POLYTECH SORBONNE

INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET

Millan Mégane
(megane.millan@inria.fr)
2024 - 2025

PRÉSENTATION DU COURS

Objectif du cours

- Savoir utiliser git et les environnements virtuels
- Comprendre et maîtriser les concepts de base de python
- Découvrir la Programmation Orientée Objet (POO)

Déroulement

- 4 séances de cours (2h)
- 4 séances de TP (4h)
- 1 séance d'évaluation (1h)

Évaluation

- 60% TP
- 40% QCM

PROGRAMME & CONTENU

Utilisation de git et des environnement de programmation pour python.

Introduction à Python

Introduction à la POO

INTRODUCTION AUX OUTILS DE PROGRAMMATION - GIT ET ENVIRONNEMENT VIRTUEL

GIT

QU'EST-CE QUE GIT ?

- **Définition**
  - Système de contrôle de version distribué qui permet de suivre les modifications dans le code source tout au long du développement d'un projet.

POURQUOI GIT ?

- **Gestion Efficace des Versions**
  - Traçabilité - Chaque changement est sauvegardé.
  - Sauvegarde - Facile de revenir à une version stable du code.
- **Facilitation de la Collaboration**
  - Travail en équipe et fusion facile des différents développement.
- **Expérimentation et Innovation**
  - Branches de Fonctionnalités - Possibilité de créer des branches pour expérimenter de nouvelles fonctionnalités sans risquer de casser le reste du code

Lien vers un Tutoriel git : https://gitlab.inria.fr/git-tutorial/git-tutorial/-/tree/main?ref_type=headst

ENVIRONNEMENT VIRTUEL

ENVIRONNEMENT VIRTUEL ?

- **Définition**
  - Un environnement virtuel est un espace isolé pour un projet Python où vous pouvez installer des paquets spécifiques sans interférer avec les autres projets ou la configuration globale du système

POURQUOI ?

- **Isolation des Dépendances**
  - Eviter les conflits entre différentes versions de paquets utilisés dans différents projets.
- **Reproductibilité**
  - Assurer que d'autres développeurs ou environnementspeuvent reproduire exactement votre environnement.
- **Gestion Facile des Projets**
  - Faciliter la gestion de projets avec des dépendances spécifiques.

INSTALLATION D'UN ENVIRONNEMENT VIRTUEL - CONDA

QU'EST-CE QUE CONDA ?

- Gestionnaire de paquets open-source et un système de gestion d'environnements, qui fonctionne sur **Windows**, **macOS** et **Linux**.
- Installation, mise à jour et gestion des logiciels et des bibliothèques, ainsi que création des environnements virtuels isolés.

POURQUOI CONDA ?

- **Gestion des Paquets facile**
  - Gestion des paquets Python, mais aussi d'autres langages comme R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN, etc.
- **Gestion des Dépendances**
  - Résout automatiquement les dépendances de paquets, évitant les conflits.
- **Large Écosystème**
  - Distribution populaire pour la science des données et le machine learning, avec plus de 7,500 paquets scientifiques et analytiques.

INSTALLATION DE CONDA

Lien vers Installation Conda

INTRODUCTION PYTHON

PYTHON

PYTHON ?

- langage polyvalent et haut niveau
- Syntaxe simple et lisible
- Langage interprété, facilitant le développement rapide

SYNTAXE ?

- chaque instruction occupe une ligne
- Indentation pour différencier les blocs (for, if, while...)
- Autorise la manipulation de type sans déclaration

```
a = 7.5   # déclaration et affectation d'un nombre
b  = 'toto' # affectation d'une chaîne de caractères
if(a < 4):
     print('a is small')
```

EXEMPLE DE CODE PYTHON

Calcul des racines d'un polynôme du second degré

```python
from cmath import sqrt

#Coefficient
a = 2
b = 4
c = -5

# Discriminant
delta = b**2 - 4*a*c

# Calcul des racines
x1 = (-b + sqrt(delta)) / (2*a)
x1 = (b + sqrt(delta)) / (2*a)

print("Les racines sont ", x1, " et ", x2)
```

TYPES DE DONNÉES

- **Type Numérique**
  - Entier (int), flottant (float) et complexe (complex)
- **Type Chaine de Caractères**
  - Délimité par des guillemets simple (') ou double (")
- **Type Booléen**
  - Deux valeurs possibles True ou False

```
a = 2
f = 5.4
c = 4 + 6j
s = "Hello World !"
b = True
```

LES LISTES

- Séquence **ordonnée** de valeur
  - Éléments accessibles par leurs indices (0...n)
  - Taille accessible via la fonction len()

```
l = [1, 7, 3, -1, 9]

print(len(l)) #5
```

LES LISTES

OPÉRATIONS POSSIBLES

- Accès ([])
- Concaténation (+)
- Répétition (*)
- Appartenance (in)
- Comparaison
- Sous-liste ([:])
- Suppression (del)

```
l = [1, 7, 3, -1, 9]

print(l[0])
print(7 in l)
del(l[2])
print(l[0:3]) # équivalent à l[:3]

k = [5, 1]
print(k * 2)
print(l + k)
print(k == [5, 1])
```

Un peu de pratique

Vous êtes un assistant du maire du village médiéval de PyTown. Votre mission est d'aider à gérer les informations sur les habitants, leurs biens et leur participation à la vie du village à travers plusieurs événements. Le maire souhaite une liste des habitants du village et leurs âges pour gérer les festivités annuelles.

- **Création d'une liste d'habitants :**
  - Créez une liste habitants contenant les noms de 5 habitants du village.
  - Ajoutez 2 nouveaux habitants à la liste.
  - Supprimez l'un des habitants qui a quitté le village.
  - Affichez la longueur de la liste.
- **Ajout des âges des habitants :**
  - Créez une liste ages contenant les âges des habitants (assurez-vous qu'elle correspond à la liste habitants).
  - Le maire souhaite organiser une fête pour les habitants âgés de plus de 18 ans. Filtrez la liste des habitants et affichez uniquement ceux qui ont plus de 18 ans.
- **Modification de la liste :**
  - Un nouveau-né vient de naître dans le village, ajoutez son prénom et son âge (0) à la liste.
  - Affichez la liste mise à jour des habitants.

LES TUPLES

- **Séquence ordonnée et non modfialle d'éléments**
  - Les éléments d'un tuple ne peuvent pas être modifiés
  - Définit par ()
    - Parenthèse non obligatoire lorsqu'au moins un élément

```
#Tuple vide
a = ()

# Tuples contenant un seul élément
b = 1,
c = (1,)

# Tuples contenant trois éléments
d = 1, 2, 3
e = (1, 2, 3)
```

LES TUPLES

- Accès aux éléments d'un tuple avec les crochets
  - En lecture seulement, l'accès en écriture est interdit
- Taille d'un tuple obtenue avec la fonction len()
- Parcours avec for or while
- Définition avec parenthèses parfois **obligatoire**
  - dans des appels de fonctions
  - Création d'un tuple vide

```
def sum(values):
 sum = 0
 for v in values:
  sum += v
 return sum

sum(1, 2, 3)    # Erreur
sum((1, 2, 3))  # Correct
```

Un peu de pratique

Chaque habitant du village possède une maison. Le maire souhaite garder une trace des maisons et de leurs localisations dans le village.

1. **Création d'un tuple pour chaque maison :**
   - Créez un tuple pour chaque habitant, contenant son nom, l'adresse de sa maison (un numéro de rue), et la taille de la maison (en mètres carrés). Exemple : ("Alice", "Rue des Oliviers, 5", 100)
2. **Affichage des informations des maisons :**
   - Affichez les informations de chaque maison sous forme de texte : "Alice vit à Rue des Oliviers, 5 dans une maison de 100m^2."
3. **Modification des tuples :**
   - L'un des habitants a agrandi sa maison. Étant donné que les tuples sont immuables, reconvertissez les informations en liste, modifiez la taille de la maison, puis recréez un tuple pour cet habitant.

LES ENSEMBLES

- **Collection non-ordonnée d'éléments distincts**
  - Pas de doublons et pas d'ordre entre les éléments
  - Définit par {}

```
 ens = {54, 6, 8}

print(len(ens))
print(6 in ens)

for el in ens:
    print(el)
```

LES ENSEMBLES

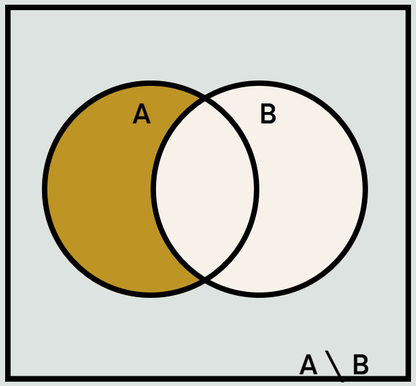
- **Ensemble vide avec set()**
- **Création à paritr d'une séquence**
  - Suppression automatique des doublons
- **Modification d'un ensemble**
  - fonctions add et remove

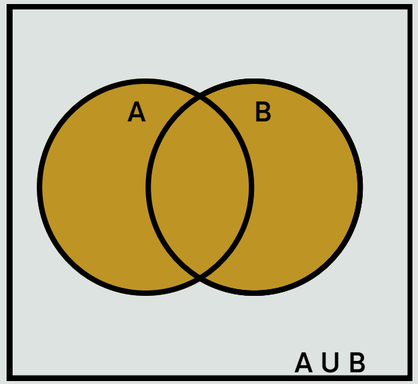
Attention - Element unique = non modifiable

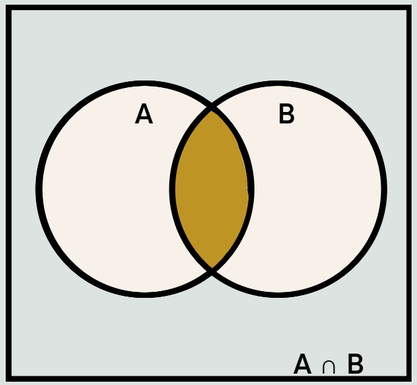
```
G = set('robotique')
F = set()
S = {n for n in range(100) if n%4 == 0}

F.add(5)
F.add(67)
S.remove(4)
```

LES ENSEMBLES

- **Opérations possibles sur les ensembles**
  - Différence, Union et intersection








```
A = {0, 2, 4, 5}
B = {3, 4, 5}

print(A - B)      # {0, 2}
print(A & B)     # {4, 5}
print(A | B)       # {0, 2, 3, 4, 5}
```

Un peu de pratique

Le maire organise une grande fête au village. Il y a plusieurs événements, et il souhaite savoir qui y participe pour éviter les doublons dans les invitations.

1. **Création d'ensembles pour les événements :**
   - Créez deux ensembles event_musique et event_danse, contenant les noms des habitants qui participent respectivement aux événements de musique et de danse.
   - Exemple : event_musique = {"Alice", "Bob", "Claire"} et event_danse = {"Claire", "David", "Eve"}
2. **Manipulation des ensembles :**
   - Affichez les habitants qui participent aux deux événements.
   - Affichez les habitants qui participent à au moins un des événements.
   - Affichez les habitants qui participent uniquement à l'événement de musique.
3. **Ajout et suppression d'éléments :**
   - Un habitant décide de rejoindre l'événement de danse. Ajoutez son nom à event_danse.
   - Un autre habitant décide de ne plus participer à l'événement de musique. Retirez son nom de event_musique.

LES DICTIONNAIRES

- Ensemble de **paires** clé-valeur
  - Les clés sont uniques et non-modifiables
- Définition avec dict() (vide) ou {}

```
dic = {'Luc' : 45, 'Lea' : 34,  'Henri': 12}

print(dic)
print(len(dic))
print(Luc in dic)

map = {i : i for i in range(54, 70)}
```

QUALITÉ DU CODE

- Différence entre code **fonctionnel** et code de **qualité**

- **Pourquoi vouloir un code de qualité ?**
  - Facilité de lecture
  - Mise à jour plus aisée
  - Moins de bug

- Ensemble de règles de bonne pratique

QUALITÉ DU CODE

- **Vérification automatisée de nombreuses règles**
  - Convention de codage
  - Détection d'erreurs (interface, import...)
  - Aide au refactoring (code dupliqué....)
- **Suggestion d'amélioration du code**
- **Interface graphique simple en Tkinter**


```
conda install conda-forge::pylint
```

OK for Anaconda

pylint * for all files, otherwise defien the file asked

Un peu de pratique

Le maire souhaite maintenant gérer les biens des habitants. Chaque habitant possède un certain nombre de vaches, de moutons, et de champs.

1. **Création d'un dictionnaire des biens :**
   - Créez un dictionnaire biens_habitants où chaque clé est le nom d'un habitant et la valeur associée est un sous-dictionnaire contenant :
     - Le nombre de vaches.
     - Le nombre de moutons.
     - Le nombre de champs.
2. **Accès et modification des données :**
   - Le maire souhaite savoir combien de vaches possède chaque habitant. Parcourez le dictionnaire et affichez cette information.
   - Un habitant vient de vendre deux moutons. Mettez à jour le dictionnaire en conséquence.
3. **Ajout de nouveaux habitants :**
   - Un nouvel habitant arrive au village avec des biens. Ajoutez-le au dictionnaire.
4. **Somme totale des biens :**
   - Calculez le nombre total de vaches, de moutons, et de champs dans le village en parcourant le dictionnaire.
   -

TEST UNITAIRES

- **Objectif**
  - Faire un programme qui marche !

- Cycle de développement habituel
  - Coder une fonction
  - Ecrire un main
  - Tester rapidement sans forcément voir les cas limites
  - ...

TEST UNITAIRES

- **Problèmes**
  - Très artisanal
  - Pas du tout exhaustif
  - Test mélangés avec le main

- Les tests sont fondamentaux dans un projet
  - Temps de débug **>** temps d'écriture
  - Isolement des bugs plus tôt
  - Des bons tests peuvent servir d'exemple

TEST UNITAIRES

Objectifs - Détecter les bugs le plus tôt possible dans le cycle de développement

- Tester une nouvelle méthode dès qu'elle est écrite
- Répeter l'ensemble des tests après chaque modification

Question à se poser

- Quel sont les cas limites ? Les cas usuels ?
- Après modification du code, a-t-on toujours le même résultat ?

TEST UNITAIRES

Caractéristiques - Au niveau de l'unité logicielle (ici, une méthode ou une classe)

- Un jeu de test par classe
- Une ou plusieurs fonction de test par méthode
- **Automatique** - On doit pouvoir exécuter l'ensemble des tests automatiquement quand on veut
- **Disponible** - Les tests doivent être fourni avec le code source

TEST UNITAIRES

En Python - plusieurs paquets disponibles (pyunit, unittest...)

- Modules intégrés dans la bibliothèque standard de Python
- Facilite l'écriture et exécution de tests unitaires

```
import unittest

def addition (a , b ) :
  return a + b

class TestAddition ( unittest . TestCase ) :
  def test_addition_entiers ( self ) :
  resultat = addition (2 , 3)
  self . assertEqual ( resultat , 5)

def test_addition_floats ( self ) :
  resultat = addition (2.5 , 3.5)
  self . assertEqual ( resultat , 6.0)
```

Un peu de pratique

Pour les fonctions précédentes, vous allez écrire des tests unitaires.

- **Listes**
  - Test unitaire pour la fonction de filtrage
- **Tuples**
  - Test unitaire pour la fonction modifier_maison
- **Ensemble**
  - Test unitaire pour toutes les fonctions
- **Dictionnaire**
  - Test unitaire pour compter les animaux