R201 — Développement orienté objets

Programmation Objet en Python

Clément Pagès
clement.pages@u-pec.fr

Université Paris-Est Créteil
BUT Informatique 1re année

2025-2026

Programmation objet en Python

- Python propose de la POO de manière facultative.
- Classes, instances, héritage, surcharge d'opérateurs (non disponible en Java).
- Les attributs et méthodes n'ont pas de visibilité.

Programmation objet en Python

- Python propose de la POO de manière facultative.
- Classes, instances, héritage, surcharge d'opérateurs (non disponible en Java).
- Les attributs et méthodes n'ont pas de visibilité.

ville.py

```
class Ville:
    """ Classe définifissant une ville définie par :
    - son nom
    - sa lattitude
    - sa longitude
    - son nombre d'habitant.es
    - sa surface(km^2)"""
```

Sommaire

Attributs et constructeurs

- Les attributs sont définie directement dans le constructeur.
- Un seul constructeur possible.

Attributs et constructeurs

- Les attributs sont définie directement dans le constructeur.
- Un seul constructeur possible. Mais on peut attribuer une valeur par défaut.

Attributs et constructeurs

- Les attributs sont définie directement dans le constructeur.
- Un seul constructeur possible. Mais on peut attribuer une valeur par défaut.

ville.py

```
class Ville:
    # Constructeur
    def __init__(self, nom, lat=91, lng=181, nbhab=-1, surf=-1):
        self.nom = nom
        self.lat = lat # ° latitude N/S
        self.lng = lng # ° longitude E/O
        self.nbhab = nbhab
        self.surf = surf # Km2

p = Ville('Paris', 48.856578, 2.351828, 2229621, 105.40)
r = Ville('Rouen', 49.443232, 1.099971, 110618) # r.surf = -1
h = Ville('Le Havre', lat=49.49, nbhab=172807, surf=46.95)
```

Attributs et constructeurs

- Les attributs sont définie directement dans le constructeur.
- Un seul constructeur possible. Mais on peut attribuer une valeur par défaut.

ville.py

```
class Ville:
    # Constructeur
    def __init__(self, nom, lat=91, lng=181, nbhab=-1, surf=-1):
        self.nom = nom
        self.lat = lat # ° latitude N/S
        self.lng = lng # ° longitude E/O
        self.nbhab = nbhab
        self.surf = surf # Km2

p = Ville('Paris', 48.856578, 2.351828, 2229621, 105.40)
r = Ville('Rouen', 49.443232, 1.099971, 110618) # r.surf = -1
h = Ville('Le Havre', lat=49.49, nbhab=172807, surf=46.95)
```

- Le mot-clé `self` désigne l'objet courant.

Méthodes d'instances

ville.py

```
class Ville:
    # …
    def densite_population():
        if self.surf>0 and self.nbhab>0:
            return self.nbhab/self.surf
        else:
            raise ValueError("données manquantes.")

try:
    print(f"{p.nom} : {p.densite_population()} hab/hm^2.")
    print(f"{r.nom} : {r.densite_population()} hab/hm^2.")
except ValueError as e:
    print(e)
```

Méthodes d'instances

ville.py

```
class Ville:
    # …
    def densite_population():
        if self.surf>0 and self.nbhab>0:
            return self.nbhab/self.surf
        else:
            raise ValueError("données manquantes.")

try:
    print(f"{p.nom} : {p.densite_population()} hab/hm^2.")
    print(f"{r.nom} : {r.densite_population()} hab/hm^2.")
except ValueError as e:
    print(e)
```

Terminal

```
Paris : 21153.899430740035 hab/hm^2.
Rouen : données manquantes.
```

Attributs de classes

- Les attributs de classe sont définis à l'extérieur du constructeur.

Attributs de classes

- Les attributs de classe sont définis à l'extérieur du constructeur.

ville.py

```
class Ville:
    nb_villes = 0
    def __init__(self, nom, lat=91, lng=181, nbhab=-1, surf=-1):
        # …
        nb_villes += 1

p = Ville('Paris', 48.856578, 2.351828, 2229621, 105.40)
r = Ville('Rouen', 49.443232, 1.099971, 110618) # r.surf = -1
h = Ville('Le Havre', 49.49, , 172807, 46.95)   # h.lng = 181
print(Ville.nb_villes) # Affiche 3
```

Attributs de classes

- Les attributs de classe sont définis à l'extérieur du constructeur.

ville.py

```
class Ville:
    # …
    def distance_a(self, autre_ville):
        R = 6378 # Rayon équatorial en km
        latA = radians(self.lat)
        latB = radians(autre_ville.lat)
        longA = radians(self.lng)
        longB = radians(autre_ville.lng)
        res = (R * acos(sin(latA)*sin(latB) +
                cos(latA)*cos(latB)*cos(longB-longA)))
        return res
```

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.
- Une méthode de classe peut modifier la classe et ses attributs.

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.
- Une méthode de classe peut modifier la classe et ses attributs.
- Précédées par l'indication `@classmethod`.

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.
- Une méthode de classe peut modifier la classe et ses attributs.
- Précédées par l'indication `@classmethod`.

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.
- Une méthode de classe peut modifier la classe et ses attributs.
- Précédées par l'indication `@classmethod`.

ville.py

```
class Ville:
    # …
    @classmethod
    get_nb_villes(cls): # cls : la classe elle-même.
        return Ville.nb_villes
```

Méthodes de classe

- Méthode qui agit sur la classe elle-même, ou sur les attributs de classe.
- Une méthode de classe peut modifier la classe et ses attributs.
- Précédées par l'indication `@classmethod`.

ville.py

```
class Ville:
    # …
    @classmethod
    get_nb_villes(cls): # cls : la classe elle-même.
        return Ville.nb_villes
```

- `cls` est l'analogue de `self` pour la classe.
- Une méthode de classe est appelée via `nom_classe.nom_methode()`.

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.
- Ne peuvent pas modifier la classe et ses attributs.

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.
- Ne peuvent pas modifier la classe et ses attributs.
- Précédées par l'indication `@staticmethod`.

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.
- Ne peuvent pas modifier la classe et ses attributs.
- Précédées par l'indication `@staticmethod`.

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.
- Ne peuvent pas modifier la classe et ses attributs.
- Précédées par l'indication `@staticmethod`.

ville.py

```
class Ville:
    # …
    @staticmethod
    def message_bienvenue():
        print("Bienvenue dans notre appli de gestion des villes.")
```

Méthodes `static`

- Ce sont des méthodes qui servent de fonctions au sens de la programmation impérative.
- Ne peuvent pas modifier la classe et ses attributs.
- Précédées par l'indication `@staticmethod`.

ville.py

```
class Ville:
    # …
    @staticmethod
    def message_bienvenue():
        print("Bienvenue dans notre appli de gestion des villes.")
```

- Appel via `Ville.message_bienvenue()`.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.
- Se manipule uniquement *via* une méthode.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.
- Se manipule uniquement *via* une méthode.
- S'utilise comme un attribut d'instance.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.
- Se manipule uniquement *via* une méthode.
- S'utilise comme un attribut d'instance.
- Indiquée par `@property`.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.
- Se manipule uniquement *via* une méthode.
- S'utilise comme un attribut d'instance.
- Indiquée par `@property`.

Les propriétés

En lecture seule

- Une propriété permet de créer des attributs composés accessibles en lecture seule.
- Se manipule uniquement *via* une méthode.
- S'utilise comme un attribut d'instance.
- Indiquée par `@property`.

ville.py

```
class Ville:
    # …
    @property
    def localisation_ville(self):
        d = dict()
        d[self.nom] = (self.lat, self.lng)
        return d

# Affiche {'Paris': (48.856578, 2.351828)}
print(p.localisation_ville)
```

Les propriétés

En lecture et écriture

- On peut modifier une propriété en utilisant un setter.

Les propriétés

En lecture et écriture

- On peut modifier une propriété en utilisant un setter.

ville.py

```
class Ville:
    # …
    @property
    def localisation_ville(self):
        d = dict()
        d[self.nom] = (self.lat, self.lng)
        return d

    @localisation_ville.setter
    def localisation_ville(self, nom_, coordonnes_):
        self.nom = nom_
        self.lat = coordonnes_[0]
        self.lng = coordonnes_[1]
```

Sommaire

Attributs d'instances

Accès en lecture

- Accès via la syntaxe `objet.attribut`.

Attributs d'instances

Accès en lecture

- Accès via la syntaxe `objet.attribut`.
- Ou avec un *getter* prédéfini en Python : `getattr(nom_objet, attribut)`.

Attributs d'instances

Accès en lecture

- Accès via la syntaxe `objet.attribut`.
- Ou avec un *getter* prédéfini en Python : `getattr(nom_objet, attribut)`.

Python
```
>>> getattr(p, nom)
 "Paris"
>>> r.nom
 "Rouen"
```

Attributs d'instances

Accès en lecture

- Accès via la syntaxe `objet.attribut`.
- Ou avec un *getter* prédéfini en Python : `getattr(nom_objet, attribut)`.

```Python
>>> getattr(p, nom)
 "Paris"
>>> r.nom
 "Rouen"
```

- Tout objet est muni d'un attribut spécial `__dict__` : le dictionnaire des attributs.

Attributs d'instances

Accès en lecture

- Accès via la syntaxe `objet.attribut`.
- Ou avec un *getter* prédéfini en Python : `getattr(nom_objet, attribut)`.

```Python
>>> getattr(p, nom)
 "Paris"
>>> r.nom
 "Rouen"
```

- Tout objet est muni d'un attribut spécial `__dict__` : le dictionnaire des attributs.

```Python
>>> print(p.__dict__)
 {'nom': 'Paris', 'lat': 48.856578, 'lng': 2.351828,
  'nbhab': 2229621, 'surf': 105.4}
```

Opérations sur les attributs

Accès en écriture

- Avec l'opérateur d'affectation `p.nom = "Paris !`

Opérations sur les attributs

Accès en écriture

- Avec l'opérateur d'affectation `p.nom = "Paris !`
- *Setter* prédéfini : `setattr(p, nom, "Paris ! :-)")`

Opérations sur les attributs

Accès en écriture

- Avec l'opérateur d'affectation `p.nom = "Paris !`
- *Setter* prédéfini : `setattr(p, nom, "Paris ! :-)")`
- Via le dictionnaire des attributs : `p.__dict__["nom'] = "Paris"`

Attributs d'instances

Quelques fonctions utiles

- `hasattr(objet, attr)` renvoie `True` si `objet` possède un attribut `attr`.

Attributs d'instances

Quelques fonctions utiles

- `hasattr(objet, attr)` renvoie `True` si `objet` possède un attribut `attr`.
- `delattr(objet, attr)` efface l'attribut `attr` de l'instance `objet`.

Attributs d'instances

Quelques fonctions utiles

- `hasattr(objet, attr)` renvoie `True` si `objet` possède un attribut `attr`.
- `delattr(objet, attr)` efface l'attribut `attr` de l'instance `objet`.
- `dir(objet)` renvoie la liste de toutes les méthodes d'instance disponibles pour `objet`.

Attributs de classe

- Même principe que pour les attributs d'instance.
- Dictionnaire des attributs : `Ville.__dict__`.

Visibilité des attributs

- Il n'existe pas de visibilité en POO Python : les méthodes et attributs sont lisibles et modifiables depuis les autres classes.

Visibilité des attributs

- Il n'existe pas de visibilité en POO Python : les méthodes et attributs sont lisibles et modifiables depuis les autres classes.
- On peut brouiller l'accès à une méthode ou à un attribut en le nommant `__attributBrouille`.

Visibilité des attributs

- Il n'existe pas de visibilité en POO Python : les méthodes et attributs sont lisibles et modifiables depuis les autres classes.
- On peut brouiller l'accès à une méthode ou à un attribut en le nommant `__attributBrouille`.

Visibilité des attributs

- Il n'existe pas de visibilité en POO Python : les méthodes et attributs sont lisibles et modifiables depuis les autres classes.
- On peut brouiller l'accès à une méthode ou à un attribut en le nommant `__attributBrouille`.

velo.py

```
class Velo:
    def __init__(self, marque_, distance_parcourue):
        self.__marque = marque_ # Attribut brouillé
        self.distance_parcourue = distance_parcourue_

    def set_marque(self, m):
        self.__marque = m
    def get_marque(self):
        return self.__marque

v = Velo("Giant", 25000)
print(v.distance_parcourue)  # Affiche 25000
print(v.__marque)            # Erreur
print(v.get_marque())        # Affiche Giant
print(getattr(v, "__marque") # Erreur
```

Sommaire

Héritage simple

- On reprend les classes `Personnage`, `Gaulois`, `Romain`.

Héritage simple

- On reprend les classes `Personnage`, `Gaulois`, `Romain`.
- `Gaulois` et `Romain` héritent de `Personnage`.

Héritage simple

- On reprend les classes `Personnage`, `Gaulois`, `Romain`.
- `Gaulois` et `Romain` héritent de `Personnage`.

Personnage.py

```
class Personnage:
    def __init__(age_ = 0, nom_ = ""):
        self.age = age_
        self.nom = nom_
```

Héritage simple

- On reprend les classes `Personnage`, `Gaulois`, `Romain`.
- `Gaulois` et `Romain` héritent de `Personnage`.

Personnage.py

```
class Personnage:
    def __init__(age_ = 0, nom_ = ""):
        self.age = age_
        self.nom = nom_
```

Gaulois.py

```
class Gaulois(Personnage): # Classe mère entre parenthèses
    def __init__(age_ = 0, nom_ = "", taille_moustache_ = 0):
        super().__init__(age_, nom_)
        self.taille_moustache = taille_moustache_
```

Héritage simple

- On reprend les classes `Personnage`, `Gaulois`, `Romain`.
- `Gaulois` et `Romain` héritent de `Personnage`.

Personnage.py
```
class Personnage:
    def __init__(age_ = 0, nom_ = ""):
        self.age = age_
        self.nom = nom_
```

Gaulois.py
```
class Gaulois(Personnage): # Classe mère entre parenthèses
    def __init__(age_ = 0, nom_ = "", taille_moustache_ = 0):
        super().__init__(age_, nom_)
        self.taille_moustache = taille_moustache_
```

Romain.py
```
class Romain(Personnage): # Classe mère entre parenthèses
    def __init__(age_ = 0, nom_ = "", grade_ = ""):
        super().__init__(age_, nom_)
        self.grade = grade_
```

Héritage multiple

- En Python, une classe peut hériter de plusieurs classes en même temps.

Héritage multiple

- En Python, une classe peut hériter de plusieurs classes en même temps.
- Cela s'appelle l'héritage multiple.

Héritage multiple

- En Python, une classe peut hériter de plusieurs classes en même temps.
- Cela s'appelle l'héritage multiple.

Héritage multiple

- En Python, une classe peut hériter de plusieurs classes en même temps.
- Cela s'appelle l'héritage multiple.

Combattant.py

```
class Combattant:
    def __init__(force_ = 0, arme_ = ""):
        self.force = force_
        self.arme = arme_
```

Héritage multiple

- En Python, une classe peut hériter de plusieurs classes en même temps.
- Cela s'appelle l'héritage multiple.

Combattant.py

```
class Combattant:
    def __init__(force_ = 0, arme_ = ""):
        self.force = force_
        self.arme = arme_
```

IrreductibleGaulois.py

```
class IrreductibleGaulois(Gaulois, Combattant):
    def __init__(nom_ = "", age_ = 0, taille_moustache_ = 0,
                 force_ = 0, arme_ = "Mains nues",
                 potion_magique_ = 0):
        super().__init__(nom_, age_, taille_moustache_)
        super().__init__(force_, arme_)
        self.potion_magique = potion_magique_
```

Surcharge d'opérateurs

Quelques exemples

- On souhaite donner un sens aux opérateurs usuels (`+ - * / print < >`, etc.) pour nos objets.

Surcharge d'opérateurs

Quelques exemples

- On souhaite donner un sens aux opérateurs usuels (`+ - * / print < >`, etc.) pour nos objets.
- Cas simple : on voudrait pouvoir écrire `print(p)` pour afficher les attributs de l'objet p.

ville.py

```
class Ville:
    # …
    def __str__(self):
        s = f"{self.nom}({self.lat}, {self.lng}), {self.nbhab},
            {self.surf} km^2"
        return s
```

Surcharge d'opérateurs

Quelques exemples

- On souhaite donner un sens aux opérateurs usuels (`+ - * / print < >`, etc.) pour nos objets.
- Cas simple : on voudrait pouvoir écrire `print(p)` pour afficher les attributs de l'objet p.

ville.py

```
class Ville:
    # …
    def __str__(self):
        s = f"{self.nom}({self.lat}, {self.lng}), {self.nbhab},
            {self.surf} km^2"
        return s
```

- On souhaite définir l'opérateur − (soustraction !) pour qu'il calcule la distance entre deux villes.

Surcharge d'opérateurs

Quelques exemples

- On souhaite donner un sens aux opérateurs usuels (`+ - * / print < >`, etc.) pour nos objets.
- Cas simple : on voudrait pouvoir écrire `print(p)` pour afficher les attributs de l'objet p.

ville.py

```
class Ville:
    # …
    def __str__(self):
        s = f"{self.nom}({self.lat}, {self.lng}), {self.nbhab},
            {self.surf} km^2"
        return s
```

- On souhaite définir l'opérateur - (soustraction !) pour qu'il calcule la distance entre deux villes.

ville.py

```
class Ville:
    # …
    def __sub__(self, autre_ville):
        return self.distance_a(autre_ville)
print(p-r) # Distance Paris - Rouen
```

Surchage d'opérateurs

Opérateurs de comparaison

Opérateur	Méthode associée
`==`	`__eq__(self, objet_a_comparer)`
`!=`	`__ne__(self, objet_a_comparer)`
`>`	`__gt__(self, objet_a_comparer)`
`>=`	`__ge__(self, objet_a_comparer)`
`<`	`__lt__(self, objet_a_comparer)`
`<=`	`__le__(self, objet_a_comparer)`

Surchage d'opérateurs

Opérateurs de comparaison

Opérateur	Méthode associée
==	`__eq__(self, objet_a_comparer)`
!=	`__ne__(self, objet_a_comparer)`
>	`__gt__(self, objet_a_comparer)`
>=	`__ge__(self, objet_a_comparer)`
<	`__lt__(self, objet_a_comparer)`
<=	`__le__(self, objet_a_comparer)`

combattant.py

```
class Combattant:
    # …
    __lt__(self, adversaire):
        return self.force<adversaire.force
```

Surchage d'opérateurs

Opérateurs arithmétiques

Opérateur	Méthode associée
`+`	`__add__(self, objet_a_additionner)`
`-`	`__sub__(self, objet_a_soustraire)`
`*`	`__mul__(self, objet_a_multiplier)`
`/`	`__truediv__(self, objet_diviseur)`
`//`	`__floordiv__(self, objet_diviseur)`
`%`	`__mod__(self, objet_diviseur)`

Surchage d'opérateurs

Opérateurs arithmétiques

Opérateur	Méthode associée
+	`__add__(self, objet_a_additionner)`
-	`__sub__(self, objet_a_soustraire)`
*	`__mul__(self, objet_a_multiplier)`
/	`__truediv__(self, objet_diviseur)`
//	`__floordiv__(self, objet_diviseur)`
%	`__mod__(self, objet_diviseur)`

ville.py

```
class Ville:
    # …
    def __add__(self, autre_ville):
        return self.surf + autre_ville.surf
print(p+r) # 104.4
```