Les Fondamentaux de Python pour le BTS SIO SLAM

Un cours complet pour maîtriser Python - de la syntaxe de base à la programmation orientée objet avancée

Formation destinée aux étudiants de BTS SIO option SLAM (2ème année)

David DONISA

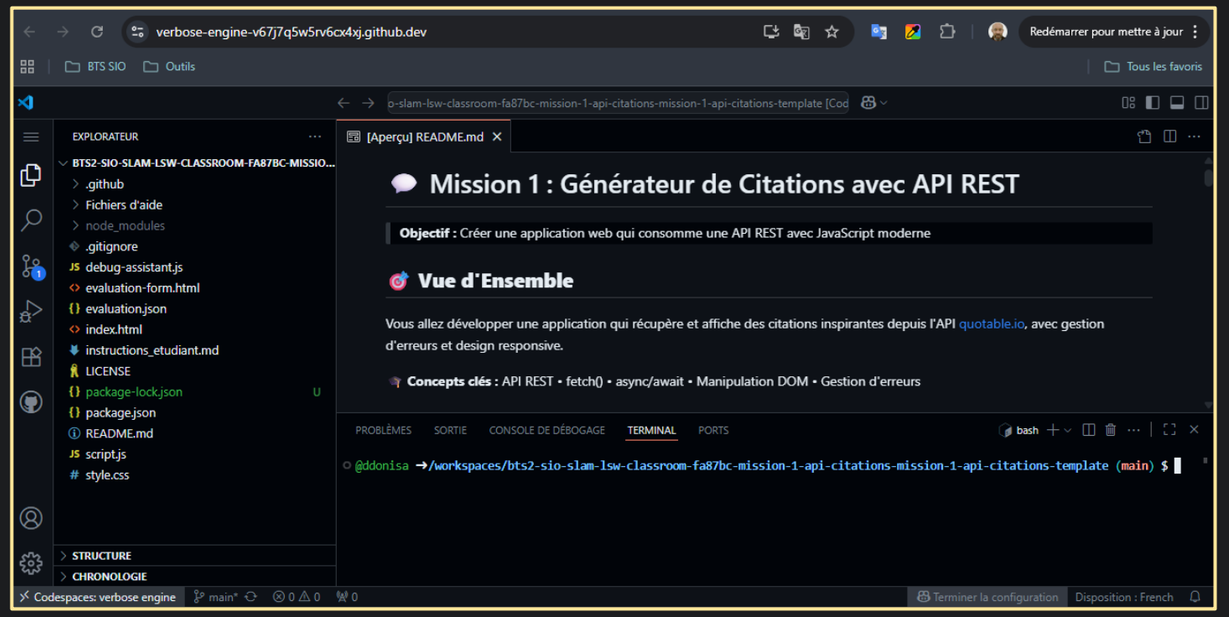
Python et son écosystème

Présentation de Python

- Langage interprété, à typage dynamique
- Créé par Guido van Rossum en 1991
- Versions majeures: Python 2 (obsolète) et Python 3
- Python 3.13/3.14: versions récentes avec gains de performance

Notre environnement de développement

- GitHub Codespaces avec VS Code en ligne
- Avantages: configuration uniforme, accessible partout
- Extensions recommandées: Python, Pylance, GitLens



Structure d'un projet Python

Organisation des fichiers

Structure recommandée pour les projets Python:

```
mon_projet/
├── main.py           # Point d'entrée
├── requirements.txt  # Dépendances
├── README.md         # Documentation
├── mon_package/      # Package Python
│   ├── __init__.py   # Rend le dossier importable
│   ├── module1.py    # Modules Python
│   └── module2.py
└── tests/            # Tests unitaires
    ├── __init__.py
    └── test_module1.py
```

Bonnes pratiques

- Séparation du code en modules logiques
- Utilisation de virtualenv pour isoler les dépendances
- Documentation claire (docstrings, README)
- Tests unitaires
- Respect des conventions PEP 8

Lors de ce cours, nous créerons progressivement un projet complet suivant cette structure.

Syntaxe de base de Python

Variables

```
# Déclaration et typage dynamique
nom = "Alice"       # str
age = 20            # int
moyenne = 15.5      # float
est_present = True  # bool

# Python détermine le type
automatiquement
print(type(nom))
```

Structures conditionnelles

```
# if, elif, else
if age >= 18:
    print("Majeur")
elif age >= 16:
    print("Presque majeur")
else:
    print("Mineur")

# Opérateur ternaire
statut = "Admis" if moyenne >= 10
else "Ajourné"
```

Structures itératives

```
# Boucle for
for i in range(5):
    print(i)  # Affiche 0, 1, 2, 3,
4

# Boucle while
compteur = 0
while compteur < 3:
    print(compteur)
    compteur += 1
```

La syntaxe Python privilégie la lisibilité avec l'indentation obligatoire et une approche minimaliste (pas de point-virgule, pas d'accolades).

Exercice: Syntaxe de base

Exercice 1: Calcul de moyenne

Écrivez un programme qui:

1. Demande à l'utilisateur de saisir 3 notes (utilisez `input()`)
2. Calcule la moyenne de ces notes
3. Affiche "Admis" si la moyenne est ≥ 10, sinon "Refusé"
4. Ajoute une mention selon la moyenne:
   - ≥ 16: "Très bien"
   - ≥ 14: "Bien"
   - ≥ 12: "Assez bien"

Solution

```
# Saisie des notes
note1 = float(input("Note 1: "))
note2 = float(input("Note 2: "))
note3 = float(input("Note 3: "))

# Calcul de la moyenne
moyenne = (note1 + note2 + note3) / 3

# Décision admission
resultat = "Admis" if moyenne >= 10 else "Refusé"

# Attribution mention
if moyenne >= 16:
    mention = "Très bien"
elif moyenne >= 14:
    mention = "Bien"
elif moyenne >= 12:
    mention = "Assez bien"
else:
    mention = "Pas de mention"

# Affichage résultats
print(f"Moyenne: {moyenne:.2f}")
print(f"Résultat: {resultat}")
print(f"Mention: {mention}")
```

Fonctions et Modules

Fonctions en Python

```
# Définition d'une fonction
def calculer_moyenne(notes):
    """Calcule la moyenne d'une liste de notes."""
    return sum(notes) / len(notes)

# Fonction avec valeurs par défaut
def saluer(nom, message="Bonjour"):
    return f"{message}, {nom}!"

# Fonction avec nombre variable d'arguments
def somme(*nombres):
    return sum(nombres)

# Appels de fonctions
mes_notes = [12, 15, 18]
print(calculer_moyenne(mes_notes))  # 15.0
print(saluer("Marie"))              # Bonjour, Marie!
print(somme(1, 2, 3, 4))            # 10
```

Un module est un fichier `.py` contenant du code réutilisable.Un package est un dossier contenant plusieurs modules avec un fichier `__init__.py`.
La difference avec un script classique , c'est qu'il n'a pas nécessairement de section :
if __name__ == '__main__':

Modules et Packages

```
# Import d'un module entier
import math
print(math.sqrt(16))  # 4.0

# Import sélectif
from datetime import datetime
print(datetime.now())

# Import avec alias
import numpy as np
tableau = np.array([1, 2, 3])

# Création d'un module (fichier calculs.py)
def addition(a, b):
    return a + b

# Import de notre module
from calculs import addition
```

Quand le module n'en a pas , il est destine uniquement à être importé et réutilisé
Quand en a une , il agit alors comme un script python classique

Les conteneurs en Python

Listes (mutable)

```
# Création
nombres = [1, 2, 3, 4, 5]
mixte = [1, "deux", 3.0, True]

# Manipulation
nombres.append(6)         # Ajout
nombres.pop(1)            # Suppression
nombres = 10              # Modification
sous_liste = nombres[1:3] # Découpage
```

Tuples (immutable)

```
# Création
point = (10, 20)
personne = ("Jean", 25, "Paris")

# Accès
x, y = point              # Déballage
print(personne)           # "Jean"

# Immuabilité
# point = 5               # Erreur!
```

Sets (mutable, unique)

```
# Création
couleurs = {"rouge", "bleu", "vert"}
nombres = set([1, 2, 2, 3])  # {1, 2, 3}

# Opérations
couleurs.add("jaune")        # Ajout
couleurs.remove("rouge")     # Suppression
union = couleurs | {"noir"}  # Union
```

Dictionnaires (mutable)

```
# Création
etudiant = {
    "nom": "Dupont",
    "prenom": "Pierre",
    "notes": [12, 15, 18]
}

# Manipulation
etudiant["age"] = 20                    # Ajout
del etudiant["notes"]                   # Suppression
print(etudiant.get("ville", "Inconnu"))
```

La mutabilité détermine si un objet peut être modifié après sa création. Les listes, sets et dictionnaires sont mutables. Les tuples et strings sont immutables.

La Programmation Orientée Objet en Python

Classes et instances

```
class Etudiant:
    """Classe représentant un étudiant."""

    # Variable de classe
    ecole = "BTS SIO"

    # Constructeur
    def __init__(self, nom, prenom, age=18):
        # Attributs d'instance
        self.nom = nom
        self.prenom = prenom
        self.age = age
        self._notes = []  # Attribut "protégé"

    # Méthode d'instance
    def ajouter_note(self, note):
        self._notes.append(note)

    def calculer_moyenne(self):
        if not self._notes:
            return 0
        return sum(self._notes) / len(self._notes)

    # Propriété (getter/setter)
    @property
    def notes(self):
        return self._notes.copy()

    # Méthode de classe
    @classmethod
    def changer_ecole(cls, nouvelle_ecole):
        cls.ecole = nouvelle_ecole

# Création d'instances
alice = Etudiant("Durand", "Alice", 20)
bob = Etudiant("Martin", "Bob")

# Utilisation
alice.ajouter_note(15)
```

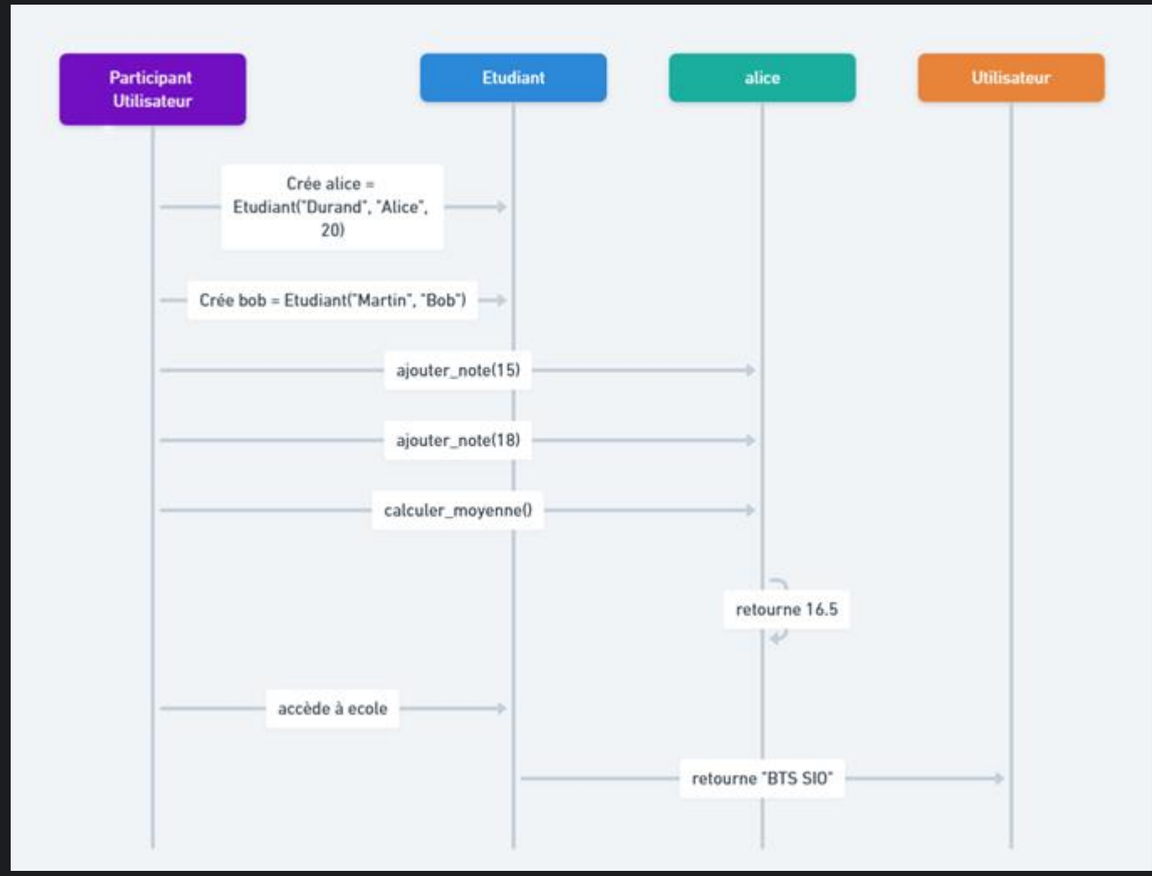


Diagramme UML de séquence

Terminologie POO :

- Classe: modèle/plan définissant structure et comportement
- Instance/Objet: réalisation concrète d'une classe
- Attribut: variable associée à une classe/instance
- Méthode: fonction associée à une classe

Héritage et polymorphisme

1

Héritage simple

```
class Personne:
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def se_presenter(self):
        return f"Je suis {self.prenom} {self.nom}"

class Etudiant(Personne):
    def __init__(self, nom, prenom, numero_etudiant):
        # Appel du constructeur parent
        super().__init__(nom, prenom)
        self.numero_etudiant = numero_etudiant

    # Redéfinition (polymorphisme)
    def se_presenter(self):
        return f"{super().se_presenter()} (N°{self.numero_etudiant})"
```

2

Héritage multiple

```
class Stagiaire:
    def __init__(self, entreprise):
        self.entreprise = entreprise

    def lieu_stage(self):
        return f"Stage chez {self.entreprise}"

class EtudiantStagiaire(Etudiant, Stagiaire):
    def __init__(self, nom, prenom, numero, entreprise):
        Etudiant.__init__(self, nom, prenom, numero)
        Stagiaire.__init__(self, entreprise)

    def se_presenter(self):
        return f"{Etudiant.se_presenter(self)}, {self.lieu_stage()}"
```

Méthodes magiques et gestion des exceptions

Méthodes magiques (dunder methods)

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Représentation sous forme de chaîne
    def __str__(self):
        return f"Vecteur({self.x}, {self.y})"

    # Représentation pour développeurs
    def __repr__(self):
        return f"Vecteur({self.x}, {self.y})"

    # Addition de vecteurs (v1 + v2)
    def __add__(self, autre):
        return Vecteur(self.x + autre.x, self.y + autre.y)

    # Comparaison d'égalité (v1 == v2)
    def __eq__(self, autre):
        return self.x == autre.x and self.y == autre.y

    # Accès comme un conteneur (v, v)
    def __getitem__(self, index):
        if index == 0:
            return self.x
        elif index == 1:
            return self.y
        raise IndexError("Index hors limites")
```

Gestion des exceptions

```
def division_securisee(a, b):
    try:
        resultat = a / b
        return resultat
    except ZeroDivisionError:
        print("Division par zéro impossible!")
        return None
    except TypeError:
        print("Types incompatibles!")
        return None
    finally:
        print("Opération terminée")

# Création de nos propres exceptions
class MoyenneInvalideError(Exception):
    def __init__(self, valeur):
        self.valeur = valeur
        super().__init__(f"Moyenne invalide: {valeur}")

def verifier_moyenne(moyenne):
    if not 0 &lt;= moyenne &lt;= 20:
        raise MoyenneInvalideError(moyenne)
    return True

# Utilisation
try:
    verifier_moyenne(25)
except MoyenneInvalideError as e:
    print(e)  # "Moyenne invalide: 25"
```