# Éléments de Programmation
# Cours 11 - Paradigmes de Programmation
# Programmation Orientée Objet

Romain Demangeon

1I001 - Section 11 PEIP

28/11/2017

UPMC SORBONNE UNIVERSITÉS

# Précisions

- Tutorat:
  - Lundi 04/12 - 18h00 - Atrium 423
  - Mercredi 13/12 - 18h00 - Atrium 253
  - Jeudi 13/12 - 18h00 - Atrium 253
  - Lundi 18/12 - 18h00 - Atrium 249
  - Mardi 19/12 - 18h00 - Atrium 257
- Examen Final:
  - Jeudi 11/01/18 de 14h à 16h.
- Cours 11:
  - Paradigmes de programmations.
  - Introduction à la Programmation Orientée Objet.
  - Exemple de projet POO: un jeu vidéo.

# Projets

- On programme rarement une seule fonction à la fois.
- Programmer un projet ?
  - savoir ce qu'on programme,
  - connaître un style de programmation adéquat,
  - choisir un langage adapté,
  - choisir des outils,
  - construire une approche.
- Exemple 1:
  - programmer des calculs sur le génôme à partir de données ADN,
  - programmation impérative sur les chaînes,
  - choisir Python,
  - utiliser Biopython,
  - spécifier les fonctions et leurs dépendances.
- Exemple 2:
  - programmer une application web,
  - programmation objet et web,
  - choisir Python pour le serveur, Javascript pour le client,
  - utiliser Django,
  - décider la division client/serveur/persistance/métier.

# Projets (II)

- Exemple 3:
  - programmer un serveur de données,
  - programmation objet et BDD,
  - choisir MySQL pour les requêtes, Java pour la logique,
  - utiliser la bibliothèque Hibernate,
  - construire la relation objet / entrée (ORM).
- Exemple 4:
  - programmer un jeu vidéo,
  - programmation objet,
  - choisir C++ (logique, netcode), Python (scripts), Lua (IA), ...
  - utiliser un moteur graphique,
  - séparer gameplay/graphismes/DA/netcode/logique, répartir les tâches.
- Souvent, projet d'équipe:
  - chef de projet: connaissance techniques, de management, de gestion, finance, ....

# Paradigmes de programmation (Rappels)

## Paradigme Impératif

- brique de base: l'instruction.
- composition: séquences d'instructions.
- itération: les boucles (`while`).
- langages: C, Pascal, ...

## Paradigme Fonctionnel

- brique de base: la fonction.
- composition: appels et compositions de fonctions.
- itération: la récursion,
- langages: OCaml, Haskell, ...

## Paradigme Objet

- brique de base: l'objet.
- composition: appels de méthodes depuis des objets.
- itération: boucles et récursion de méthodes.
- langages: Java, C++, ...

- Python est multi-paradigme.
- le Cours 1I001 est orienté impératif.
  - hors-programme: des fonctionnelles (Cours 04, Cours 08), des objets (Cours 11).

# Dualité Fonction/Procédure

- Distinction historique:
  - langages de programmation procédurale et impérative,
  - langages de programmation fonctionnelle et récursive.
- Langages archétypaux:
  - FORTRAN, Pascal, C,
  - Lisp (Scheme), Caml, Haskell.
- Distinction de moins en moins présente:
  - récursion efficace en C,
  - boucles et mémoires en OCaml.
- Langages de programmation multi-paradigme (Python)

# Programmation Impérative

- *impero*: régner, puis ordonner.
- Approche historique: Machines de Von Neumann (1950),
- Monde de l'instruction:
  - donner des ordres à la machine,
  - style historiquement (et moralement) premier.
- Interagir avec une machine à mémoires:
  - notion de variables globales qui persistent en mémoire et que l'on peut modifier.
  - pas de fonctions mais des instructions qui modifient la mémoire de la machine:
    - copier le contenu de mémoires,
    - additionner ou multiplier des mémoires, ...
  - structures de contrôle:
    - simple: alternatives, boucles,
    - complexes: sauts.
  - entrée (clavier, fichier) / sortie (écran, fichier)
- pas de fonctions.

# Programmation Impérative (II)

```
print("Calcul de la factorielle n!")
print("============================")

print("Saisir la valeur de n: ", end='')

entree = int(input())
sortie = 1

while entree > 0:
    sortie = sortie * entree
    entree = entree - 1

print("n! = " + str(sortie))
```

- Approche bas-niveau (instructions machines),
  - aujourd'hui, toujours la seule manière de parler au matériel.
- Programmation possible en Python,
- Pas d'étude des variables globales (en dehors des fonctions) en 1I001,
  - apparaîssent quand même: `BD`, `BaseUPMC`, ...
- Par abus de langage: la programmation impérative désigne le style s'appuyant la manipulation de variables et l'utilisation de boucles

# Programmation Procédurale

- Programmation (avec variables et boucles) de plus haut-niveau,
- Création de procédures avec des variables locales,
  - paramètres instanciés par des arguments lors d'appels de fonctions.
  - automatisation/généralisation des calculs,
  - organisation des calculs.
  - économie de mémoire.
- Style du Cours 1I001:
  - les procédures Python sont les fonctions

```
def factorielle(n):
    """ int -> int
    Hypothese : n >= 0 """

    # resultat : int
    resultat = 1

    # k : int
    for k in range(2, n + 1):
        resultat = resultat * k
    return resultat
```

- Variable locale:
  - Variable privée à la fonction (pas de risque),
  - Plusieurs appels à la même fonction ont des variables locales différentes.
- Nécessite une gestion plus fine de la mémoire (pile et tas)

# Programmation Procédurale (Effets de bords)

- Un appel à `factorielle` ne modifie ni les variables globales ni les entrées/sorties:
  - c'est le cas de toutes les fonctions de 1I001.
- Effet de bord: appeler la fonction (de n'importe où) modifie l'état (var. glob., i/o) de la machine.
  - **`print`**,
  - modification de l'argument.
- Effets de bord possibles en Python, pas (vraiment) au programme de 1I001 (`turtle`, **`print`**).

```
def frequences(L):
    """ list[alpha] -> dict[alpha:int]"""

    # freqs : dict[alpha:int]
    freqs = dict()

    # e : alpha
    for e in L:
        if e in freqs:
            freqs[e] = freqs[e] + 1
        else:
            freqs[e] = 1
    return freqs
```

```
def supprimer_lettre_repetees(L):
    """ list[alpha] -> NoneType
    Supprime les lettres presente au moins deux fois
    dans la liste L."""

    # freqs : dict[int:int]
    freqs = frequences(L)

    # e : alpha
    for e in L[:]:  # Remarque : il faut copier L
        if freqs[e] > 1:
            L.remove(e)
```

- Modification directe en mémoire du paramètre de la fonction.

# Programmation Récursive

- Approche historique: fonctions récursives (primitives) de Ròzsa Péter (1920-1930).
  - fonction définie par des opérations de base et un point fixe.
  - récursion: fonction qui calcule en s'utilisant elle-même.
- Approche de haut-niveau, mathématique:
  - naturelle pour le programmeur.
  - efficacité ?
- Possible en Python, en Complément en 1I001.
- Pas de variables, pas d'effet de bord.
  - Calcul: réduction (évaluation) d'une expression.
  - Exemple: factorielle $n! = \begin{cases} 0 & \text{si } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$

```
def factorielle(n):
    """ int -> int
    Hypothese : n >= 0
    Retourne la factorielle de n."""

    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)
```

# Programmation Récursive (II)

- Variante: utilisation de variables locales uniquement.
  - effets de bord dans les fonctions.
- Mais, pas de variables globales ni d'effets de bords à *top level*:
  - les fonctions (re)construisent leur résultats.

```
def suppression_lettres_repetees_fun(L):
    """ list[alpha] -> list[alpha]
    Retourne une liste composee des elements
     de L sans les lettres repetees."""

    # freqs : dict[int:int]
    freqs = frequences(L)
    # LR : list[alpha]
    LR = []

    # e : alpha
    for e in L:
        if freqs[e] == 1:
            LR.append(e)
    return LR
```

```
def suppression_lettres_repetees_fun2(L, F):
    """ list[alpha] * dict[alpha:int] -> list[alpha]
    Retourne une liste composee des elements
    de L sans les lettres repetees, a partir d'un dictionnaire
    de frequence F"""

    if L == []:
        return []
    elif F[L[0]] == 1:
        return [L[0]] + suppression_lettres_repetees_fun2(L[1:],F)
    else:
        return suppression_lettres_repetees_fun2(L[1:],F)
```

- Naturelle pour les suites, les listes (séparation tête/queues), etc
- Compréhensions: esprit de la programmation récursive.
  - écriture mathématique naturelle.

# Programmation Fonctionnelle

- Evolution de la programmation récursive.
- Motto: Considérer les fonctions comme citoyens de première classe.
- Approche historique: le $\lambda$-calcul d'Alonzo Church (1930).
  - langage mathématique où tout est fonction.
- Concrètement:
  - les fonctions peuvent prendre en paramètre des fonctions.
  - on peut définir des fonctions de manière abstraite
    - sans nom, sans **def**,
    - n'importe où dans une expression.

```
def map_fun(f, L):
    """ alpha -> beta * list[alpha] -> list[beta]
    applique la fonction f a tous les elements de L"""

    if L == []:
        return []
    else:
        return [f(L[0])] + map_fun(f, L[1:])

def liste_carres(L):
    """ list[int] -> list[int]
    renvoie la liste des carres des nombres L"""

    return map_fun(lambda x: x * x, L)
```

- Possible en Python (et en Java !), mais pas en 1I001.

UPMC SORBONNE UNIVERSITÉS

# Programmation Orientée Objet (OOP)

- Approche historique: programmation avec données (1990)
- Objectif: rapprocher les données de leur traitement au sein d'une entité appelée Objet.
- Mélange la programmation fonctionnelle (traitement par fonction, variables locales dans les objets) et impérative (effets de bords, objets comme variables globales).
- Langages OO stars: C++, Java.
  - Python initialement OO.
  - plupart des langages récents: OCaml, JavaScript (prototypes).
- Avantages:
  - Représenter des objets du monde réel.
  - Créer des types de données.

# Objets du monde réel

- Approche historique: Simula-67
  - écriture de simulations.
  - représentations d'entités réelles en machine.
  - comportement simulé par un programme.

## Feu Tricolore

Un feu tricolore fournit une information de couleur permettant de réguler la circulation:

- la couleur rouge ferme la circulation
- la couleur vert ouvre la circulation
- la couleur orange est une couleur transitoire du vert au rouge

Spécification:

- Etats possibles de l'objet.
  - état stocké dans des variables propres à l'objet.
- Opérations effectuées par l'objet.
  - fonctions propres à l'objet.

# Classes et Objets

## Schéma Classe/Objet

- une classe définit le modèle (patron) des objets.
- les objets sont des instances de cette classe.
  - il peut exister, à un moment donné, plusieurs (ou zéro) objets d'une même classe.
- un objet possède des attributs qui lui sont propres.
- un objet possède des méthodes qui permettent l'interaction.
- à la création d'un objet, on initialise ses attributs.
- un objet peut avoir une représentation.

## Syntaxe

- On définit une classe avec le mot-clef `class`.
- Une classe contient une suite de méthodes:
  - une méthode d'initialisation `__init__`.
  - des méthodes d'utilisation.
  - une méthode de représentation (facultative) `__repr__`

# Classe et Objets (II)

- L'objet lui-même est désigné par `self`.
- Les méthodes manipulent des attributs:
  - privés `self._attr` .
  - publics (cf. plus loin) `self.attr` .

```
class FeuTricolore:
    """Representation d'un feu tricolore de circulation."""

    def __init__(self):
        """ self -> []
        Construit le feu tricolore initialement au rouge."""
        self._couleur = 'rouge'  # attribut prive pour la couleur

    def couleur(self):
        """ self -> str
        Accesseur pour la couleur du feu."""
        return self._couleur

    def change(self):
        """ self -> NoneType
        Change le feu (cycle vert-orange-rouge)."""

        if self.couleur() == 'vert':
            self._couleur = 'orange'
        elif self.couleur() == 'orange':
            self._couleur = 'rouge'
        elif self.couleur() == 'rouge':
            self._couleur = 'vert'
        else:
            raise Exception('Mauvaise couleur')

    def __repr__(self):
        return "Feu({})".format(self.couleur())
```

# Classes et Objets (III)

- On crée un objet avec `obj = Classe()`
  - appel de la méthode `__init__`.
- On appelle une méthode avec `obj.met()`
- On peut vérifier qu'un objet est une instance d'une classe avec `isinstance`

```
# feu : FeuTricolore
feu = FeuTricolore()

# feu2 : FeuTricolore
feu2 = FeuTricolore()

feu.change()

feu.couleur()
feu2.couleur()

isinstance(feu, FeuTricolore)
```

## Points du plan.

- Attributs: abscisse et ordonnée.
- Méthodes: récupérer les coordonnées, modifier les coordonnées, calculer la distance à un autre point.

# Classes et Objets (IV)

```
class Point:
    """Representation d'un point du plan."""

    def __init__(self, x, y):
        """ self * float * float -> Point
        Construit point de coordonnees initiales (x, y)."""
        self._abscisse = x
        self._ordonnee = y

    def coord(self):
        """ self -> tuple[float, float]"""
        return (self._abscisse, self._ordonnee)

    def regle_abs(self, x):
        """ self * float -> NoneType """
        self._abscisse = x

    def regle_ord(self, y):
        """ self * float -> NoneType """
        self._ordonnee = y

    def distance(self, p2):
        """ self * Point -> floar"""
        x1, y1 = self.coord()
        x2, y2 = p2.coord()
        return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

    def __repr__(self):
        return "Point {}".format(self.coord())
```

- `self` comme premier argument pour définir une méthode.

# Héritage

- Idée: classifier et hierarchiser les objets.
- Définir des sous-classes (tous les `Etudiant` sont des `Personne`).
- Une sous-classe hérite de toutes les méthodes de sa super-classe.
- On peut redéfinir des méthodes.
- Syntaxe: `class Sous(Super)`

```
class PointCouleur(Point):
    """Representation d'un point colore du plan."""

    def __init__(self, x, y, coul):
        """ self * float * float * str -> PointCouleur
        Construit point de coordonnees initiales (x, y)."""
        self._abscisse = x
        self._ordonnee = y
        self._couleur = coul

    def couleur(self):
        """ self -> tuple[float, float]"""
        return self._couleur

    def regle_coul(self, could):
        """ self * str -> NoneType """
        self._couleur = coul

    def meme_coul(self, p2):
        """ self * Point -> bool"""
        return self.couleur() == p2.couleur()

    def __repr__(self):
        return "Point {} de couleur {}".format(self.coord(), self.couleur())
```

# Heritage (II)

```
class Point3D(Point):
    """Representation d'un point de l'espace."""

    def __init__(self, x, y, z):
        """ self * float * float * float-> Point3D
        Construit point de coordonnees initiales (x, y, z)."""
        self._abscisse = x
        self._ordonnee = y
        self._profondeur = z

    def coord(self):
        """ self -> tuple[float, float, float]"""
        return (self._abscisse, self._ordonnee, self._profondeur)

    def regle_prof(self, z):
        """ self * float -> NoneType """
        self._ordonnee = z

    def distance(self, p2):
        """ self * Point -> float"""
        x1, y1, z1 = self.coord()
        if isinstance(p2, Point3D):
            x2, y2, z2 = p2.coord()
        else:
            x2, y2 = p2.coord()
            z2 = 0
        return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2 + (z1 - z2) ** 2)
```

- on a redéfini `p.coord()`, pour calculer la distance, on peut tester la classe du point paramètre.

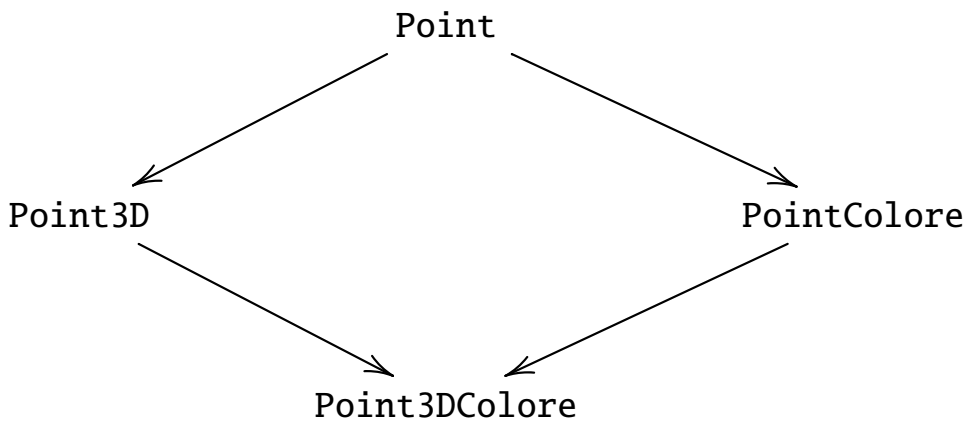
# Héritage Multiple

- On peut hériter de plusieurs classes à la fois.
- Attention aux problèmes de redéfinition.

```
class Point3DCouleur(Point3D, PointCouleur):
    """Representation d'un point de l'espace en couleur"""

    def __init__(self, x, y, z, coul):
        """ self * float * float * float * str-> Point3DCouleur
        Construit point de coordonnees initiales (x, y, z)."""
        self._abscisse = x
        self._ordonnee = y
        self._profondeur = z
        self._couleur = coul
```

- Les classes permettent de représenter la hierarchie des entités manipulées dans un projet.
  - la structure du code correspond à la structure du projet.
  - lien avec les *diagrammes de classes* (UML)



# Créer des types de données

- Précédemment, manipulation de types de données mis à disposition par Python:
  - types simples `bool`, `int`.
  - séquences `range`, `str`, `list[alpha]`,
  - *n*-uplets `tuple[T1, ..., Tn]`
  - tableaux d'association, `set[alpha]`, `dict[alpha : beta]`
- En fait,

# Créer des types de données

- Précédemment, manipulation de types de données mis à disposition par Python:
  - types simples `bool`, `int`.
  - séquences `range`, `str`, `list[alpha]`,
  - *n*-uplets `tuple[T1, ..., Tn]`
  - tableaux d'association, `set[alpha]`, `dict[alpha : beta]`
- En fait, implémentés en Python avec des objets.
  - `isinstance(1, int)` donne `True`
  - `isinstance([1, 2, 3], list)` donne `True`
- Donc, parallèle entre les Classes et les Types de Données.
- Créer une nouvelle classe (`Point`, `FeuTricolore`), c'est créer un nouveau type de données.
  - On peut s'en servir pour créer nos propres types.

# Enregistrements

## Définition

Un enregistrement est un rassemblement d'un nombre fixé de données de types fixés.

- C'est une sorte (plus relâchée) de *n*-uplets.
- Une seule méthode, pour l'initialisation.
- Python autorise l'accès direct aux attributs publics:
  - Syntaxe: `obj.attribut`
  - plus besoin de l'ordre du *n*-uplet.

```
class Personne:
    """ Representation d'une personne dans une base de donnees."""

    def __init__(self, nom, prenom, age, marie):
        """ self * str * str * int * bool -> []
        Construit une personne."""
        self.nom = nom
        self.prenom = prenom
        self.age = age
        self.marie = marie

# VGE : Personne
VGE = Personne('Giscard d\'Estaing', 'Valerie', 88, True)
```

- Attention: type mutable.

# Types numériques

- Exemple: représentation des fractions
- Couple d'entiers, normalisable.

```
class Ratio:
    """Representation des fractions"""
    def __init__(self, n, m):
        if m == 0:
            raise ZeroDivisionError()

        self.quo = n
        self.div = m
        self.normalisation_coefs() # Normalisation a l'initialisation

    def normalisation_coefs(self): # divise quotient et diviseur par leur pgcd
        p = pgcd(self.quo, self.div)
        self.quo = self.quo // p
        self.div = self.div // p

    def __mul__(self, r): # redefinition *
        return Ratio(self.quo * r.quo, self.div * r.div)

    def __floordiv__(self, r): # redefinition //
        return Ratio(self.quo * r.div, self.div * r.quo)

    def __repr__(self):
        return "{}/{}".format(self.quo, self.div)
```

- redéfinition des symboles `*` et `//`.

# Types itérables

- Exemple: intervalles de lettres de l'alphabet

```
class CharRange:
    """ Representation d'intervalles de lettres de l'alphabet"""

    def __init__(self, cmin, cmax): #initialise avec un couple de lettre
        self.cmin = cmin
        self.cmax = cmax

        self.current = cmin

    def __iter__(self): # redefinition for
        return self

    def __next__(self): # definit la notion de prochain
        prochain = self.current
        if ord(self.current) == ord(self.cmax) + 1:
            raise StopIteration() # arrete le for
        self.current = chr(ord(self.current) + 1)
        return prochain

    def __repr__(self):
        return "CharRange('{}', '{}')".format(self.cmin, self.cmax)

# cr : CharRange
cr = CharRange('a', 'j')

# essai : list
essai = [c + 'a' for c in cr]
```

- une itération c'est un élément initial, une notion de prochain élément et une condition d'arrêt.
- Définition de l'itération: redéfinition du `for`.

UPMC SORBONNE UNIVERSITÉS

# Exemple de projet: Jeu vidéo

## Distinction

- Partie non-informatique (pas au programme de la Licence Info):
  - règles du jeu, *game design*,
  - graphismes: contenu, direction artistique,
  - *marketing*, *management*.
- Partie informatique (au programme de la Licence Info):
  - logique du jeu (implémentation des règles),
  - graphismes: moteur graphique,
  - *netcode*,
  - gestion de projet.

## Projet

- programmer un jeu (interactivité) vidéo (représentation visuelle).
- programmation objet.
- langage Python de 1I001.
- outils: une bibliotheque graphique et d'évènements (`tkinter`)
- approche ?

# Cahier des charges

## Fonctionnalités

- le joueur contrôle un personnage au clavier.
- le terrain de jeu est grille comportant des murs.
- des montres se déplacent aléatoirement sur la grille.
- un monstre peut frapper un joueur adjacent, le joueur perd une vie.
- le joueur peut tuer un monstre en se déplaçant sur sa case.
- le joueur est temporairement invincible apres avoir été touché.

Souvent, description textuelle de *use cases*: *"Alice lance une partie. Elle voit son personnage à l'écran dans un labyrinthe et cinq monstres autour. Elle se déplace vers un monstre, qui la frappe. Son personnage devient invincible pendant quelques secondes, pendant lesquelles elle réussit à se déplacer sur la case du monstre, qui disparait."*.

# Approche (Générale)

## Choix implémentation

- Le Terrain de jeu est une grille.
- Joueur, Monstres et Murs sont des Objets.
  - position stockée par des coordonnées (`get_pos()` et `set_pos(x, y)`).
  - méthode graphique `dessine()`
  - méthode de déplacement pour joueur et monstres `deplace()`
- L'etat du jeu contient liste d'Objets.
- Une boucle principale calcule les tours de jeu, à chaque tour:
  - On dessine chaque Objet de l'état.
  - On déplace les Objets mobiles de l'état.

Mécanismes génériques, il reste à décrire les mécanismes particuliers (déplacement des monstres et du joueur, initialisation, invincibilité).

# Classe Etat

```
class Etat:
    """ Represente l'etat du jeu."""

    def __init__(self):
        """ self ->  Nonetype """
        # self._obs : list[Objet]
        self._obs = [] # liste des objets de l'etat
        # self._gagne : bool
        self._gagne = False # True si le jeu est gagne
        # self._perdu : bool
        self._perdu = False # True si le jeu est perdu
        # self._tour : int
        self._tour = 0 # tour courant du jeu

    def compte(self):
        """ self -> int
        Donne le tour courant et incremente ce nombre."""
        self._tour += 1
        return self._tour

    def get_obs(self):
        """ self -> list[Object]
        Renvoie la liste des objets."""
        return self._obs

    def ajoute_ob(self, ob):
        """ self * Objet -> Nonetype
        Ajoute un objet a l'etat. """
        self._obs.append(ob)
```

```
    def enleve_ob(self, ob):
        """ self * Objet -> Nonetype
        Enleve un objet a l'etat. """

        # R : list[Objet]
        R =[] # liste resultat

        # e : Objet
        for e in self._obs:
            if e != ob:
                R.append(e)
        self._obs = R

    def set_gagne(self):
        """ self -> Nonetype
        Met l'etat du jeu en victoire."""
        self._gagne = True

    def get_gagne(self):
        """ self -> bool
        Decide si l'etait est une victoire."""
        return self._gagne

    def set_perdu(self):
        """ self -> Nonetype
        Met l'etat du jeu en defaite."""
        self._perdu = True

    def get_perdu(self):
        """ self -> bool
        Decide si l'etait est une defaite."""
        return self._perdu
```

- Stocke l'état du jeu: liste d'objets, tour courant, victoire/defaite.

UPMC SORBONNE UNIVERSITÉS

# Classe Ob

```
class Ob:
    """ Un objet du jeu. """

    def __init__(self, x, y):
        """ self * int * int -> Nonetype """
        self._coord = (x, y) # coordonnees de l'objet

    def dessine(self, fen):
        """ self * GraphWin -> Nonetype
        Dessine l'objet dans la fenetre fen."""

        return None # par defaut, ne dessine rien

    def get_pos(self):
        """ self -> tuple(int, int)
        Retourne la position de l'objet."""

        return self._coord

    def set_pos(self, x, y):
        """ self * int * int -> Nonetype
        Deplace l'objet dans la case (x, y)"""

        self._coord = (x, y)
```

- un seul attribut (la position) avec *get* et *set*.
- par défaut la méthode `dessine` ne fait rien (mais elle existe).
- `dessine` prend en entrée une fenêtre (où dessiner).

# Classe Mur

```
class Mur(Ob):
    """ Un mur inanime. """

    def dessine(self, fen, etat):
        """ <Redefinition> """

        x, y = self.get_pos()
        rect = Rectangle(Point(x*32, y*32),Point(x*32 + 31, y*32 + 31))
        rect.setFill('black')
        rect.draw(fen)
```

- Hérite de `Ob`.
- Redéfinit la méthode `dessine`.

On écrit des fonctions utiles pour les déplacements futurs..

```
def est_vide(etat, x, y):
    """ Etat * int * int -> bool
    Decide si la case (x,y) de l'etat est vide."""

    # e : Objet
    for e in etat.get_obs(): # parcours des objets de l'etat
        if (x, y) == e.get_pos():
            return False
    return True

def est_dans_cadre(etat, x, y):
    """ Etat * int *int
    Decide si la case (x,y) n'est pas hors-limite."""

    return (0 <= x < colonnes) and (0 <= y < lignes)
```

# Classe Monstre (I)

# Initialisation

# Boucle principale

# Classe Monstre (I)

# Conclusion

- Paradigmes de Programmation:
  - impératif, puis procédural,
  - récursif, puis fonctionnel,
  - orienté objet.
- OOP:
  - les classes comme modèle,
  - attributs et méthodes internes,
  - objets instances de classe,
  - héritage entre les classes,
  - définition de types par le programmeur.

## Examen le Jeudi 11 Janvier

Bon Courage !