normaux
 - cas limites
 - cas anormaux

L'instruction assert

```
def division(nominateur, denominateur):  
    assert denominateur != 0 # le denominateur ne doit pas être égal à 0  
    return nominateur / denominateur
```

Cette instruction permet de vérifier des conditions, et de lever une exception en cas de non-respect.

Exceptions : assert

L'instruction `assert` lèvera une exception de type `AssertionError`. Il est possible de lever d'autres exceptions, avec l'instruction `raise`. Cela permet également de donner un message d'erreur.

```
def division(numerateur, denominateur):  
    if(denominateur == 0): # le denominateur ne doit pas être égal à 0  
        raise ValueError("Le denominateur ne doit pas être 0")  
    return numerateur / denominateur
```

Exceptions : try...except

Un bloc d'instruction `try...except` permet de capturer une exception, et de proposer un code alternatif en cas de problème.

```
variable_bonjour = "bonjour"

try: # On essaye de convertir cette chaîne de caractères en entier
    variable_bonjour = int(variable_bonjour)
except:
    print("Erreur, bonjour n'est pas un nombre")
```

Principe

- Le code d'un projet est stocké dans un serveur
- Les développeurs soumettent des modifications avec des commentaires à chaque fois
- Le serveur conserve l'historique des mises à jour

Pourquoi la gestion de version ?

- Pour travailler de manière harmonieuse en équipe sans se marcher dessus
- Pour revenir en arrière en cas de problèmes
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

- Logiciel de gestion de version le plus populaire
- Serveur gratuit github
- Version libre de logiciel serveur : gitlab
- Gestion de version décentralisée : la gestion de version se fait aussi en local
- Les développeurs soumettent des modifications avec des commentaires à chaque fois
- Le serveur conserve l'historique des mises à jour

Utilisation de git

- Via l'IDE
- En ligne de commande : commande git

Exemples de commandes git

```
git clone adresse_projet
```

⇒ Clone un projet en local depuis un serveur

```
git add nom_de_fichier
```

⇒ Ajoute un fichier à la prochaine mise à jour.

```
git commit -m "commentaire"
```

⇒ Fait une mise à jour en local

```
git push
```

⇒ Pousse les mises à jour locales sur le serveur

```
git pull
```

⇒ Récupère les mises à jour du serveur en local

Description de l'UE

Les Objets

Héritage et Composition

Comment gérer un projet

Que savez-vous ?/Quelques Rappels...

Ce qu'on a vu aujourd'hui

- Rappels de Python
- Bases de Programmation Orientée Objet (de son petit nom, POO)
- Gestionnaires de fichier
- Tests unitaires