Introduction à Python

Planning

Cours avec projet (1.5j)

Présentation projet de groupe (0.5j)

Projet (2.5j)

Review de chaque projet(0.5j)

Modalités d’évaluation

Note de groupe (67%): projet

Note individuelle (33%):

- projet (implication, impact, présentation)
- QCM
- exercice

Sommaire

Introduction

Objectifs:

- compréhension de Python
- écrire du code Python
- savoir installer et préparer un environnement de développement pour Python
- concevoir et développer un projet pratique

Histoire

- publié en 1991
- conçu par Guido Van Rossum
  - néérlandais
  - université d’Amsterdam
  - entreprises: Google, Dropbox, retraite, Microsoft
- débuté pendant une semaine de vacances à Noël
- une référence à Monty Python's Flying Circus

Histoire

- ABC: langage de programmation créé en 1975
- Guido participait au développement de ABC
- Idées d’améliorations -> le rendre interfaçable
- 1989: semaine de vacances -> Python est né
- 1995: travaille au CNRI pour des améliorations
- 2000: Python déménage à Be Open

Point forts

- vaste communauté de développeurs
- bonne documentation
- simplicité
- lisibilité

Utilisation

- Parmi les 3 langages les plus populaires
- Développement de logiciels
  - scripts
  - backend
- Intelligence artificielle (IA)
  - data engineering
  - data science
  - Machine Learning
- très répandu en entreprises → compétence très recherchée

Exemple de code

```
def hello(name):    # function definition
    print("Hello there " + name + "!")

name = "Anakin" # declaration variable
print(hello(name))  # Hello there Anakin!
```

Caractéristiques

- langage interprété
  - langage compilé (Java, C#): code source compilé, converti en code machine par un compilateur
  - langage interprété (Python): code source exécuté ligne par ligne par un interpréteur
  - détection d'erreurs: à l'exécution pour un langage interprété, à la compilation pour un compilé
  - plus de facilité de développement pour un langage interprété
  - exécution plus rapide pour langage compilé (traduction en code machine)
- variables dynamiquement typées
  - le type d'une variable peut changer pendant l'exécution du programme
  - pas de déclaration du type d'une variable
- gestion automatique de la mémoire
- langage orienté objet

```
x = 5   # int
x = 'Anakin'    # String
```

Installation

- version actuelle: Python 3 (3.10)
- download
  - Windows: https://www.python.org/downloads/windows/
  - MacOS:
    - https://www.python.org/downloads/macos/
    - homebrew: `$ brew install python`
  - Linux: command line
    - `$ sudo apt-get update`
    - `$ sudo apt-get install python3.10`
- Vérifier l'installation: `python --version`

IDE

- écrire, éditer, compiler, exécuter et debugger du code
- facilite la programmation
- IDE célèbres pour Python
  - PyCharm
  - VSCode
  - Jupyter Notebook
  - Atom
  - Spyder
- VSCode: https://code.visualstudio.com/download

Premiers pas

1. créer un projet sur VSCode `movie-list`
2. créer un fichier `main.py`
3. coder la fonction `say_hello(name)`
4. appeler la fonction pour plusieurs noms

Remarque: on pourra utiliser la string interpolation pour la fonction

Bases

Variables et types

- variable: sert à stocker des données
- types de données
  - int: entier
  - float: permet de stocker des nombres décimaux
  - str: chaîne de caractères
  - bool: booléen (True ou False)
  - list: liste
- la fonction type renvoie le type d’une donnée
  - exemple: type(“Obi Wan”) → str

Opérations

- Arithmétiques
  - addition, soustraction: +, -
  - multiplication: *
  - modulo: %
  - quotient: //
- Comparaisons
  - égalité: ==
  - comparaison: >, >=, <, <=
- Logiques
  - ET: AND
  - OU: OR
  - NON: NOT
- Inclusion:
  - a IN b : renvoie True si a est contenu dans b (listes, strings)
- Opérations sur les bits
  - & : et
  - | : ou
  - >> et << : décalage des bits

Instructions

- print
- if
- else
- elif
- while
- for
  - range(a): 0, 1, 2, … a-1
  - range(a, b): a, a+1, …, b-1
  - range(a, b, pas): a, a+pas, a+2*pas, …
- break
  - dans une boucle, permet d'en sortie
- continue
  - permet de continuer la boucle en passant l'itération actuelle

Strings

- chaînes de caractères
- propriétés:
  - `len(s)` : longueur de la chaîne s
  - `s[i]` : i ème caractère de s
  - `s + t` : concaténation de la chaîne de caractères s avec t
  - `s in t` : vérifie si la chaîne s est contenue dans la chaîne t
  - `s.split()` : renvoie une liste des mots de s

```
s = 'il fait beau'
mots = s.split()
print(mots)     # ['il', 'fait', 'beau']
```

Exercices

1. Afficher tous les nombres entiers de 0 à 500
2. Somme des 1000 premiers entiers
3. Donné un entier N, afficher si il est pair ou impair
4. Retourner la moyenne des éléments d'une liste
5. Ecrire un programme qui
    a. affiche tous les nombres entre 1 et N compris
    b. s'arrête lorsque l'on dépasse 500
    c. n'affiche pas les multiples de 5
6. Ecrire un programme qui renvoie uniquement les voyelles d'un mot en entrée (renvoyer les voyelles dans l'ordre initial dans le mot)

Structures de données - collections

Listes

- Collection d’éléments: [ ], [1,2,3]
- Indexation de 0 à L-1 avec L la longueur de la liste
- propriétés:
  - `len(L)` : longueur de la liste
  - `L[i]` : élément i de la liste, avec 0 <= i < L
  - `L.append(elem)` : ajoute l’élément à la fin de la liste
  - `del(L[i])` : supprime l’élément i de la liste

Dictionnaires

- appelés aussi HashMap
- associent des clés à des valeurs
- clés: données immuables → int, str
- valeurs: int, str, list, set, ect.
- opérations
  - ajouter des clés / valeurs
  - modifier la valeur d'une clé
  - supprimer une clé
- très utilisés

Dictionnaires

- déclaration : `{}`
- ajout / modification d’une clé : `d[key] = value`
- suppression d’une clé: `del[d[key]]`
- defaultdict:
  - librairie Python
  - permet de spécifier une valeur par défaut pour les clés qui n'existent pas encore dans le dictionnaire
  - gestion d’erreurs pour des clés inexistantes ou inconnues

Sets

- collection non ordonnées d’éléments uniques
- stocke des éléments uniques sans ordre spécifique
- très utile tester l'appartenance d'éléments et éliminer les doublons
- déclaration: `set()`
- opérations
  - `add`: ajoute un élément au set
  - `remove`: supprime un élément du set
  - union de deux sets: `s1.union(s2)`
  - différence de deux sets: `s1 - s2`
  - intersection de deux sets: `s1 & s2`

Tuples

- Similaires aux listes
- immuables: ne peuvent être modifiés
- collection ordonnée d'éléments
- syntaxe en Python
  - `tuple = ()`
  - `tuple = (1, 2, 'toto')`
  - `tuple[0] = 2` ERROR
- concaténation de deux tuples: `tuple1 + tuple2`

Fonctions et modules

Fonctions

- bloc de code réutilisable (qui peut être appelé)
- encapsuler une ou plusieurs instructions
- définir nom avec arguments
- corps
- valeur de retour
    - note: une fonction peut avoir plusieurs valeurs de retour !

```
def nom_de_la_fonction(arguments):
    # Corps de la fonction
    # Instructions
    return valeur
```

Fonctions avec arguments optionnels

Peut offrir de la flexibilité dans certains cas

```
def say_hello(name, hello="Hello"):
    return hello + " " + name + '!'
say_hello("Anakin") # Hello Anakin!
say_hello("Anakin", "Hi") # Hi Anakin!
```

Exercices

1. Définir une fonction `create_movie` qui prend 3 arguments:
   - nom: nom du film
   - genre: genre du film
   - favori: True si le film est favori, False sinon
2. Définir une fonction `add_movie` qui prend un film en entrée et l'ajoute dans la liste des films

   Astuce: on pourra définir la liste des films comme un dictionnaire (nom du film -> film)
3. Définir une fonction `delete_movie` qui prend un nom de film en entrée et supprime le film (on supposera que le film est toujours dans la liste de films)

Exercices

4. Définir une fonction `get_names` qui renvoie une liste alphabétique des noms des films
5. Définir une fonction `display_all` qui affiche tous les film
6. Définir une fonction `get_genres` qui renvoie un dictionnaire avec les genres en clé, et le nombre de film correspondant au genre en valeur
7. Définir une fonction `get_favorites` qui renvoie une liste alphabétique des films favoris
8. Définir une fonction `get_names_by_genre` qui prend entrée un genre et renvoie une liste alphabétique des films pour ce genre

Modules

- lorsque le projet grossit → séparer le code
- un seul fichier unique est ILLISIBLE et difficile à maintenir
- votre code doit être
  - **understandable (compréhensible)**
  - **maintainable (maintenable)**
- penser à importer vos modules !

```
import sys
import movie_list as mv
from movie_list import *
```

Exercice

1. Créer un module `movie.py` qui contient toutes les fonctions précédentes
2. importer le module dans `main.py`

Gestion d’erreurs

Gestion d’erreurs

- gérer les erreurs de code est très important

  → rend le code maintenable

- la gestion des erreurs rend le code fiable

Types d’erreurs

- Erreurs syntaxiques
- Erreurs d’exécution :
  - IndexError
  - KeyError
  - TypeError
  - ValueError
  - ZeroDivisionError

```python
def div(a: int, b: int):
    return a/b

div(5, 0) # ZeroDivisionError
```

Gestion des exceptions

→ L’exception est attrapée dans un try/except

```python
def div(a: int, b: int):
    try:
        res = a / n
        return res
    except ZeroDivisionError as e:
        print("Erreur : Division par zéro n'est pas autorisée.")


div(5, 0) # Erreur : Division par zéro n'est pas autorisée
```

Exercice

Rajouter une gestion d’erreur dans la fonction `delete_movie` si le film n’est pas dans la liste

Documenter le code

- un code maintenable doit être documenté
- commentaires: précédés d'un #
  - toujours commenter les fonctions, noms de variables quand ils ne sont pas clairs

- **Docstring**
  - documentation de fonction
  - placé au début de la fonction
  - fournir une explication claire et explicite
  - 3 guillemets ouvrants et fermants

```
def addition(a, b):
    """
    Cette fonction additionne deux nombres.

    :param a: Premier nombre.
    :param b: Deuxième nombre.
    :return: La somme des deux nombres.
    """
    return a + b
```

Exercice

Documenter toutes les fonctions de `movie.py`

Bibliothèques

Bibliothèques

- ensemble de fonctionnalités à importer
- collection de modules existants en Python, faits par d'autres développeurs
- permet d'éviter d'écrire du code supplémentaire

→ installation avec pip

PIP - Python Install Packages

- Outil de gestion de paquets pour installer et gérer des bibliothèques Python
- installer une lib: `pip install numpy`
- upgrade: `pip install --upgrade`
- la librairie est ensuite installée au niveau système
- et si la bibliothèque doit être utilisée dans différents projets avec des versions différentes?
  - ex: numpy 1.14 en 2018, 1.26 en 2023
  - comment être sûr que chaque projet utilisera sa propre version ?

  → **environnement virtuel**

Environnement virtuel

- outil qui permet de créer un système de fichier indépendant du système global
- avantage: spécifique à chaque projet

```
pip install virtualenv      # Installation de virtualenv
virtualenv venv         # Création d'un environnement virtuel
venv\Scripts\activate      # Activation de l'environnement virtuel (sous Windows)
source venv/bin/activate       # Activation de l'environnement virtuel (sous Linux/Mac)
```

- l'environnement virtuel est ensuite présent à la racine dans le dossier **venv**
- Bonne pratique : fichier requirements.txt
  - lister les dépendance du projet
  - créer rapidement l'environnement virtuel du projet

Exercice

1. Créer le fichier requirements.txt à la racine du projet pour lister les dépendances
2. Créer l’environnement virtuel

Interactions avec les fichiers

Python offre la possibilité d'interagir avec le file system en lecture

```
f = open("nom_fichier.txt", "r")    # ouverture d'un fichier en lecture
f = open("nom_fichier.txt", "w")    # ouverture d'un fihcier en écriture
f.close()   # fermeture d'un fichier
```

```
contenu = f.read()  # lecture de tout le contenu
ligne = f.readline()    # lecture d'une ligne
lignes = f.readlines()  # Lecture de toutes les lignes
```

```
# lecture ligne par ligne
line = f.readline()
print(line)
while line: # Quand line == None, la fin du fichier est atteinte
    line = f.readline()
    print(line)
```

Interactions avec les fichiers

Interaction en écriture dans un fichier

```
with open('myfile.txt','w') as f:
    # Ecriture d'une ligne
    f.write("Nouveau contenu\n")    # Le caractère d'échappement \n indique un saut de ligne
    f.write("Une 2eme ligne\n")
```

Exercice

1. Créer un module `logger.py`
2. Ecrire une fonction `log` qui prend en entrée un message et l'écrit dans un fichier movie.log à la racine du projet
3. Appeler la méthode log dans toutes les fonctions de `movie.py`

CLI

- La CLI (Command Line Interface) permet à l'utilisateur d'interagir avec un programme via des commandes textuelles.
- exemple: ls -l dossier/
  - fournit un argument l à la commande ls
  - la liste des fichiers sera détaillée, notamment avec le type de fichier
- Module `sys` pour les Arguments de la Ligne de Commande :
  - Importation du module `sys` : `import sys`.
  - Accès aux arguments de la ligne de commande sous forme de liste : `arguments = sys.argv`.
  - exemple : `python script.py argument1 argument2`

Exercice

Fournir une CLI qui:

1. affiche les logs avec le paramètre `-log`
2. affiche la liste de tous les films avec le paramètre `-lm`
3. affiche la liste des films favoris avec le paramètre `-fv`
4. affiche la liste de films pour un genre donné avec le paramètre `-g` suivi du nom du genre

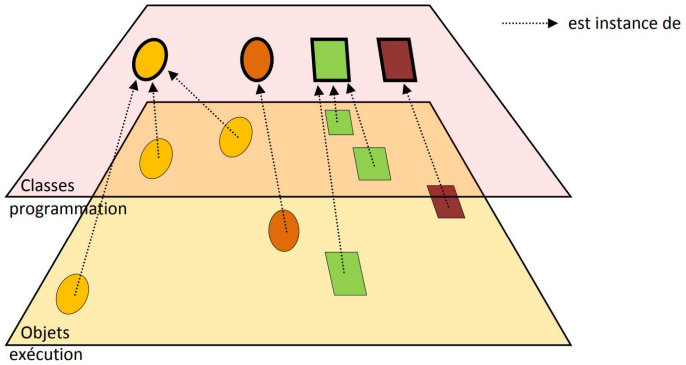
POO - Programmation orientée objet

POO

- programmation orientée objet
- méthode utilisée en programmation pour structurer le code
  - se retrouve dans d’autres langages célèbres (Java, c++, c#)
- très utile et populaire
  - développement de logiciels
  - modélisation du monde réel

Classes et objets

- Classe
  - sert à définir un type concret
  - définie par un nom, la définition des attributs et l'implémentation des méthodes
- Objet
  - instance d'une classe



Classe et objet en Python

- mot clé class
- __init__ : méthode (constructeur) pour initialiser les attributs de l'objet
- possibilité d'ajouter des méthodes à la classe (getters, setters, etc)

```python
class Voiture:

    def __init__(self, marque, modele, annee):
        self.marque = marque
        self.modele = modele
        self.annee = annee

    def afficher_details(self):
        print(f"{self.marque} {self.modele} {self.annee}")
```

Attributs de classe et d’instances

Attribut de classe : partagé par toutes les instances de la classe

Attribut d’instance : propre à chaque instance

```python
class Voiture:

    moteur = "essence"  # attribut de classe

    def __init__(self, marque, modele, annee):
        self.marque = marque    # attribut d'instance
        self.modele = modele    # attribut d'instance
        self.annee = annee  # attribut d'instance

    def afficher_details(self):
        print(f"{self.marque} {self.modele} {self.annee}")
```

Encapsulation

- attribut privé
- double underscore pour la déclaration
- peut être donné en lecture par une méthode publique

```python
class Voiture:

    moteur = "essence"  # attribut de classe

    def __init__(self, marque, modele, annee, proprietaire):
        self.marque = marque    # attribut d'instance
        self.modele = modele    # attribut d'instance
        self.annee = annee  # attribut d'instance
        self.__proprietaire = proprietaire

    def afficher_details(self):
        print(f"{self.marque} {self.modele} {self.annee}")

    def afficher_proprietaire(self):
        print(f"Propriétaire: {self.__proprietaire}")
```

Héritage

- création d'une nouvelle classe (sous-classe) à partir d'une classe existante (super-classe).

→ La sous-classe hérite des attributs et des méthodes de la super-classe.

```
class Animal:

    def __init__(self):
        self.type = type
        self.poids = poids

    def manger(self):
        print("L'animal mange.")

class Chien(Animal):

    def aboyer(self):
        print("Le chien aboie.")
```

Exercices

1. créer une classe `Movie`
2. créer une classe `MovieList`
3. Adapter le code du `main.py` en utilisant les deux classes précédentes

Bonus : tests unitaires

Pourquoi les tests unitaires ?

- définition
  - scénarios isolés
  - automatisés
  - une entrée → une sortie
- vérifier chaque partie isolée du code
- code maintenable
- code vérifié en continu, même après modification

Test unitaire en Python

Les fonctions de test commencent toujours par test !

```python
def sum(a, b):
    return a+b
```

```python
# Unit tests for sum function
import unittest
from sum import sum

class TestSum(unittest.TestCase):
    def test_add_positive_numbers(self):
        result = sum(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        result = sum(-2, 3)
        self.assertEqual(result, 1)

    def test_add_zero(self):
        result = sum(0, 5)
        self.assertEqual(result, 5)

if __name__ == '__main__':
    unittest.main()
```

Exercice

Ecrire des tests unitaires pour les fonctions `get_names` et `get_favorites`