0.1 Un paradigme de programmation

Dans la vie réelle, un objet possède des *caractéristiques* et nous permet de réaliser des *actions*. Un chat par exemple possède une taille, une couleur, un nom, etc., ce sont ses caractéristiques. Il peut miauler, se déplacer etc., ce sont les actions qu'il peut réaliser.

Le concept d'objet en informatique s'inspire fortement de cette définition de la vie réelle : on va appeler "objet" un bloc cohérent de code qui possède ses propres variables ou **attributs** (équivalents des caractéristiques des objets de tous les jours) et fonctions ou **méthodes** (qui sont les actions que l'objet peut réaliser). Ces objets vont interagir entre eux et avec le monde exterieur.

Jusqu'ici, nous avons programmé en découpant les programmes en fonctions. La programmation orientée objet (POO) propose un autre **paradigme de programmation** où tout ce que vous avez appris va être utilisé.

Vous avez déjà utilisé des objets. Par exemple, lorsque vous utilisez l'instruction:

`L = [3, 1, 4, 1, 5]`,

vous créez une **instance** L qui est un objet de la classe `list`. L'instruction `liste.append()` utilise la **méthode** append() sur l'instance L.

En Python, *tout est objet*: `int, float, dict` etc...

Nous allons apprendre à créer de nouvelles classes d'objet.

C++ langage compilé créé en 1983 par *Bjarne Stroustrup* ajoute le paradigme objet au langage C.

0.2 Création d'une première classe

Les classes sont les principaux outils de la POO. En Python, on définit une classe en suivant la syntaxe `class NomClasse:` Il est préférable d'utiliser pour des noms de classes la convention dite **Camel Case**. Cette convention n'utilise pas le signe souligné _ pour séparer les mots. Le principe consiste à mettre en majuscule chaque lettre débutant un mot.

Exemple: Créons une classe `Point` définie à l'aide de ses coordonnées:

```
>>> class Point:
...     """
...     point géom. défini par:
...     - son nom
...     - son abscisse, son ordonnée.
...
...     """
...     pass
...
>>> p1 = Point()
>>> print(p1)
<__console__.Point object at 0x7f9210824d30>
>>> type(p1)
<class '__console__.Point'>
```

Lorsqu'on crée un objet à partir d'une classe avec l'instruction `p1 = Point()`, on dit que l'on instancie une classe: on crée une nouvelle instance de cette classe. Remarquons que le type de `p1` est `Point`, on a donc créé un nouveau type de donnée. La fonction native *print* ne fonctionne pas sur ce nouveau type d'objet: il faudra redéfinir son fonctionnement.

Bien sûr, cette classe ne fait pas grand chose. Ajoutons deux variables sur cette instance: son abscisse et son ordonnée.

0.2.1 Attributs d'instance

On utilise la notation pointée pour accéder aux attributs de l'objet que l'on peut aussi appeler variables d'instances:

```
>>>p1 = Point()
>>>p1.x = 4
>>>p1.y = 10
>>>print(p1.x)
4
```

Ces attributs sont **encapsulés** dans l'objet et sont donc distincts d'autres variables pouvant porter le même nom. Nous pourrions écrire: x = p1.x sans que cela prête à confusion.

Les objets créés peuvent être passés comme argument dans une fonction; vous l'avez déjà fait avec les objets `list, dict` etc. De plus, ces objets pourront être réutilisés dans d'autres classes: cela s'appelle la composition. Une classe `Vecteur` pourrait avoir besoin d'une classe `Point`.Voir Exercice.

Visibilité des attributs

La pratique décrite jusqu'ici permet de modifier les attributs des objets par la notation pointée, en dehors de la classe. Cela peut sembler être un inconvénient mineur mais ce n'est pas le cas car cela peut avoir des conséquences importantes en développement logiciel.

Modifier les attributs en dehors de la classe modifie l'interface, les objets ne sont plus représentés de la même façon. Les contournements de constructeurs (et sélecteurs) définis par l'interface sont source de problèmes. C'est pourquoi l'encapsulation implique le masquage des données: l'objet a la maitrise de ses attributs via ses méthodes seules les méthodes sont accessibles. **Dans l'esprit de la POO, l'interaction avec les objets d'une classe se fait avec essentiellement avec les méthodes.** On y utilise des **accesseurs** et des **mutateurs**: ce sont des méthodes qui permettent respectivement de consulter ou de modifier ces attributs d'instance.

Cela est tout à fait possible avec Python mais peut aussi être géré autrement. Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier l' attribut d'un objet, vous écrivez `objet.attribut = nouvelle_valeur` mais vous pouvez **prévenir** l'utilisateur que des variables sont à **visibilité privée** en les préfixant avec un tiret du 8:`__attribut_privé`.

0.3 Méthodes

Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe. Les attribut et les méthodes constituent donc l'objet et sont encapsulés dans cet objet. Créeons une classe `Voiture` pour illustrer cela. Vous complèterez la classe `Point` en exercice.

Exemple

```
p1 = Point()
p1.x = 4
p1.y = 10
print(p1.x)
```

Les méthodes d'instance prennent toujours en premier paramètre `self`: référence à l'instance de l'objet manipulé car il faut toujours pouvoir désigner l'instance à laquelle la méthode sera associée.

```
>>> class Voiture():
...     def rouler(self):
...         print("VRrrooOmMM")
...     def affichercouleur(self):
...         print("la voiture est rouge")
...
>>> titine = Voiture()
>>> titine.rouler()
VRrrooOmMM
>>> titine.affichercouleur()
la voiture est rouge
```

Mais les instances de la classe voiture ne seront pas toutes rouges. Nous voudrions donc pouvoir passer la couleur en paramètre à l'instanciation de l'objet.

0.3.1 Méthodes réservées

Les méthodes réservées ou *dunder methods* sont des méthodes d'instance que Python reconnaît et sait utiliser dans certains contextes: elles permettent de définir le comportement des fonctions natives comme `print()`, `len()` ou des opérateurs `+, <, >` ou `in`. Il y en a environ une centaine.

Le nom d'une méthode spéciale prend la forme :`__nom_methode__`

La méthode `__init__` Une méthode d'instance spéciale est pour prévue passer en paramètre la valeur d'un attribut à l'instanciation Appelée abusivement **constructeur** de la classe (c'est en réalité un initialisateur), elle porte toujours le nom réservé `__init__`.

On y définit les attributs d'une instance en suivant cette syntaxe :`self.nom_attribut = valeur`. Il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

```
>>> class Voiture():
...     def __init__(self, couleur= "non renseignée", marque="non renseignée"):
...          self.c = couleur
...          self.m = marque
...     def rouler(self):
...          print("VRrrooOmMM")
...     def affichercouleur(self):
...          print(f"La voiture est de couleur {self.c}")
...     def affichermarque(self):
...         print(f"La voiture est de marque {self.m}")
...
>>> titine = Voiture("verte", "Peugeot")
>>> titine.rouler()
VRrrooOmMM
>>> titine.affichercouleur()
La voiture est de couleur verte
>>> bolide = Voiture("rouge", "Lotus" )
>>> bolide.affichercouleur()
La voiture est de couleur rouge
>>> print(bolide)
<__console__.Voiture object at 0x7f921082a7f0>
```

On remarque que l'affichage avec `print` n'est pas vraiment convivial, on peut modifier cela avec une autre méthode réservée.

La méthode `__str__` Lorsque l'on passe des paramètres à la fonction `print`, celle-ci appelle la fonction `str()` pour les convertir en chaînes de caractères. Définir cette méthode à l'intérieur d'une classe permet d'utiliser cette fonction `str()` sur les instances de l'objet. `__str__` doit se comporter comme `str()` et donc renvoyer une chaine de caractère. Ajoutons cette méthode à notre classe avec les lignes:

```
class Voiture():
    def rouler(self):
        print("VRrrooOmMM")

>>> bolide = Voiture("Noire", "Lotus")
>>> print(bolide)
Lotus Noire
```

La méthode `__eq__` Notez que si vous instanciez deux objets de la classe `Point` avec les mêmes valeurs, les deux objets ne sont pas reconnus comme égaux avec l'opérateur `==`:

```
>>> p1 = Point(1, 3)
>>> p2 = Point(1, 3)
>>> p1 == p2
False
```

Pour modifier ce comportement, vous devez utiliser la méthode réservée `__eq__`, à laquelle python fait appel lorsque l'opérateur `==` est utilisé. Voir exercice 1.

Note:

- Il existe bien sûr d'autres méthodes réservées qu'il peut être utile de connaitre comme la méthode `__add__` qui permet d'adapter le comportement de l'opérateur + sur les objets créés ou `__repr__` qui fonctionne un peu comme `__str__` mais pour la représentation de l'objet dans le shell.
- Comme les fonctions et les méthodes, des commentaires peuvent être associés à une classe, ils sont affichés grâce à la fonction `help()`.

0.4 Composition

Les objets peuvent bien sûr être réutilisés pour créer d'autre objet. Ecrivons une classe `Garage` qui comportera plusieurs éléments de type `Voiture`.

```
class Garage():
    def __init__(self):
        self.stock = [Voiture("Noire", "Lotus"), Voiture("Bleue", "Peugeot"),
                      Voiture("Verte", "Renault")]
```

Nous avons composé un nouvel objet `Garage` en prenant des objets `Voiture` comme attributs, ce procédé s'appelle la composition.

A faire: Ecrire les méthodes d'instance suivantes:

1. Une méthode `ajouter()` qui permettra d'ajouter une voiture au stock.
2. Une méthode `get_stock()` renvoyant la liste des voitures du stock. C'est un **accesseur**.
3. Une méthode `nb_voiture` renvoyant le nombre de voitures du stock.
4. La méthode spéciale `__str__` afin d'afficher clairement le stock avec la fonction `print()`.

0.5 Attributs de classe

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe :

```
class Chien:

    espece = 'canine'          # class variable shared by all instances

    def __init__(self, nom):
        self.nom = name     # instance variable unique to each instance
```

```
>>> d = Chien('Nicky')
>>> e = Chien('Apollo')
>>> d.espece              # shared by all dogs
'canine'
>>> e.espece              # shared by all dogs
'canine'
>>> d.nom                  # unique to d
'Nicky'
>>> e.nom                # unique to e
'Apollo'
```

Les attributs de classe étant partagés par toutes les instances, **on évitera d'utiliser des données mutables pour ces attributs**. En effet, toute modification à partir d'une instance entraine une modification au niveau de la classe. *(source:https://docs.python.org/fr/3/tutorial/classes.html#class-objects)*

0.6 Espaces de noms des classes et instances

Vous savez que les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de cette fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence. Pour décrire la même chose en d'autres termes, nous pouvons dire que **chaque fonction possède son propre espace de noms**, indépendant de l'espace de noms principal.

Vous avez appris également que les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais en consultation seulement : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction `global`). Il existe donc une sorte de hiérarchie entre les espaces de noms. Nous allons constater la même chose à propos des classes et des objets. En effet :

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées variables de classe ou attributs de classe.
- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou attributs d'instance.
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.
- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Si le même nom d'attribut apparaît à la fois dans une instance et dans une classe, alors la recherche d'attribut donne la priorité à l'instance :

Exemple:

```
>>> class Entrepot:
        fonction = 'stockage'
        region = 'ouest'

>>> w1 = Entrepot()
>>> print(w1.fonction, w1.region)
stockage ouest
>>> w2 = Entrepot()
>>> w2.region = 'est'
>>> print(w2.fonction, w2.region)
stockage est
```

Règle d'or: Les attributs sont déclarés privés: ils sont accessibles uniquement au sein de la classe

Python laisse les méthodes et les attributs publics par défaut Pour se confirmer à la règle ci-dessus, la plupart des langages objets font exactement le contraire ! En Python, pour rendre une méthode ou un attribut **privé**, on le précède de `__attributprive`. Du moins c'est une première approche.

0.7 En résumé

En POO, un programme est un ensemble d'entités qui interagissent. Ces entités sont appelés objets. Une classe permet de définir une famille d'objets. A partir d'une classe, on peut créer

aurtant d'objets que l'on veut: ce sont les instances de la classe. La classe est en quelque sorte le moule servant à créer les objets.

Définir une classe, c'est définir les attributs et les méthodes caractérisant tous les objets instances de la classe. On respecte en POO les trois principes suivants:

1. L'encapsulation: Les méthodes et attributs relatifs à une classe sont tous codés dans la même unité de conception. Cela implique de regrouper les données (attributs) avec les traitements qui les manipulent (méthodes).
2. L'interface: un objet est documenté et présenté avec ses services tout en cachant sa structure interne.
3. Le code des objets est mis en commun lorsqu'ils ont des comportments similaires.

De nouveaux types Certains langages offrent au programmeur des moyens de déclaration de nouveaux types comme l'instuction `struct` en C. En Python, un de ces moyens est de passer par la programmation orientée objet en définissant de nouvelles classes. Cela permettra de contrôler le type de certaines données, imaginons que vous utilisez un dictionnaire pour gérer un groupe d'étudiant, et un autre pour gérer les salles de cours du lycée. Vos déclarations pourraient ressembler à ceci:

```
>>> etudiant1 = {'nom' = 'Rodriguez', 'prenom' = 'Bender'}
>>> salle_cours1 = {'nom' = 'LLR1', 'PC' = 'True', 'places'=35 }
>>> type(etudiant1) == type(salle_cours1)
True
```

La création d'une classe `étudiant` et d'une classe `salle_cours` évitera ces problèmes: *etudiant1* aura un type `etudiant` et *salle_cours1* un type `salle_cours` .

Pour distinguer classe, objet, implémentation et interface, voici une image qui pourra vous aider:

Une classe , c'est le plan d'une maison (abstrait).

Un objet , c'est la maison issue du plan (concret). Et une autre maison issue du même plan est aussi un objet qui peut avoir des caractéristiques différentes (peinture, mobilier, etc) de la première maison.

Une interface , c'est le bouton "régler la température de la maison sur 20°C": l'utilisateur n'a pas besoin de connaître l'intérieur du climatiseur pour s'en servir.

L'implémentation , c'est le climatiseur en lui-même. On peut changer de climatiseur, le bouton reste le même, avec la même consigne : obtenir 20°C. (et le climatiseur en question doit réaliser cette opération!)

0.8 Exercices

Exercice

1. Reprenez la classe `Point` et remplacez la méthode `affichecoord()` par la méthode réservée `__str__()`.

```
>>>A = Point(1, 4)
>>>print(A)
(1,4)
```

2. Ajoutez une méthode `distAB(self, p)` qui calcule la distance entre deux points A et B.

```
>>>A = Point(1, 4)
>>>B = Point(3, 3)
>>>A.distAB(B)
2.23606797749979
```

3. Ecrivez la méthode `__eq__()` pour que deux points ayant les mêmes coordonnées soient reconnues comme *égaux*.

Exemple:

```
>>> p1 = Point(1, 3)
>>> p2 = Point(1, 3)
>>> p1 == p2
True
```

4. Ecrivez une méthode `get_coord()` renvoyant les coordonnées du point sous la forme d'un tuple.

Margaret Hamilton, cheffe du projet informatique de la mission Apollo (1969). La POO fait alors partie de son domaine d'expertise.

Exercice Définissez une classe `Domino()` qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0). Deux autres méthodes seront définies :

- la méthode `__str__()` pour afficher les points présents sur les deux faces.
- une méthode `valeur()` qui renvoie la somme des points présents sur les 2 faces.
- une méthode `somme()` qui renvoie la somme des points présents sur deux dominos.
- Utilisez la méthode réservée `__eq__()` pour que deux dominos soient reconnus "égaux" lorsque la somme de leur face est égale.

Exemples d'utilisation de cette classe :

```
>>>d1 = Domino(2,6)
>>>d2 = Domino(4,3)
>>>print(d1)
face A : 2 face B : 6
>>>print(d2)
face A : 4 face B : 3
>>>d1.valeur() + d2.valeur()
15
```

Exercice Définissez une classe `CompteBancaire()` qui permette d'instancier des objets tels que compte1, compte2, etc. Le constructeur de cette classe initialisera deux attributs d'instance nom et solde, avec les valeurs par défaut 'Dupont' et 1000. Trois autres méthodes seront définies : - `depot(somme)` permettra d'ajouter une certaine somme au solde - `retrait(somme)` permettra de retirer une certaine somme du solde - `affiche()` permettra d'afficher le nom du titulaire et le solde de son compte. Exemples d'utilisation de cette classe :

```
>>> compte1 = CompteBancaire('Haddock', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Haddock est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

Exercice

1. Créer une classe `Chat`. Celle-ci permettra de créer des instances ayant pour attribut le nom, la couleur et l'âge.
2. Une méthode d'instance `Miauler` permettra d'afficher "Miaou", "Mewou" ou "MaaAAww" de façon aléatoire. Un attribut de classe `espece` sera affecté à `"Félin"`.
3. Créer une classe `FamilleChat`. Celle-ci permettra de créer des instances ayant pour attribut le nom de la famille et ayant une méthode `afficher` donnant le nombre de châton (âge < 1 an) et le nombre d'adultes.

Exercice Créer une classe `Vecteur`. Celle-ci permettra de créer des instances ayant pour attribut une abscisse et une ordonnée à partir de deux points de type `Point`, utilisez la classe `Point` de l'*exercice 1*. Pour rappel, l'abscisse d'un vecteur $\overrightarrow{AB}$ est $x_B - x_A$.

Exercice Créez une classe `Chien` et proposez méthodes et attributs pour que les instances de cette classe aient des points de santé , un nom (Nicky, Elfy, Apollo) etc. De plus:

- Un chienA peut mordre un chienB (fait baisser le point de santé de B).
- Un chien peut manger, ce qui augmente ses points de santé.
- Un chien peut mâchouiller une chaine de caractères (ce qui renvoie la même chaîne mélangée par exemple "chausson" $\longrightarrow$ "socahnsu")
- Un chien peut grogner (ce qui renvoie "Grrrr...." + son aboiement) ...
- Ajoutez un attribut de classe `espece` affecté à `canine`.

Exercice: jeu de cartes Le but de cet exercice est de créer une classe `Carte` qui permettra de créer des instances pour représenter des cartes comme dans un jeu de 32 cartes avec 4 couleurs: trèfle, pique, carreau, coeur et 8 valeurs entières de 7 à 14 puis une classe `PaquetCartes`.

1. Ecrivez une classe `Carte` dont le constructeur possèdera deux attributs d'instance `valeur` et `couleur`. Utilisez la méthode réservée `__str__()` pour proposer un affichage convivial d'un objet instance de la classe `Carte`.

   ```
   >>> carte1 = Carte(8, "pique")
   >>> print(carte1)
   8 de pique
   ```

2. Utilisez une **assertion** pour éviter qu'une valeur ou une couleur de carte proposée par l'utilisateur soit erronée.
3. Ecrivez une classe `JeuCartes` **composée** des 32 cartes.
4. Utilisez la méthode réservée `__str__()` pour proposer un affichage convivial d'un objet instance de la classe `JeuCarte`.
5. Ecrivez une méthode:
   - `tirer()` qui prélève une carte dans le jeu: elle renvoie une carte et la supprime du jeu.
   - `melanger()` qui mélange de façon aléatoire le jeu.

   Les questions qui suivent sont hors programme NSI:
6. **Exceptions:** Utilisez un bloc `try...except` pour capturer l'erreur `IndexError` qui survient si on veut tirer une carte alors que le jeu est vide. On peut imaginer de retourner `None` dans ce cas.

```
try:
    instruction
except IndexError as erreur:
    print("Le paquet est vide")
    return None
```

   Note sur les exceptions: `IndexError` est une **exception**, on trouve parmi ces exceptions: `ZeroDivisionError` ou encore `TypeError` dont le nom parle de lui-même. Il serait fastidieux d'en lister plus; pour un complément sur les exceptions en Python, voir: `https://python.doctor/page-apprendre-exceptions-except-python-cours-debutant` La gestion des exceptions n'est bien sûr pas spécifique à la POO.
7. **Héritage:** Nous pouvons voir le paquet de carte comme un jeu de carte particulier. En POO, on dira que la classe `PaquetCartes` hérite de la classe (mère) `JeuCartes`. Les avantages sont nombreux: on récupère les méthodes et attribut visibles de la classe mère. Cela factorise le code. Voici la syntaxe pour créer une classe héritant d'une autre, ici `PaquetCartes` hérite de `JeuCartes` qui est sa classe mère:

```
class PaquetCartes(JeuCartes):

def __init__(self):
    super().__init__(paramètres)
```

```
class PaquetCartes(JeuCartes):

def __init__(self):
    JeuCartes.__init__(self, paramètres)
```

   `super()` fait référence à l'objet classe mère; c'est pour cela que nous n'utilisons pas `self.` On peut aussi écrire `JeuCartes.__init__(self)` dans ce constructeur. Créer une classe `PaquetCartes` héritant de la classe `JeuCartes`.

   (a) On utilise un paramètre de type booléen dans le constructeur: s'il est à `True`, le jeu créé sera vide, et nous pourrons ajouter des cartes afin de créer le paquet voulu à l'aide d'une méthode `ajouter` de notre classe `PaquetCartes`. Modifiez la classe `JeuCartes` afin de créer un jeu de cartes vide lorsque ce paramètre vaut `True`.

   (b) Ecrivez la méthde `ajouter` de la classe `PaquetCartes`.

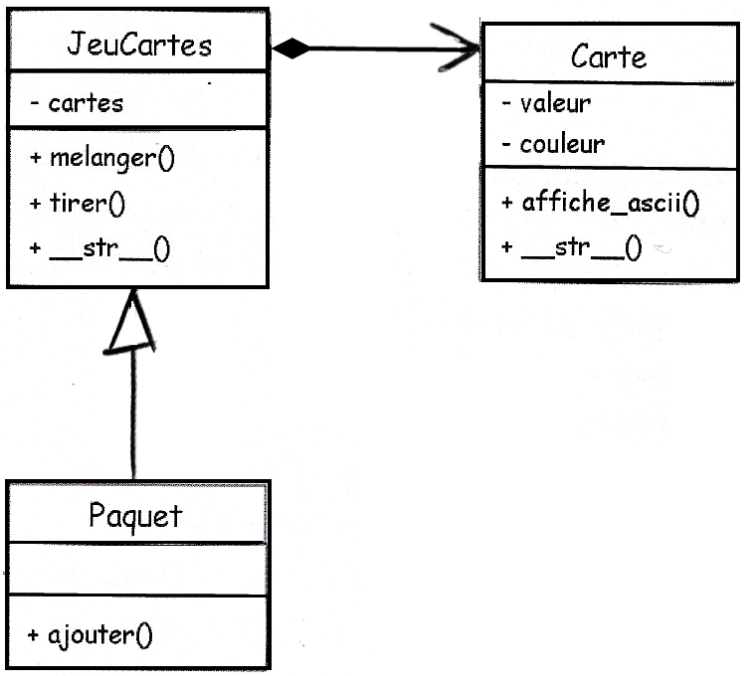
   (c) Créez une instance de cette classe et ajoutez deux cartes à ce paquet vide.

```
class PaquetCartes(JeuCartes):

def __init__(self):
    JeuCartes.__init__(self, True)

def ajouter(self, carte):
    pass
```

Modéliser les objets:UML Dans les faits, on décrit d'abord son programme avec les objets et leurs interactions. Les diagrammes UML (Langage de Modélisation Unifié) permettent de générer différents types de diagramme et les logiciels (Umbrello, Modelio, Papyrus etc.) génèrent une partie du code correspondant au diagramme de classe dessiné en un clic:



Chaque rectangle représente une classe avec:

- Son nom
- Ses attributs
- Ses méthodes.

Les flèches indiquent les interactions, composition(le losange noir indique la classe utilisatrice) ou héritage. UML permet de générer d'autres types de diagramme, voir `https://fr.wikipedia.org/wiki/UML_(informatique)`)

0.9 Références

- *Python au lycée: Tome 2 (Arnaud Bodin)*:`http://exo7.emath.fr/`
- `https://docs.python.org/fr/3/tutorial/classes.html`
- Programmer avec Python3 (G.Swinnen).
- *Linux Magazine HS 77.*
- Exercices en vrac, liste de diffusion NSI.