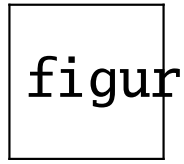
UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS



Cours : Classes et objets en Python

Python : objets et classes

présenté par

Revekka Kyriakoglou

le

15 février 2026

Plan du cours

1 Objectifs et organisation

2 Objets en Python

3 Définir une classe : attributs et méthodes
- Structure minimale

Objectifs

Objectifs

- Comprendre **ce qu'est un objet** en Python (tout est objet).
- Savoir **définir une classe** et instancier des objets.
- Maîtriser `__init__`, attributs, méthodes, `self`.
- Comprendre **encapsulation** (propriétés), **composition** et **héritage**.
- Appliquer à un mini-cas **linguistique** (tokens, corpus, concordance).

Tout est objet

Idée

En Python, nombres, chaînes, listes, fonctions... sont des **objets**. Ils ont :

- un **type** (classe),
- des **attributs** et des **méthodes**.

```
x = 3
s = "chat"
L = [1, 2, 3]

print(type(x), type(s), type(L))
print(s.upper())   # méthode de str
print(L.append(4)) # méthode de list (modifie en place)
```

Attributs, méthodes, introspection

```
s = "linguistique"
print(s.upper)      # référence à une méthode
print(s.upper())    # appel

print(dir(s)[:10]) # liste partielle des attributs/méthodes
```

Attributs, méthodes, introspection

```python
s = "linguistique"
print(s.upper)      # référence à une méthode
print(s.upper())    # appel

print(dir(s)[:10]) # liste partielle des attributs/méthodes
```

Rappel : une méthode est une fonction **attachée à un objet**.

Exercice 1

Dans l'interpréteur Python :

1. Créez `s = "Le chat dort"`.
2. Affichez `type(s)` et `dir(s)` (ou une tranche `dir(s)[:20]`).
3. Trouvez une méthode pour tester si une chaîne commence par `"Le"` et utilisez-la.

Exercice 1

Dans l'interpréteur Python :

1. Créez `s = "Le chat dort"`.
2. Affichez `type(s)` et `dir(s)` (ou une tranche `dir(s)[:20]`).
3. Trouvez une méthode pour tester si une chaîne commence par `"Le"` et utilisez-la.

Méthode utile : `s.startswith("Le")`.

Classe : définition minimale

Syntaxe

- `class Nom:` définit une classe.
- `__init__` initialise l'objet à la création.
- `self` désigne l'instance courante.

```
class Token:
    def __init__(self, forme):
        self.forme = forme

t = Token("chat")
print(t.forme)
print(type(t))
```

`__init__` : le constructeur (ce qui se passe à la création)

Idée

Quand on écrit `t = Token("chat")`, Python :

1. crée un nouvel objet `Token`
2. appelle automatiquement `Token.__init__` pour **l'initialiser**

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

t = Token("chat")  # <-- appelle __init__ automatiquement
```

`__init__` : le constructeur (ce qui se passe à la création)

Idée

Quand on écrit `t = Token("chat")`, Python :

1. crée un nouvel objet `Token`
2. appelle automatiquement `Token.__init__` pour **l'initialiser**

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

t = Token("chat")  # <-- appelle __init__ automatiquement
```

`__init__` ne crée pas l'objet, il l'initialise (donne ses attributs, vérifie des choses, etc.).

`self` : l'objet courant

Qu'est-ce que `self` ?

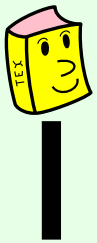
`self` est **le paramètre qui reçoit l'objet** sur lequel on travaille.

```python
class Token:
    def __init__(self, forme):
        self.forme = forme  # attribut stocké DANS l'objet self

t1 = Token("chat")
t2 = Token("chien")

print(t1.forme)  # "chat"
print(t2.forme)  # "chien"
```

`self` : l'objet courant



Qu'est-ce que `self` ?

`self` est **le paramètre qui reçoit l'objet** sur lequel on travaille.

```python
class Token:
    def __init__(self, forme):
        self.forme = forme  # attribut stocké DANS l'objet self

t1 = Token("chat")
t2 = Token("chien")

print(t1.forme)  # "chat"
print(t2.forme)  # "chien"
```

Chaque objet a ses propres attributs : `t1.forme` et `t2.forme` peuvent être différents.

Lien entre `t.methode(...)` et `Classe.methode(t, ...)`

 Quand vous écrivez `t.longueur()`, Python passe `t` automatiquement comme premier argument.

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def longueur(self):
        return len(self.forme)

t = Token("morphologie")
print(t.longueur())          # appel "normal"
print(Token.longueur(t))     # appel équivalent (self = t)
```

Lien entre `t.methode(...)` et `Classe.methode(t, ...)`


Quand vous écrivez `t.longueur()`, Python passe `t` automatiquement comme premier argument.

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def longueur(self):
        return len(self.forme)

t = Token("morphologie")
print(t.longueur())          # appel "normal"
print(Token.longueur(t))     # appel équivalent (self = t)
```


`self` n'est pas un mot-clé : c'est une convention, mais il faut la respecter.

? Pourquoi `self.forme` et pas juste `forme` ?

Deux niveaux

- `forme` : variable locale (existe seulement dans `__init__`)
- `self.forme` : attribut de l'objet (reste après `__init__`)

```
class Token:
    def __init__(self, forme):
        forme = forme.upper()    # variable locale (disparaît
        ↪  après)
        self.forme = forme       # attribut stocké dans l'objet
t = Token("chat")
print(t.forme)  # "CHAT"
```

Trace mentale : exécution de `Token("chat")`

Étape 1 : création d'un objet vide (instance)
Étape 2 : appel `__init__(self, "chat")`
Étape 3 : affectation `self.forme = "chat"`
Résultat : l'objet stocke un attribut `forme`

```
class Token:
    def __init__(self, forme):
        print("self =", self)
        self.forme = forme
        print("attribut forme =", self.forme)

t = Token("chat")
```

Trace mentale : exécution de `Token("chat")`

Étape 1 : création d'un objet vide (instance)
Étape 2 : appel `__init__(self, "chat")`
Étape 3 : affectation `self.forme = "chat"`
Résultat : l'objet stocke un attribut `forme`

```
class Token:
    def __init__(self, forme):
        print("self =", self)
        self.forme = forme
        print("attribut forme =", self.forme)

t = Token("chat")
```

Vous pouvez utiliser ce type de `print` pour comprendre ce qui se passe pendant l'initialisation.

Exercice 2 : `self` et plusieurs objets

1. Créez deux objets : `t1 = Token("chat")` et `t2 = Token("chien")`.
2. Changez `t1.forme = "CHAT"`.
3. Affichez `t1.forme` et `t2.forme`. Expliquez pourquoi `t2` n'a pas changé.

Exercice 2 : `self` et plusieurs objets

1. Créez deux objets : `t1 = Token("chat")` et `t2 = Token("chien")`.
2. Changez `t1.forme = "CHAT"`.
3. Affichez `t1.forme` et `t2.forme`. Expliquez pourquoi `t2` n'a pas changé.

Conclusion attendue : `self` référence **l'objet courant**, et chaque objet a ses propres attributs.

Ajouter des méthodes

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def longueur(self):
        return len(self.forme)

t = Token("morphologie")
print(t.longueur())
```

Ajouter des méthodes

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def longueur(self):
        return len(self.forme)

t = Token("morphologie")
print(t.longueur())
```

`t.longueur()` équivaut à `Token.longueur(t)` : `self` est passé automatiquement.

Une **méthode** est une fonction définie dans une classe.

- Le **premier paramètre** est `self` (l'objet).
- Les paramètres suivants sont ceux de votre choix.
- `return` renvoie un résultat (sinon `None`).

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

    def prefixe(self, n):
        return self.forme[:n]

    def afficher(self):
        print("Token =", self.forme)  # renvoie None
t = Token("morphologie")
print(t.prefixe(4))  # "morp"
print(t.afficher())  # affiche puis imprime None
```

Deux styles fréquents

- Méthode **pure** : calcule et **retourne** une valeur, sans modifier l'objet.
- Méthode **mutante** : **modifie** les attributs de l'objet (*effet de bord*).

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def normalise(self):
        self.forme = self.forme.lower()  # modifie l'objet

t = Token("Chat")
t.normalise()
print(t.forme)  # "chat"
```

Après l'appel `t.normalise()`, l'objet `t` a changé : ce n'est pas juste un calcul, c'est une modification.

Comparer : méthode pure vs méthode mutante

```python
class Token:
    def __init__(self, forme):
        self.forme = forme
    # pure : renvoie une nouvelle chaine, sans changer self
    def normalisee(self):
        return self.forme.lower()
    # mutante : modifie self
    def normalise(self):
        self.forme = self.forme.lower()
t = Token("Chat")
print(t.normalisee())  # "chat"
print(t.forme)         # "Chat"  (inchangé)
t.normalise()
print(t.forme)         # "chat"  (modifié)
```

Convention utile : nommer la version mutante comme un verbe (`normalise`), et la version pure comme un adjectif/participe (`normalisee`).

Pourquoi c'est important ? (effets de bord)

Risque classique

Si deux variables référencent le **même objet**, une méthode mutante impacte tout le monde.

```
t1 = Token("Chat")
t2 = t1                 # t2 et t1 pointent vers le même objet !

t2.normalise()          # modifie l'objet partagé
print(t1.forme)         # "chat"
print(t2.forme)         # "chat"
```

Pourquoi c'est important ? (effets de bord)

Risque classique

Si deux variables référencent le **même objet**, une méthode mutante impacte tout le monde.

```
t1 = Token("Chat")
t2 = t1                 # t2 et t1 pointent vers le même objet !

t2.normalise()          # modifie l'objet partagé
print(t1.forme)         # "chat"
print(t2.forme)         # "chat"
```

`t2 = t1` ne copie pas : ça crée un **alias** (même objet, deux noms).

Copie (simple) pour éviter les surprises

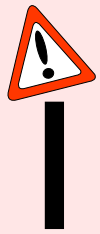
```python
class Token:
    def __init__(self, forme):
        self.forme = forme

    def copie(self):
        return Token(self.forme)  # nouvel objet

t1 = Token("Chat")
t2 = t1.copie()

t2.normalise()
print(t1.forme)  # "Chat"
print(t2.forme)  # "chat"
```

Ici `t2` est un **nouvel objet** : les modifications ne touchent pas `t1`.



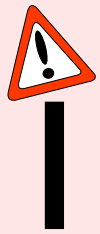
Exemple typique

Si une phrase contient des tokens, normaliser la phrase peut modifier chaque token.

```
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def normalise(self):
        for t in self.tokens:
            t.normalise()   # mutation de chaque token

s = Sentence([Token("Le"), Token("Chat"), Token("Dort")])
s.normalise()
print([t.forme for t in s.tokens])  # ["le","chat","dort"]
```



Exemple typique

Si une phrase contient des tokens, normaliser la phrase peut modifier chaque token.

```
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def normalise(self):
        for t in self.tokens:
            t.normalise()   # mutation de chaque token

s = Sentence([Token("Le"), Token("Chat"), Token("Dort")])
s.normalise()
print([t.forme for t in s.tokens])  # ["le","chat","dort"]
```

---

la méthode change l'état.

Exercice 3 : choisir pure ou mutante

Dans `Token`, ajoutez :

1. `supprime_punct()` : enlève `".,;!?"` de `forme`.
2. Faites **deux versions** :
   - `supprime_punct()` (mutante) qui modifie `self.forme`
   - `sans_punct()` (pure) qui renvoie une nouvelle chaîne

Testez sur `Token("Chat,")` puis sur une liste de tokens dans une phrase.

Exercice 3 : choisir pure ou mutante

Dans `Token`, ajoutez :

1. `supprime_punct()` : enlève `".,;!?"` de `forme`.
2. Faites **deux versions** :
   - `supprime_punct()` (mutante) qui modifie `self.forme`
   - `sans_punct()` (pure) qui renvoie une nouvelle chaîne

Testez sur `Token("Chat,")` puis sur une liste de tokens dans une phrase.

Aide : vous pouvez faire une petite boucle sur les caractères et garder ceux qui ne sont pas dans l'ensemble de ponctuation.

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

    # mutante : modifie self.forme
    def supprime_punct(self):
        ponct = {".", ",", ";", "!", "?"}
        nouvelle = ""
        for c in self.forme:
            if c not in ponct:
                nouvelle += c
        self.forme = nouvelle

    # pure : renvoie une nouvelle chaîne, sans modifier l'objet
    def sans_punct(self):
        ponct = {".", ",", ";", "!", "?"}
        nouvelle = ""
        for c in self.forme:
            if c not in ponct:
                nouvelle += c
        return nouvelle
```

Tests : sur un token puis sur une phrase (liste de tokens)

```python
# Test 1 : sur un token
t = Token("Chat,")
print(t.sans_punct())  # "Chat"
print(t.forme)         # "Chat," (inchangé)

t.supprime_punct()
print(t.forme)         # "Chat"  (modifié)

# Test 2 : sur une liste de tokens (phrase)
tokens = [Token("Le"), Token("Chat,"), Token("dort!")]
# pure : ne modifie pas, construit une nouvelle liste
print([tok.sans_punct() for tok in tokens])
print([tok.forme for tok in tokens])

# mutante : modifie les objets
for tok in tokens:
    tok.supprime_punct()
print([tok.forme for tok in tokens])
```