

Organisation & POO

Python B2 J1 - 2025/2026

Rappels structurants et Programmation Orientée Objet pour étudiants niveau intermédiaire

Colin Tenaguillo

formateur_colin.tenaguilo@supdevinci-edu.fr



Objectifs de la matinée



Réactivation des concepts

Révision des fondamentaux avec des exemples concrets



Bonnes pratiques

Organisation professionnelle du code et structure de projets



Gestion des dépendances

Maîtrise des requirements.txt et environnements virtuels

Rappels express - Fonctions

Bonnes pratiques

- Typage des paramètres
- Valeurs de retour explicites
- Gestion des cas limites

```
# Définition avec typage optionnel
def calculer_moyenne(notes: list[float]) -> float:
    if len(notes) == 0:
        return 0.0
    return sum(notes) / len(notes)

# Appel
resultat = calculer_moyenne([15.5, 12.0, 18.5])
```

Rappels express - Modules

Import simple

```
import math  
print(math.sqrt(25))
```

Import spécifique

```
from datetime import  
datetime  
print(datetime.now())
```

Import avec alias

```
import pandas as pd  
df = pd.DataFrame()
```

Structure d'un projet Python

Organisation recommandée

Chaque dossier a un rôle spécifique dans l'architecture du projet.

- **src/** : Code source principal
- **tests/** : Tests unitaires
- **data/** : Fichiers de données
- **docs/** : Documentation

```
mon_projet/  
├── src/  
│   ├── __init__.py  
│   └── main.py  
├── tests/  
│   └── test_main.py  
├── data/  
│   └── input.csv  
├── docs/  
├── requirements.txt  
└── README.md
```

Fichier requirements.txt

Pourquoi l'utiliser ?

- Reproductibilité des environnements
- Gestion des versions
- Installation automatisée

```
# Exemple de fichier requirements.txt
flask==2.3.3
pandas>=1.5.0
requests
numpy
```

Utilisez `==` pour fixer une version ou `>=` pour une version minimale.

Environnements virtuels

1 Création

```
python -m venv mon_env
```

2 Activation Windows

```
mon_env\Scripts\activate
```

3 Activation Mac/Linux

```
source mon_env/bin/activate
```

4 Installation

```
pip install -r requirements.txt
```

Bonnes pratiques de code

T

Noms explicites

Utilisez des noms de variables clairs et descriptifs plutôt que des abréviations.



Commentaires utiles

Expliquez le "pourquoi", pas le "comment". Évitez les commentaires évidents.

$f(x)$

Fonctions courtes

Une fonction doit avoir une seule responsabilité et tenir sur un écran.



Documentation

Utilisez les docstrings pour documenter vos fonctions et classes.

Exercice 1: Restructuration de projet

Fichiers fournis

- calculs.py (fonctions mathématiques)
- main.py (script principal)
- data.json (données)
- test_calc.py (tests)

Tâches à réaliser

1. Créer l'arborescence professionnelle
2. Déplacer les fichiers aux bons endroits
3. Créer requirements.txt et README.md



Solution type - Structure finale

```
projet_organise/  
├── src/  
│   ├── __init__.py  
│   ├── main.py  
│   └── calculs.py  
├── tests/  
│   └── test_calculs.py  
├── data/  
│   └── data.json  
├── requirements.txt  
└── README.md
```

Points clés

- Séparation logique des responsabilités
- Tests isolés du code source
- Documentation présente
- Gestion des dépendances

Programmation Orientée Objet

Découvrons les concepts fondamentaux de la POO en Python

Pourquoi la POO ?

Modularité

Code réutilisable et organisé en blocs logiques indépendants

Encapsulation

Masquer la complexité interne et exposer une interface simple

Modélisation

Représenter des objets du monde réel dans votre code

Maintenance

Code plus facile à maintenir, déboguer et faire évoluer

Concepts fondamentaux

Classe
Plan de construction, modèle
pour créer des objets

Méthodes
Fonctions qui définissent le
comportement de l'objet



Objet
Instance concrète d'une classe,
entité unique en mémoire

Attributs
Variables qui stockent les
données de l'objet

Première classe - Syntaxe

```
class Voiture:
    # Constructeur
    def __init__(self, marque: str, couleur: str):
        self.marque = marque    # Attribut
        self.couleur = couleur # Attribut

    # Méthode
    def demarrer(self):
        print(f"La {self.marque} démarre !")


# Instanciation
ma_voiture = Voiture("Peugeot", "bleue")
ma_voiture.demarrer()
```

Le constructeur `__init__` initialise les attributs lors de la création de l'objet.

Le mot-clé self

Qu'est-ce que self ?

- Représente l'**instance courante**
- Obligatoire comme premier paramètre
- Permet d'accéder aux attributs et méthodes

 **Rappel** : self n'est qu'une convention. Vous pourriez l'appeler autrement, mais respectez cette pratique !

Exercice 2 : Classe CompteBancaire

```
# À compléter
class CompteBancaire:
    def __init__(self, titulaire: str, solde_initial: float = 0):
        # Votre code ici
        pass

    def deposer(self, montant: float):
        # Votre code ici
        pass

    def retirer(self, montant: float):
        # Votre code ici
        pass
```

Implémentez une classe pour gérer un compte bancaire avec dépôts et retraits sécurisés.

Héritage - Concept

```
class Animal:
    def __init__(self, nom: str):
        self.nom = nom

    def parler(self):
        pass

class Chien(Animal): # Héritage
    def parler(self):
        return "Woof!"

class Chat(Animal): # Héritage
    def parler(self):
        return "Miaou!"
```

L'héritage permet de créer des classes spécialisées basées sur une classe parent.

Héritage - super()

Utilisation de super()

La fonction `super()` permet d'appeler les méthodes de la classe parent.

- Évite la duplication de code
- Maintient la cohérence
- Facilite la maintenance

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

```
class VoitureElectrique(Vehicule):  
    def __init__(self, marque: str, autonomie: int):  
        super().__init__(marque)  
        self.autonomie = autonomie
```

Encapsulation



Public

Accessible de partout : `attribut`



Protégé

Convention interne : `_attribut`



Privé

Name mangling : `__attribut`

```
class CompteBancaire:
    def __init__(self, solde: float, iban: str):
        self.__solde = solde # Attribut privé
        self._iban = iban # Attribut protégé

    def get_solde(self): # Getter
        return self.__solde
```

Exercice 3 : Héritage - Véhicules

Vehicule

marque, modèle

Voiture

+ nombre_portes

Moto

+ type_moteur

Créez une hiérarchie d'héritage pour modéliser différents types de véhicules.

Méthodes spéciales

```
class Livre:
    def __init__(self, titre: str):
        self.titre = titre

    def __str__(self):
        return f"Livre: {self.titre}"

    def __len__(self):
        return len(self.titre)

    def __eq__(self, other):
        if isinstance(other, Livre):
            return True
        return False

mon_livre = Livre("Python avancé")
print(mon_livre)    # Utilise __str__
print(len(mon_livre)) # Utilise __len__
```

Méthodes courantes

- `__str__`: Représentation textuelle
- `__len__`: Longueur de l'objet
- `__eq__`: Comparaison d'égalité

TP - Modélisation d'une bibliothèque



Classe Livre

Attributs : titre, auteur, ISBN



Classe LivreNumerique

Hérite de Livre + attribut taille_fichier



Classe Bibliothèque

Nom, Liste de livres + méthodes d'ajout, suppression par isbn, recherche par titre, recherche par auteur.

Récapitulatif de la journée

Organisation

Structure professionnelle des projets Python

Environnements

Gestion des dépendances et isolation

POO

Classes, objets, héritage et encapsulation

Pratique

Exercices concrets et mise en application

Bravo !

Vous maîtrisez maintenant



L'organisation
professionnelle

Structuration des projets Python
selon les meilleures pratiques



La programmation orientée
objet

Création de classes robustes avec
héritage et encapsulation



Les bonnes pratiques

Code propre, documenté et
maintenable

Prochaine étape : Continuez à pratiquer ces concepts dans vos projets personnels !