INF205
LANGAGE DE PROGRAMMATION PYTHON

Enseignant : KWETTE NOUMSI
Département Informatique – Université de Yaoundé 1

Octobre 2025

PLAN DE COURS COMPLET

1 Module 1 : Mise en place de l'environnement de travail **24**

1.1 Découvrir Python 24

1.1.1 Histoire de python 24

1.1.2 Les différents domaines d'application de python 25

Python pour l'analyse de données 26

Python pour la visualisation de données - 27

Python pour l'apprentissage automatique - 27

Python pour le développement de logiciels - 28

Python pour le développement Web - 28

Python pour l'automatisation des tâches/la création de scripts - 28

Diverses utilisations de Python - 29

1.1.3 Quels sont les emplois qui nécessitent Python ? 29

1.2 Mise en place de l'environnement de travail 29

1.2.1 Installation de python 30

Installation sous Microsoft Windows 30

Installation sous GNU/Linux 32

1.2.2 Installation d'Anaconda 33

Installation de Windows 33

Installation d'Anaconda sous Linux 38

1.2.3 Installation de PyCharm 39

Installer à l'aide de l'application Toolbox 39

1.3 Prise en main de nos environnements de travail 41

1.3.1 Prise en main de Jupyter Notebook 41

Option 1 : Anaconda Navigator (recommandé débutants) 41
Option 2 : Terminal/Ligne de commande 41
Comprendre l'interface d'accueil 42
Créer votre premier notebook 42
Étape 1 : 42
Étape 2 : 43
Sauvegarde automatique : 43
Étape 3 : 43
Vos premières cellules de code 43
Cellule 1 : Hello World en Python 44
Cellule 2 : Calculs mathématiques 44
Cellule 3 : Utiliser la variable précédente 45
Ajouter du texte avec Markdown 45
1.3.2 Prise en main de PyCharm 46
Note : 47
Créer un projet PyCharm 48
Créer un module de code Python 48
Editer votre script Python 49
Lancer votre script Python 49

2 Module 2 : Fondamentaux du langage Python **52**
2.1 Python et son écosystème 52
2.1.1 Les Forces de Python 53
2.1.2 Paradigmes de Programmation 53
2.1.3 Architecture et Fonctionnement 53
2.1.4 Écosystème Python 54
Les bibliothèques standards : 54
Data Science & IA : 54
Développement Web : 54
Automatisation & Scripting : 54

Cybersécurité & Pentesting : 55
Interfaces Graphiques : 55
Automation / IoT : 55
2.2 Les bases en python 56
2.2.1 Les types natifs de python 56
Les chaînes de caractère : str 57
Modifier une chaîne de caractères 58
Les nombres 58
Les nombres entiers : int 58
Les nombres décimaux : float 60
Les booléens 61
2.2.2 Les variables 63
La portée des variables 64
Assigner une valeur à une variable 65
Les assignations multiples 65
Les assignations parallèles 65
Les constantes 66
Concaténer des variables 66
Règles et conventions de nommage 67
2.2.3 Les types des objets et la conversion 68
Afficher le type d'une variable 68
Les fonction de conversion int et str 68
2.2.4 Interagir avec l'utilisateur 69
2.2.5 Manipuler les chaînes de caractères 70
2.2.6 Les opérateurs en Python 71
Opérateurs arithmétiques 72
Opérateurs d'affectation 72
Opérateurs de comparaison 72
Opérateurs logiques 72

Opérateurs d'appartenance 72
Opérateurs d'identité 73
Opérateurs bit-à-bit 73
La différence entres les opérateurs *is* et == 73
2.2.7 Le formatage des chaînes de caractères 73
Au commencement, il y avait la concaténation 74
La méthode format 75
Les f-string 76
Les caractères spéciaux 77
Quelques options de formatage 77
'Padder' un nombre - 77
Formater une date 78
Tronquer une chaîne de caractères - 78
2.3 TP 78
2.4 Exercices 79

3 Module 3 : Programmation procédurale et Structures de données 83

3.1 Les structures conditionnelles 83
3.1.1 La structure conditionnelle if simple 83
3.1.2 La structure conditionnelle if / else 84
3.1.3 La structure conditionnelle if / elif / else 85
3.1.4 Imbriquer des structures conditionnelles 85
3.1.5 Syntaxe compacte d'une alternative 86
Syntaxe 86
Avantages de l'opérateur ternaire 86
Quand l'utiliser et quand l'éviter 86
3.2 Les boucles 87
3.2.1 La boucle while 87
Syntaxe de la boucle while 87
while...else 88

3.2.2 La boucle for 88
Syntaxe de la boucle for 89
for ... else 89
3.2.3 Les instructions break, continue et pass 90
L'instruction break 90
L'instruction continue 91
L'instruction pass 91
3.3 Les structures de données 92
3.3.1 Les listes 93
Caractéristiques principales des listes 93
Différents éléments possibles 93
Créer une liste vide - 93
Créer une liste avec des valeurs initiales - 94
Créer une liste avec des éléments de types différents - 94
Générer automatiquement des valeurs avec range() - 94
Accéder aux éléments d'une liste - 94
Utilisation de l'indexation négative - 94
Extraire une sous-liste avec le slicing 95
Utilisation du pas (step) dans le slicing - 95
Ajouter et supprimer des éléments - 95
Ajouter un élément avec append() - 95
Insérer un élément à une position spécifique avec insert() - 95
Ajouter plusieurs éléments avec extend() : 95
Supprimer un élément avec remove() - 95
Supprimer un élément avec pop() - 96
Supprimer un élément avec del - 96
Modifier les éléments d'une liste - 96
Modifier un élément spécifique - 96
Modifier plusieurs éléments avec le slicing - 96

Ajouter ou supprimer des éléments avec le slicing - 96
Fonctions intégrées utiles pour les listes - 97
Longueur de la liste avec len() - 97
Trouver la position d'un élément avec index() - 97
Vérifier la présence d'un élément avec in - 97
Compter les occurrences d'un élément avec count() - 97
Trier une liste avec sort() - 97
Inverser une liste avec reverse() - 98
Boucles et itérations sur une liste - 98
Parcourir une liste avec une boucle for - 98
Utiliser enumerate() pour obtenir les index et les éléments - 98
Parcourir une liste avec une boucle while - 98
Listes imbriquées (listes de listes) - 98
Créer une liste imbriquée - 99
Accéder aux éléments d'une liste imbriquée - 99
Parcourir une liste imbriquée avec des boucles 99
Utiliser des compréhensions de liste avec des listes imbriquées - . . . 99
Les compréhensions de liste - 99
Syntaxe de base des compréhensions de liste - 100
Utiliser des conditions dans les compréhensions de liste - 100
Compréhensions de liste imbriquées - 100
Méthodes avancées sur les listes - 100
Copier une liste - 100
Fusionner des listes 100
Utiliser map() et filter() - 100
Bonnes pratiques pour travailler avec les listes 101
Privilégiez les compréhensions de liste pour les opérations simples - 101
Évitez de modifier une liste pendant son parcours - 101

Utilisez les copies de liste pour préserver l'intégrité de la liste d'origine - 101
Attention aux listes imbriquées et aux copies profondes - 102
Optimisez les opérations coûteuses - 102
Limitez les conversions entre listes et autres types de données - . . . 102
Utilisez join() pour assembler des listes de chaînes de caractères - . . 102
Les list comprehension 103
En conclusion 104
TP : Gestion d'une Liste de Courses 104
Énoncé : 104
Étape 1 : Initialiser la Liste 104
Étape 2 : Ajouter un Article 104
Étape 3 : Supprimer un Article 105
Étape 4 : Modifier un Article 105
Étape 5 : Afficher la Liste 106
Étape 6 : Menu Interactif 106
Étape 7 : Gérer les Erreurs d'Entrée 107
Étape 8 : Rendre le Code Plus Lisible avec des Fonctions 107
3.3.2 Les tuples 108
Qu'est-ce que sont les tuples ? 108
Pourquoi utiliser des tuples ? 108
Créer un tuple en Python 108
Accéder aux éléments d'un tuple 109
Accéder aux éléments par index - 109
Indexation négative - 110
Extraction de sous-tuples (slicing) - 110
Accéder aux éléments dans un tuple imbriqué - 110
Erreurs courantes d'indexation - 111
Les opérations de base sur les tuples 111

Concaténation de tuples - 111
Répétition d'un tuple - 111
Vérification de l'appartenance avec in - 111
Utilisation des fonctions intégrées len(), min() et max() 112
Compter les occurrences d'un élément avec count() - 112
Trouver l'index d'un élément avec index() - 112
Comparaison de tuples - 113
Conversion d'un tuple en liste - 113
Les tuples nommés (namedtuples) - 113
Créer un namedtuple - 113
Accéder aux éléments d'un namedtuple - 114
Utilisation pratique des namedtuples - 114
Convertir un namedtuple en dictionnaire - 114
Remplacer les valeurs dans un namedtuple avec $_{r}eplace()-$ 115
Utilisation des tuples pour le retour de multiples valeurs 115
Déballer les valeurs d'un tuple lors du retour - 115
Cas d'usage des tuples pour le retour de valeurs multiples - 116
TP sur les tuples 117
Scénario : 117
Étapes de l'exercice 117
Conclusion 120
3.3.3 Les sets 120
Créer un set vide 121
Créer un set et l'initialiser 121
Les opérations sur les ensembles 122
Connaître la taille d'un ensemble - 122
Savoir si un élément est présent dans un ensemble - 122
Comparer deux ensembles - 122
Connaître le plus grand élément d'un ensemble - 122

Connaître le plus petit élément d'un ensemble - 123
Quelques méthodes pour les ensembles 123
copy() : 123
add() : 123
update(iterable) : 123
Retirer certains éléments d'un set - 123
clear() : 124
union(e) : 124
intersection(e) : 124
difference(e) : 124
symmetric_difference(e) : 125
Les "set comprehension" 125
Les Frozenset 125
Ensemble, liste et tuple 126
3.3.4 Les dictionnaires 126
Création d'un dictionnaire 126
Accéder aux valeurs dans un dictionnaire 128
Gérer l'absence de clé avec get() 128
Ajouter des éléments 128
Ajouter un nouvel élément : 129
Modifier un élément existant 129
Supprimer des éléments d'un dictionnaire 129
Supprimer un élément avec pop() : 129
Supprimer un élément avec del : 130
Vider complètement un dictionnaire avec clear() : 130
Méthodes courantes sur les dictionnaires 130
Obtenir les clés avec keys() : 130
Obtenir les valeurs avec values() : 131
Obtenir les paires clé-valeur avec items() : 131

3.3.5 Compréhension de dictionnaire 131
Syntaxe de base : 131
Filtrer les éléments avec une condition : 132
Transformer les valeurs : 132
Dictionnaires imbriqués 132
Création d'un dictionnaire imbriqué : 132
Accéder aux éléments dans un dictionnaire imbriqué : 133
Modifier un élément dans un dictionnaire imbriqué : 133
Parcourir un dictionnaire imbriqué : 133
Manipuler des dictionnaires avec des fonctions avancées 134
Utiliser dict() pour créer un dictionnaire : 134
Fusionner des dictionnaires avec update() : 134
Créer un dictionnaire avec zip() : 134
Fusionner plusieurs dictionnaires avec ** 135
Utiliser des expressions générées par dictionnaire 135
TP sur les dictionnaires 135
Scénario : 135
Étapes des TP 136
Bonnes pratiques pour travailler avec les dictionnaires en Python 139
Utilisez des clés immuables et descriptives : 139
Préférez get() pour accéder aux valeurs lorsque les clés peuvent être absentes : 139
Utilisez des méthodes de parcours adaptées au contexte : 139
Évitez d'imbriquer trop de niveaux de dictionnaires : 140
Utilisez des compréhensions de dictionnaire pour des transformations simples : 140
Videz un dictionnaire avec clear() plutôt que de recréer une nouvelle variable : 140
Évitez les modifications de dictionnaire pendant un parcours : . . . 140
Commentez les structures complexes : 141

Testez l'existence d'une clé avant de modifier ses valeurs : 141
En conclusion sur les dictionnaires 142
3.3.6 Les chaînes 142
3.3.7 Les tableaux 142
3.3.8 Maîtriser le slicing en Python 142
Syntaxe du slicing en Python 142
Comprendre les indices négatifs 143
Slicing sur les listes 144
Extraire une sous-liste : 144
Utiliser des indices négatifs : 144
Slicing avec un pas (step) : 144
Modifier une liste avec le slicing : 144
Cas particulier : inverser une liste - 145
Erreurs courantes à éviter 145
Slicing sur les chaînes de caractères 145
Extraire une sous-chaîne : 145
Utiliser des indices négatifs : 146
Slicing avec un pas (step) : 146
Inverser une chaîne de caractères : 146
Différence entre split() et slicing : 146
Cas d'usage pratique : récupérer un mot spécifique - 146
Erreurs courantes à éviter 147
Slicing sur les tuples 147
Extraire une sous-partie d'un tuple : 147
Utiliser des indices négatifs : 148
Slicing avec un pas (step) : 148
Différence avec les listes : immuabilité des tuples - 148
Utilisation du slicing sur les tuples en pratique 148
Slicing et copies en mémoire 149

Copie superficielle (shallow copy) - 149
Copie profonde (deep copy) : 149
Conclusion sur le slicing 149
3.4 Les fonctions 150
3.4.1 Qu'est-ce qu'une fonction ? 150
3.4.2 Donner des informations à vos fonctions 150
Fonction avec un paramètre 151
Paramètres avec valeurs par défaut 151
3.4.3 Récupérer un résultat de vos fonctions 151
3.4.4 Exemple pratique : créer une calculatrice simple 152
3.4.5 Concepts avancés (optionnels pour débuter) 152
Les fonctions imbriquées 152
Les fonctions courtes (lambda) - Version simplifiée 153
Les arguments variables (*args et **kwargs) 154
Les annotations de type 154
Les fonctions récursives 155
3.4.6 Bonnes pratiques pour écrire des fonctions en Python 155
Nommer les fonctions de manière claire et descriptive 155
Garder les fonctions courtes et spécifiques 155
Utiliser les docstrings pour documenter les fonctions 156
Bonnes pratiques pour les docstrings : 157
Éviter les effets de bord 157
Utiliser les annotations de type 157
3.4.7 Gérer les exceptions de manière appropriée 157
Éviter les arguments par défaut mutables 158
Respecter les conventions de style 158
3.4.8 Travaux pratiques sur les fonctions 159
Objectifs pédagogiques 159
3.5 Les modules et les packages 162

3.5.1 Qu'est-ce qu'un Module Python ? 162
3.5.2 Pourquoi Utiliser des Modules ? 162
3.5.3 Comment créer un Module Python 163
3.5.4 Importation et Utilisation d'un Module 163
3.5.5 Organisation des Modules avec les Packages 164
3.5.6 Gestion des Importations dans les Packages 165
3.5.7 Documentation des Modules 166
3.5.8 Création de Modules Installables 166
3.5.9 Publication des Packages sur PyPI 167
Prérequis 167
Structure du Package 167
Création de l'Archive 168
Publication avec Twine 169
3.5.10 Installation du Package 169
3.5.11 Mise à Jour de Votre Package 169
3.5.12 Bonnes Pratiques 169
Pour la Publication sur Pypi 170
3.5.13 En conclusion sur les modules et les packages 170
3.6 TP sur le chapitre 171
3.6.1 TP 1 : Système de gestion d'étudiants 171
3.6.2 TP 2 : Le nombre mystère 178
Enoncé 178
Proposition de correction 179
3.7 EXERCICE 180

4 Module 4 : Programmation Orientée Objet et les Décorateurs 182

4.1 Introduction à la POO 182
4.2 Les Concepts de Base de la POO 184
4.2.1 Les classes et les objets 184
4.2.2 Les attributs et les méthodes 184

4.2.3 L'instanciation d'un objet 184
4.2.4 L'encapsulation 184
4.2.5 Le polymorphisme 185
4.2.6 L'héritage 185
4.3 Les attributs de classe et d'instance 186
4.4 Créer sa Première Classe en Python 186
4.5 Les Méthodes Spéciales 188
La Méthode __str__ 189
La Méthode __eq__ 189
La Méthode __iter__ 189
La Méthode __add__ 190
La Méthode __repr__ 191
La Méthode __del__ 191
La Méthode __call__ 191
Autres méthodes spéciales utiles 192
4.6 Les décorateurs 192
4.6.1 Comment fonctionnent les fonctions en Python 192
4.7 Créer un décorateur simple 194
4.7.1 Structure de base d'un décorateur 194
4.7.2 Utilisation d'un décorateur 194
4.7.3 La syntaxe @ pour simplifier 195
4.7.4 Pourquoi utiliser des décorateurs ? 196
4.7.5 Empilement de décorateurs 196
Comment empiler des décorateurs 196
Ordre d'exécution 197
Étendre l'empilement aux méthodes de classes 198
4.7.6 Décorateurs dans des classes 199
4.7.7 Décorateurs pour les propriétés 200
4.7.8 Les décorateurs de Click 200

4.7.9 Conclusion sur les décorateurs 201

5 Module 5 : Gestion des exceptions et des fichiers **202**

5.1 Maîtriser la gestion des exceptions en Python 202

5.1.1 Pourquoi gérer les exceptions ? 202

5.1.2 Comprendre les types d'exceptions 203

5.1.3 Utiliser try et except 203

5.1.4 Capturer plusieurs exceptions 204

5.1.5 Le bloc finally 205

5.1.6 Lever des exceptions avec raise 205

5.1.7 Créer ses propres exceptions 205

5.1.8 Les assertions 206

5.1.9 Gestion des exceptions imbriquées 206

5.1.10 Exercice sur les exceptions 207

5.1.11 Bonnes pratiques en gestion d'exceptions 208

5.1.12 Conclusion sur les exceptions 209

5.2 Maîtriser la gestion des fichiers avec python 209

5.2.1 Pourquoi gérer les fichiers en Python 209

5.2.2 Ouvrir et fermer des fichiers 210

5.2.3 Lire des fichiers 211

5.2.4 Lire ligne par ligne 211

5.2.5 Lire plusieurs lignes avec *readlines()* 212

5.2.6 Itérer sur le fichier 213

5.2.7 Écrire dans des fichiers 213

5.2.8 Écrire plusieurs lignes 213

5.2.9 Ajouter du contenu à un fichier existant 214

5.2.10 Utiliser le bloc 'with' 214

5.2.11 Travailler avec différents formats de fichiers 214

Fichiers CSV 214

Fichiers JSON 215

5.2.12 Les exceptions liées aux fichiers 215
5.2.13 Navigation dans le système de fichiers avec Python 216
Changer de répertoire 216
Obtenir le répertoire courant 216
Lister les fichiers et répertoires 216
Créer et supprimer des répertoires 216
Gérer les chemins de fichiers 216
5.2.14 Manipulation avancée des fichiers 217
Copier et déplacer des fichiers 217
Suppression récursive de répertoires 217
Modification des permissions de fichiers 217
Lecture et écriture des métadonnées 218
Gestion des fichiers temporaires 218
5.2.15 Exemples pratiques et cas d'utilisation 218
Exemple 1 : Traitement de fichiers CSV 218
Exemple 2 : Fusion de plusieurs fichiers texte 218
Exemple 3 : Nettoyage de répertoires 219
5.2.16 Exercice sur les fichiers 219
5.2.17 Bonnes pratiques pour la gestion des fichiers 223
5.2.18 Conclusion sur les fichiers 224

6 Module 6 : Programmation réseau 225

7 Module 7 : Python et scripts système (Bash, PowerShell) 226

8 Module 8 : Interfaces graphiques 227

9 Module 9 : Développement web avec Python 228

9.1 Présentation de Django 228
9.1.1 Qu'est ce que Django ? 228
9.1.2 Pourquoi travailler avec le framework Django ? 229

9.1.3 IA et Django 230
9.1.4 API REST et Django 230
9.1.5 La structure Django 230
9.1.6 Comment les requêtes sont traitées dans Django 231
9.1.7 Une documentation officielle très pédagogique 231
9.1.8 Qui utilise Django ? 232
9.2 Installation du framework Django 233
9.2.1 Conditions préalables 233
9.2.2 Installer Python 233
9.2.3 Mise en place d'un environnement virtuel 234
9.2.4 Installer Django 235
9.3 Création d'un nouveau projet Django 235
9.3.1 Comprendre la structure du projet Django 236
9.4 Comprendre les modèles et les bases de données de Django 239
9.4.1 Création et migration de modèles 239
Création d'un modèle 239
Ajouter le modèle à la base de données 240
Créer une migration : 240
Appliquer la migration : 240
9.4.2 Mise en place de la base de données 240
Configuration de la base de données SQLite par défaut 240
9.4.3 Intégration des bases de données 241
SQLite 241
Installation : 242
Configuration : 242
PostgreSQL 242
Installation : 242
Configuration : 242
Différences : 243

MySQL / MariaDB 243
Installation : 243
Configuration : 243
Différences : 243
Oracle / Microsoft SQL Server 243
Installation : 243
Configuration : 244
Différences : 244
Alors, lequel choisir ? 244
Configuration de PostgreSQL avec Django 244
9.5 Construire des vues et des templates 246
9.5.1 Comprendre les vues 246
Vues basées sur les fonctions (FBV) 246
Vues par classe (CBV) 247
Quand utiliser les FBV ou les CBV ? 247
9.5.2 Travailler avec des templates 248
Création d'un template 248
Rendu d'un template dans une vue 249
Balises et filtres de templates 249
Extension des templates 250
Étendre le template de base dans un template enfant (homepage.html) . . 250
9.6 Travailler avec des modèles d'URL 251
9.6.1 Configuration de base de l'URL 251
URLs au niveau du projet (myproject → urls.py) 252
URLs au niveau de l'application (myproject → myapp → urls.py) 252
Routage d'URL avec des vues basées sur des fonctions (FBV) 252
Routage d'URL avec des vues basées sur des classes (CBV) 253
Paramètres d'URL et routage dynamique 253
Modèles d'URL nommés 254

9.7 Gestion des fichiers statiques dans Django 254
9.7.1 Mise en place de fichiers statiques dans Django 254
9.7.2 Utilisation de fichiers statiques dans les templates 255
9.7.3 Collecte des fichiers statiques pour le déploiement 256
9.7.4 Servir des fichiers statiques en production 256
9.8 Mise en œuvre de l'authentification et de l'autorisation des utilisateurs 257
9.8.1 Authentification de l'utilisateur 257
Configuration de la connexion et de la déconnexion des utilisateurs 257
Création d'une vue d'enregistrement des utilisateurs 258
9.8.2 Autorisation de l'utilisateur 259
Restreindre l'accès à l'aide d'une connexion obligatoire 259
Gestion des autorisations des utilisateurs 260
Restreindre l'accès en fonction des rôles des utilisateurs 260
9.9 Gestion des formulaires dans Django 261
9.9.1 Création de formulaires à l'aide de la classe Form de Django 261
9.9.2 Validation des formulaires et traitement des erreurs 263
Validation intégrée 263
Validation personnalisée 263
9.9.3 Création de formulaires à partir de modèles à l'aide de ModelForm 264
Création d'un ModelForm 265
Utilisation du ModelForm dans une vue 265
9.10 Déployer une application Django 265
9.10.1 Meilleures pratiques pour le déploiement d'un projet Django 266
9.10.2 Déployer Django sur un VPS 266
Création d'un fichier requirements.txt en Python 266
Téléchargement du projet vers un dépôt distant (GitHub) 267
Préparez votre VPS et récupérez le code du projet 268
9.11 Conclusion 271
9.12 TP : Développer une application de collecte de données avec Django 271

9.13 Prérequis 274
9.14 Etape 1 : Création du projet et de la base de données 274
9.15 Etape 2 : Configurations du projet 276
9.16 Etape 3 : Développement des modules du projet 281
9.16.1 Module de gestion des compte utilisateurs (accounts) 281
Mise en place du modèle des users 281

10 Module 10 : Python pour la Data Science **285**

10.1 Calcul scientifique avec NumPy 286
10.1.1 Importation de NumPy 286
10.1.2 Tableaux NumPy 287
10.1.3 Pourquoi utiliser un tableau NumPy ? 287
10.1.4 Création de tableaux NumPy à partir d'objets 287
À partir d'une liste Python 287
Méthodes intégrées pour créer des tableaux 288
arange : 288
zeros et ones : 288
linspace : 288
eye : 289
Random 289
rand : 289
randn : 289
randint : 289
seed : 289
10.1.5 Attributs et méthodes Array 290
Reshape 290
max, min, argmax, argmin 290
Shape 291
dtype 291
10.2 Indexation et Selection NumPy 291

10.2.1 Crochets d'indexation et de sélection 292
10.2.2 Broadcasting 292
10.2.3 293
10.2.4 Plus d'aide pour l'indexation 294
10.2.5 Sélection conditionnelle 294
10.3 Opérations NumPy 295
10.3.1 Arithmétique 295
10.3.2 Fonctions universelles de tableaux¶ 296
10.4 Statistiques sommaires sur les tableaux 296
D'autres statistiques récapitulatives incluent : 296
10.5 Logique d'axe 297
10.6 Analyse de données avec Numpy 297
10.6.1 Effectuer des comparaisons 298
Mission 298
Sélection des éléments 298
Sélection avec les conditions multiples 299
Modification de certaines valeur dans un tableau numpy 299
10.7 TP sur Numpy 302
10.8 Analyse des données avec Pandas 304

11 Module 11 : Python pour la cybersécurité **306**

Avant-Propos

Objectifs généraux du cours

- Comprendre la syntaxe, les structures et les paradigmes du langage Python.
- Être capable de développer des applications complètes : console, web, réseau, graphiques, sécurité et data science.
- Savoir manipuler les bibliothèques et outils les plus utilisés dans l'écosystème Python.

Outils de travail

- Support de cours fourni par l'enseignant
- Python 3.9+
- PyCharm
- Anaconda
- Spyder
- SGBD PostgreSQL
- Navigateur web

Système d'évaluation

- Examen théorique : 30% (QCM sur les concepts fondamentaux)
- Projets pratiques : 40%

  — La calculatrice
  — Nombre d'occurrence d'une lettre dans une phrase
  — Inverser les lettres d'un mot
  — Le nombre magique
  — Créer un générateur de mot de passe
  — Vérifier un mot de passe
  — Compter le nombre de phrases dans un texte
  — Vérifier si une phrase est un palindrome
  — Développement d'un système de chat interposé
  — Chercher un mot dans un fichier
  — Convertisseur de températures
  — Projet "Backdor : prendre le contrôle d'une machine à distance"
  — Projet "Suivi des dépenses" (Application GUI)
  — Récupérateur d'article sur Wikipedia (Web Scraping)
  — Développer une mini système de blog (Application web)

— Projet "Analyse des ventes" (Data Analysis)

- Examen théorique : 30%
- Projet final : 10%

RESSOURCES COMPLÉMENTAIRES

- Documentation officielle Python
- Stack Overflow Python section
- Real Python tutorials
- Python Weekly newsletter
- Mon coach de Data Science
- Django Documentation
- Communauté discord de python
- GitHub Python repositories
- AFPy - Association Francophone Python

CONSEILS PÉDAGOGIQUES AUX ÉTUDIANTS

- Codez quotidiennement même 30 minutes
- Lisez du code d'autres développeurs
- Soyez actifs dans les communautés/fora Python
- Ne craignez pas les erreurs - elles font partie de l'apprentissage

Chapitre 1

Module 1 : Mise en place de l'environnement de travail

Objectifs :

— Découvrir Python, son histoire, ses domaines d'application.
— Installer et configurer l'environnement (Python, pip, venv, IDE).

Contenu :

- Installation de Python sous Windows et sous Linux.
- Installation des EDI (PyCharm, Anaconda, Sypder, etc.).
- Installation de Git pour le versionning de nos projets.

1.1 Découvrir Python

Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et permet une approche simple mais efficace de la programmation orientée objet. Parce que sa syntaxe est élégante, que son typage est dynamique et qu'il est interprété, Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur la plupart des plateformes.

Python est un langage dit de haut-niveau, c'est-à-dire qu'il s'éloigne du langage machine en ajoutant des concepts et des outils le rendant plus facile à lire et à écrire, plus proche du langage humain (de l'anglais en l'occurrence).

Python est aussi un langage portable : en dehors de certains cas exceptionnels, un programme Python peut être exécuté de la même manière sur un ordinateur Windows, Mac OS ou GNU/Linux, ainsi que sur des OS mobiles comme Android ou iOS.

1.1.1 Histoire de python

Place maintenant à un petit historique du langage Python, pour mieux le situer et le comprendre.

Le langage de programmation Python a été créé en 1989 par Guido van Rossum, développeur dans un centre de recherche néerlandais (CWI - Centrum voor Wiskunde en Informatica) qui commença à développer le projet à la fin des années 1980. Il s'inspira principalement du langage de programmation ABC qui était développé au sein de ce centre mais aussi du langage C et de Modula-3 . Le nom ***Python*** vient d'un hommage à la série télévisée *Monty Python's Flying Circus* dont Guido van Rossum est fan. La première version (*0.9.0*) publique de ce langage a été publiée en 1991.

La version 1.0 de Python arriva en 1994 et ressemblait déjà beaucoup au Python d'aujourd'hui que je vais vous enseigner dans ce cours.

Suivront les versions majeures 2.0 en 2000 et 3.0 en 2008, avec bien sûr de nombreuses versions mineures intermédiaires. Elles ajouteront à Python de nouvelles influences, telles que les langages de programmation Haskell ou SETL .

La version 3 fit grand bruit puisqu'elle choisit de casser la rétrocompatibilité pour corriger des erreurs de conception des versions précédentes, il fallut une dizaine d'années pour achever la transition de Python 2 vers Python 3.

En 2001 est créée la Python Software Foundation (ou PSF) dans le but de promouvoir et de protéger le langage, supervisant son développement et récoltant les fonds pour son financement.

La fondation œuvre aussi pour organiser des conférences tout autour du monde à propos de Python, les PyCon. On trouve par exemple la PyConFr organisée chaque année par l'AFPy, relai francophone de la PSF.

À la tête du développement du projet depuis sa création en tant que BDFL (ou « Dictateur bienveillant à vie »), Guido van Rossum s'en retire en 2018 pour des « vacances permanentes », laissant place à un comité de direction. Il reste depuis très impliqué dans les évolutions du langage.

Python suit depuis 2019 un cycle annuel de sortie, chaque nouvelle version paraissant au mois d'octobre. Le calendrier de support des versions de Python est disponible sur la page de téléchargements du site officiel.

La version actuelle stable de Python est Python 3.14.0, sortie le 7 octobre 2025. Cette version est la dernière de la branche de développement de Python. Il est recommandé d'utiliser les versions de Python 3 pour tous les nouveaux projets, car la version 2 n'est plus maintenue (depuis le 1er janvier 2020) et n'est pas recommandée pour un usage général. Cependant, il est à noter qu'il se peut que vous rencontriez sur le net des exemples de code utilisant Python 2 et qui ne seraient alors pas compatibles avec votre version.

Dans le cadre de notre cours ce sera la version 3.13.3 (publiée le 8 avril 2025) qui sera utilisée.

Mais au fait, que peut-on faire avec python ?

1.1.2 Les différents domaines d'application de python

La première question que vous pourriez vous poser à propos de Python serait : « Qu'est-ce qu'il est possible de faire avec ? ». À laquelle je pourrais répondre « à peu près tout » mais ce ne serait pas très précis.

Python a une grande variété d'applications, notamment dans le développement web et mobile, la science des données, l'intelligence artificielle, l'automatisation et le développement de logiciels. Il

est aussi utilisé pour la réalisation d'interfaces graphiques, la cartographie, la finance, la modélisation scientifique et le développement de jeux vidéo. Sa polyvalence provient de la richesse de ses bibliothèques et de sa capacité à s'adapter à de nombreux domaines.

Voyons donc où exactement nous pouvons utiliser Python dans le monde réel.

FIGURE 1.1 – Domaines d'application de Python

Python pour l'analyse de données

Les données sont devenues un atout précieux dans tous les secteurs d'activité modernes, et la plupart des entreprises sont intéressées par la collecte, la manipulation et l'analyse de données pertinentes afin d'en extraire des informations utiles pour l'entreprise. Et c'est là que Python dépasse toute concurrence.

Python est particulièrement précieux car, outre sa bibliothèque standard complète, il fournit une collection impressionnante de modules supplémentaires conçus spécifiquement à des fins analytiques.

Les bibliothèques Python les plus connues pour effectuer des analyses de données sont pandas et NumPy. Ces outils vous permettent de faire presque tout avec vos données, comme les nettoyer et les manipuler, explorer les statistiques ou visualiser les tendances cachées dans vos données.

Outre ces deux bibliothèques, vous pouvez en utiliser beaucoup d'autres pour diverses tâches liées aux données, telles que la visualisation de données, l'exploration du web ou les tests d'hypothèse.

Python pour la visualisation de données -

La visualisation des données est une partie autonome de l'analyse des données qui nous aide à représenter les informations, qu'elles soient brutes ou nettoyées et transformées, sous une forme plus convaincante et plus perspicace.

C'est là que Python entre à nouveau en jeu, offrant un large spectre d'outils pour visualiser les données. Les plus populaires d'entre eux sont matplotlib et, sur sa base, seaborn. Grâce à eux, nous pouvons créer littéralement toutes sortes de visualisations, des plus simples (diagrammes linéaires, diagrammes à barres, histogrammes) aux plus avancées (diagrammes animés, lignes de courant, cartes en grappes, diagrammes de joie, etc.)

Si vous souhaitez en savoir plus sur l'utilisation de ces deux bibliothèques Python, vous pouvez suivre nos cours Introduction à la visualisation de données avec Matplotlib et Introduction à la visualisation de données avec Python.

Il existe également une pléthore d'autres bibliothèques graphiques en Python : à la fois polyvalentes (comme Plotly, Bokeh ou Altair) et plus ciblées(missingno pour la visualisation des valeurs manquantes, Toyplot pour la création de tracés interactifs pour l'édition électronique, GeoPandas pour la construction de cartes, etc.).

Python pour l'apprentissage automatique -

L'apprentissage automatique est au cœur de la majorité des tâches liées à la science des données. Il s'agit d'un domaine de l'intelligence artificielle (IA) qui s'intéresse à l'utilisation d'algorithmes permettant aux machines d'apprendre des modèles et des tendances à partir de données historiques afin de faire des prédictions sur des données inconnues.

Grâce aux techniques de ML, nous pouvons créer des modèles capables de prédire avec précision le taux d'attrition de la clientèle d'une entreprise, d'estimer le risque qu'une personne soit atteinte d'une certaine maladie, d'identifier le positionnement optimal des véhicules de taxi, etc.

Avec Python, nous pouvons construire un modèle ML en utilisant seulement trois lignes de code (voir un exemple d'un tel modèle pour prédire les transactions bancaires frauduleuses). Même si derrière ces quelques lignes de code se cachent des processus et des calculs complexes, les bibliothèques ML de Python font le plus gros du travail sous le capot, ce qui facilite considérablement la tâche de l'utilisateur. Les bibliothèques les plus courantes sont scikit-learn, Keras, TensorFlow et PyTorch.

Python pour le développement de logiciels -

Outre son application multiforme dans les domaines de la science des données, Python est utilisé à chaque étape du développement de logiciels, notamment pour le contrôle de la construction, la compilation continue automatisée, le prototypage, le suivi des bogues, les tests et la maintenance logicielle.

La flexibilité et, en même temps, la puissance de ce langage de programmation permettent de l'utiliser pour créer des applications multiprotocoles relativement simples ou très avancées, ou celles qui impliquent des calculs numériques complexes.

Avec Python, nous pouvons construire des programmes audio ou vidéo basés sur des techniques d'IA ou de ML, des API (interfaces de programmation d'applications), des GUI (interfaces graphiques), ou tout autre type de logiciel.

Python pour le développement Web -

Alors que pour construire le front-end d'un site web (la partie d'un site web accessible à ses utilisateurs), nous utiliserons principalement des langages tels que HTML, CSS et JavaScript, pour son back-end (la partie invisible d'un site web), nous opterons souvent pour Python.

À ces fins, Python est utilisé pour certains cadres communs (comme Django ou Flask) avec des modules intégrés spécialisés qui permettent de partager des données avec des serveurs, de traiter des informations, d'accéder à des bases de données, d'acheminer des URL, de gérer du contenu et de maintenir la sécurité des sites web.

Parmi les sites web et applications géants créés à l'aide de Python, il convient de citer Google, Facebook, Instagram, Youtube, Dropbox, Udemy, Spotify et Reddit.

Python pour l'automatisation des tâches/la création de scripts -

Python est un outil formidable pour écrire des programmes permettant d'automatiser différentes tâches répétitives. Ce processus est également appelé "script".

En particulier, vous pouvez utiliser des scripts pour travailler avec des fichiers et des dossiers. Par exemple, vous pouvez créer, renommer, convertir, diviser, fusionner ou supprimer des fichiers, vérifier s'ils contiennent des erreurs, des doublons ou certains types de texte, ou encore mettre à jour leur contenu.

Vous pouvez également utiliser l'automatisation de Python pour rechercher et télécharger des informations sur internet, remplir et soumettre des formulaires en ligne, et envoyer régulièrement des notifications ou des courriels.

Faire toutes ces choses manuellement, en plus d'être extrêmement ennuyeux, prendrait beaucoup de temps et serait source d'erreurs. L'utilisation de scripts Python pour effectuer de telles actions en quelques minutes seulement augmenterait considérablement votre efficacité et vous permettrait d'économiser un temps et des efforts précieux pour des tâches plus créatives.

Diverses utilisations de Python -

Outre toutes les implémentations évoquées ci-dessus, à quoi sert Python ? Vous trouverez ci-dessous quelques idées nouvelles :

- Suivi du marché boursier ;
- Récupération de données sur Internet (Web scraping) ;
- Création de robots ;
- Tenir à jour une liste de tâches ;
- ....

1.1.3 Quels sont les emplois qui nécessitent Python ?

De nombreuses professions ont directement besoin de Python pour leur travail quotidien. Vous trouverez ci-dessous un aperçu de ce que vous pouvez faire avec Python dans un cadre professionnel :

- Scientifique des données ;
- Analyste de données ;
- Ingénieur de données ;
- Ingénieur en apprentissage automatique ;
- Journaliste de données ;
- Architecte de données ;
- Développeur web complet ;
- Développeur web back-end ;
- Ingénieur DevOps ;
- Ingénieur logiciel.

Vous avez toutes les raisons pour apprendre à programmer avec ce langage aussi puissant que passionnant. Cela commence par la mise en place d'un environnement de travail adéquat.

1.2 Mise en place de l'environnement de travail

Dans cette section il sera question d'installer les outils qui nous permettrons de développer nos solutions à l'aide du langage de programmation python. Nous aurons besoin de :

- Python lui-même
- D'un environnement de développement (IDE) en occurrence PyCharm
- D'un environnement de développement pour la Datascience en occurrence Anaconda

1.2.1 Installation de python

Avant de pouvoir commencer à programmer avec Python, il nous faut faire comprendre ce langage à notre ordinateur. Et pour cela, nous avons besoin d'installer un logiciel qui lui servira d'interprète. Ce logiciel répond simplement au nom de Python. A moins d'une contrainte imposée par votre système d'exploitation ou votre gestionnaire de paquets, préférez la dernière version 3.x.y en date.

Installer Python sur votre ordinateur est simple et rapide.

Installation sous Microsoft Windows

Voici les étapes à suivre pour l'installer sous une machine dotée de Microsoft Windows :

1. Allez sur le site officiel : https://www.python.org/downloads/
2. Choisissez la version adaptée à votre système d'exploitation (Windows, macOS, Linux) :

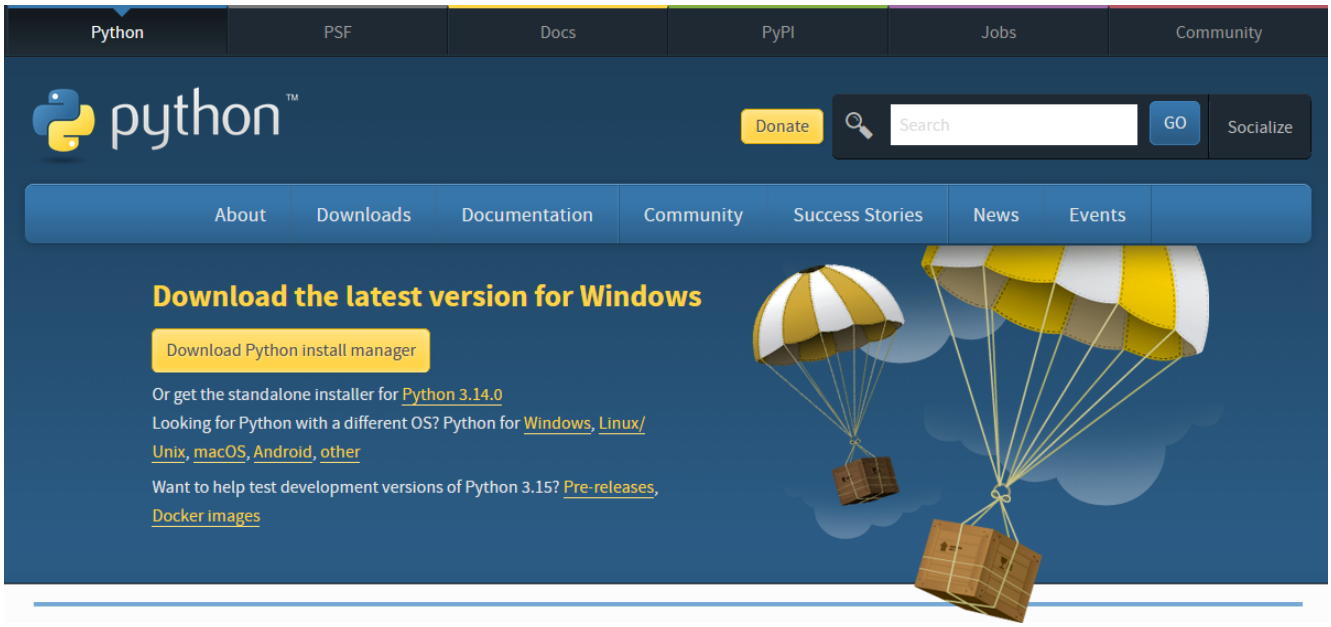


FIGURE 1.2 – Page de téléchargement de python

3. Téléchargez le fichier d'installation et exécutez-le :

FIGURE 1.3 – Installation de python

4. Sur Windows, cochez la case "Ajouter Python à PATH" pour faciliter l'utilisation en ligne de commande ; sur macOS et Linux, Python est généralement déjà dans le PATH.
5. Cliquez sur "Install Now" (Windows) ou suivez les instructions à l'écran (macOS, Linux) :

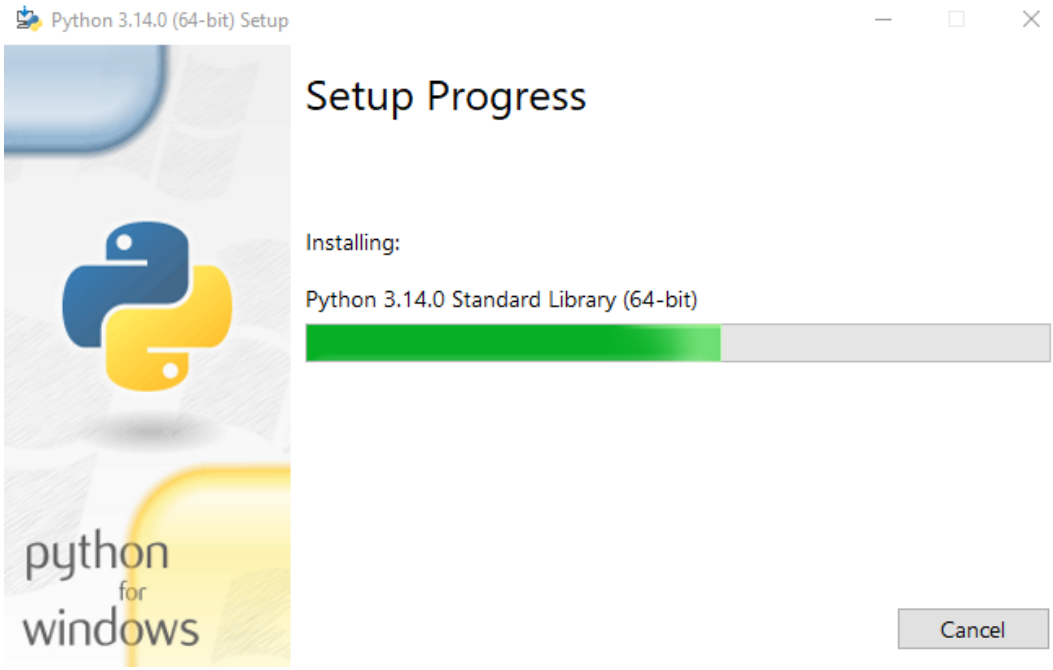


FIGURE 1.4 – Installation de python (Suite 1)

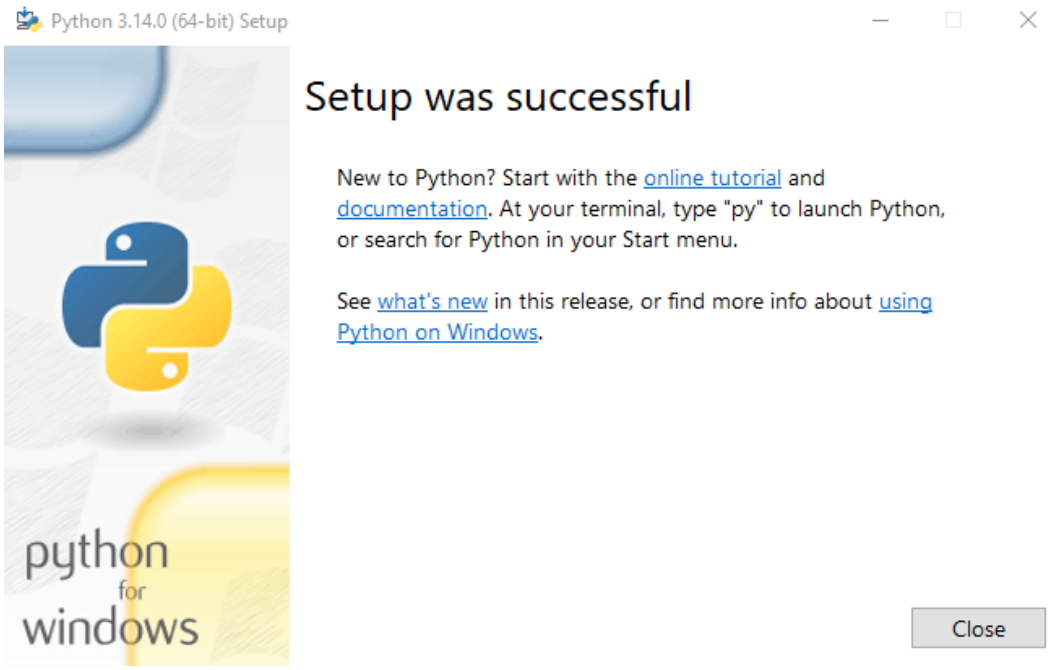
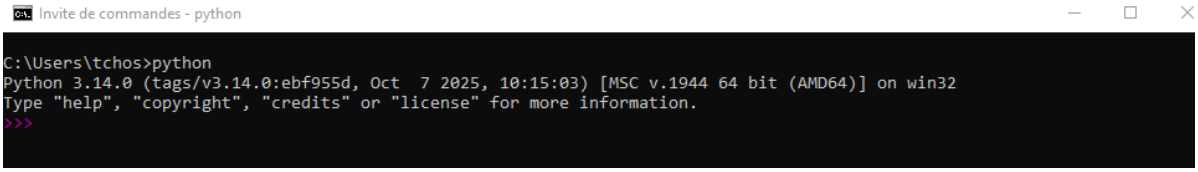


FIGURE 1.5 – Installation de python (Suite 2)

6. Après l'installation, testez Python en ouvrant un terminal (ou une invite de commandes pour Windows) et tapez python ou python3 pour lancer l'interpréteur Python interactif :



7. Essayez quelques exemples de code :

```
# Affiche "Hello, World!"
print("Hello, World!")

# Effectuez des opérations mathématiques
result = 2 + 7
print(result) # Affiche "9"

# Boucles et listes
filieres = ['ICT', 'INFO', 'MATHS']
for filiere in filieres:
    print(filiere)
```

Pour quitter l'interpréteur, tapez exit() ou appuyez sur Ctrl+Z (Windows) ou Ctrl+D (macOS, Linux).

Installation sous GNU/Linux

Si vous utilisez Linux, il est probable que Python soit déjà installé sur votre système, car il est nécessaire à certains outils d'administration. Mais il se peut qu'il ne soit pas installé dans la bonne

version.

Pour savoir s'il est déjà installé, vous pouvez entrer la commande *python -V* ou *python3 -V* dans un terminal et vous assurer de voir apparaître un message *Python 3.x.y*

L'installation de python achevée, passons à l'installation d'Anaconda.

1.2.2 Installation d'Anaconda

Anaconda est un gestionnaire de paquets, un gestionnaire d'environnement et une distribution Python qui contient une collection de nombreux paquets open source. Ceci est avantageux car lorsque vous travaillez sur un projet de science des données, vous trouverez que vous avez besoin de nombreux paquets différents (numpy, scikit-learn, scipy, pandas pour n'en citer que quelques-uns), avec lesquels une installation d'Anaconda est préinstallée.

Si vous avez besoin de paquets supplémentaires après avoir installé Anaconda, vous pouvez utiliser le gestionnaire de paquets d'Anaconda, conda, ou pip pour installer ces paquets. C'est très avantageux car vous n'avez pas à gérer vous-même les dépendances entre plusieurs paquets. En fait, l'installation d'Anaconda est également la méthode recommandée pour installer les blocs-notes Jupyter.

Voici comment télécharger et installer Anaconda :

Installation de Windows

Suivez minutueusement les étapes suivantes :

1. Téléchargez le programme d'installation à partir du site Web d'Anaconda, naviguez jusqu'à arriver à ce niveau pour télécharger la version compatible à votre système :

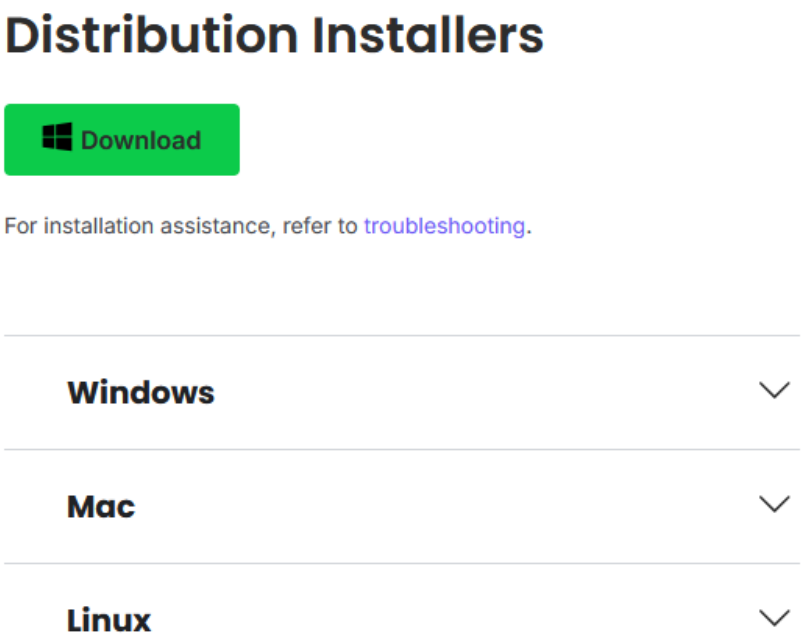


FIGURE 1.6 – Page de téléchargement d'Anaconda

2. Accédez à votre dossier Téléchargements et double-cliquez sur le programme d'installation pour lancer.

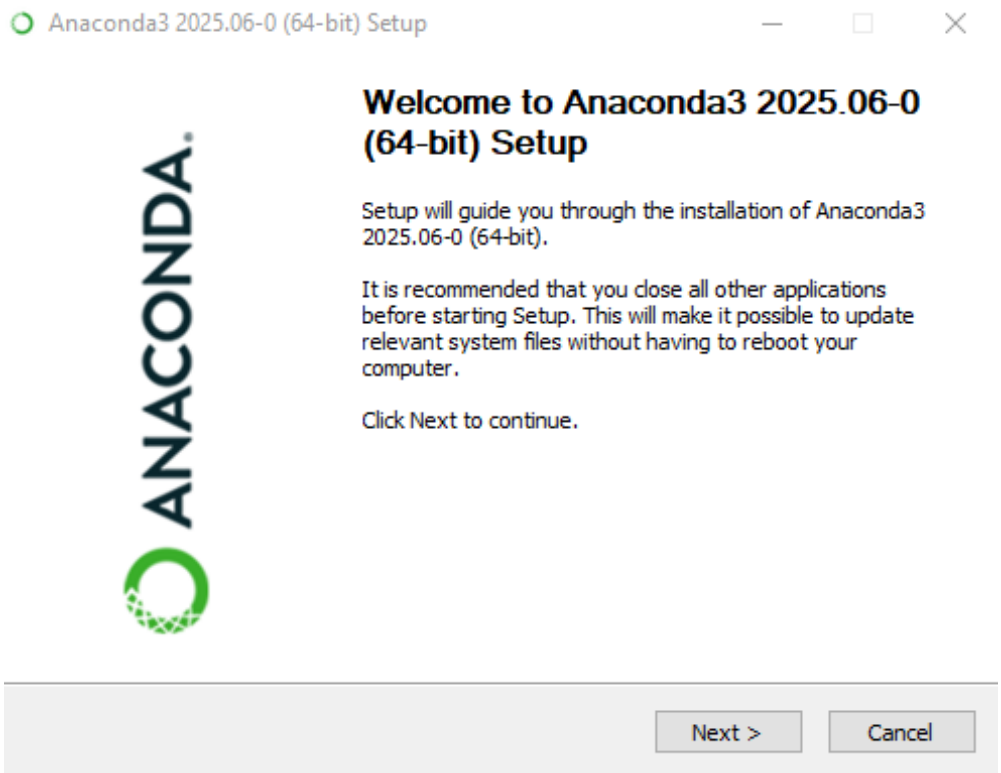


FIGURE 1.7 – Installation d'Anaconda

3. Cliquez sur Suivant, puis j'accepte d'accepter les Conditions d'utilisation (CGV) d'Anaconda :

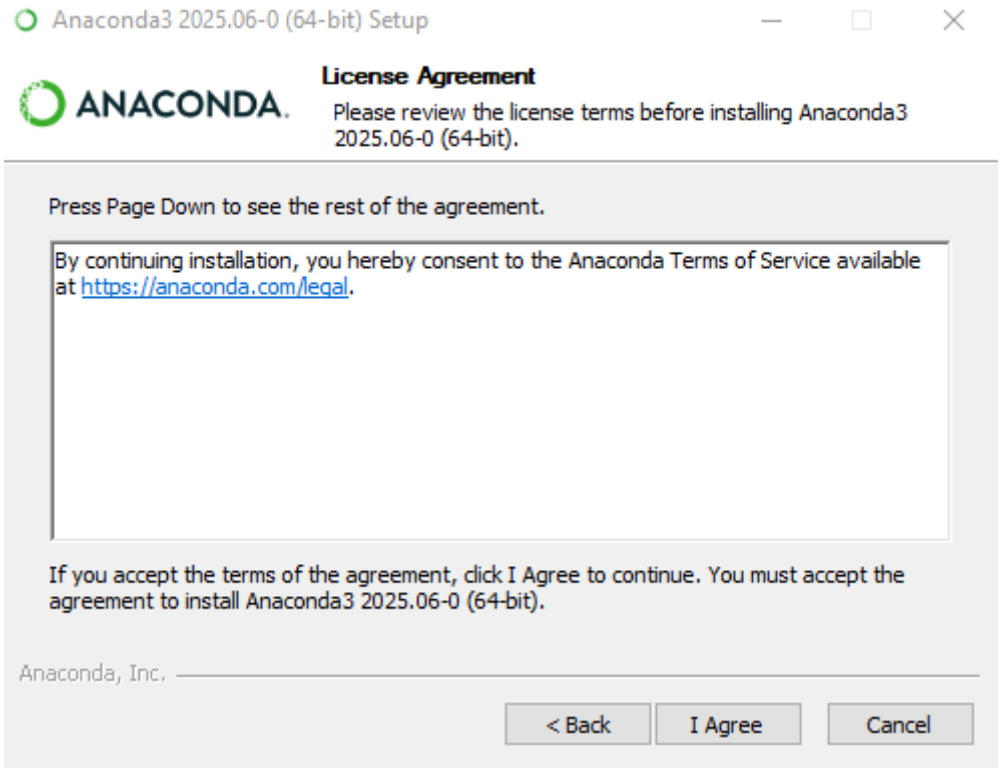


FIGURE 1.8 – Installation d'Anaconda (Suite 1)

4. Sélectionnez une option d'installation et cliquer sur Suivant (Next) :

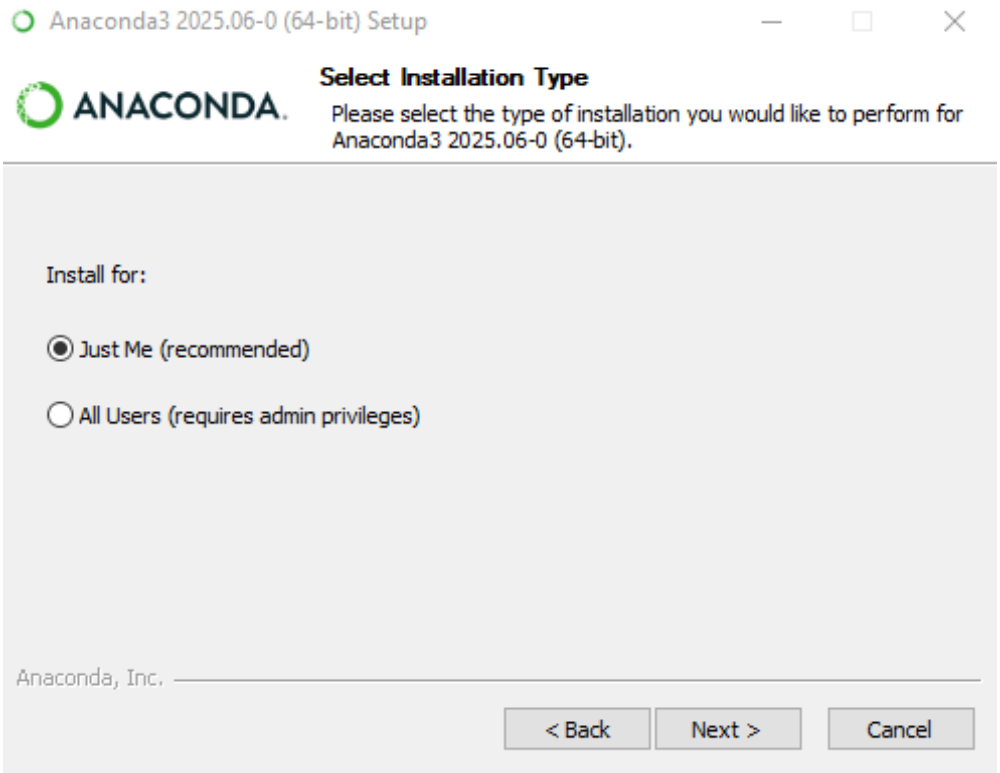


FIGURE 1.9 – Installation d'Anaconda (Suite 2)

5. Sélectionnez un dossier de destination pour installer Anaconda, puis cliquez sur Suivant (Next) :

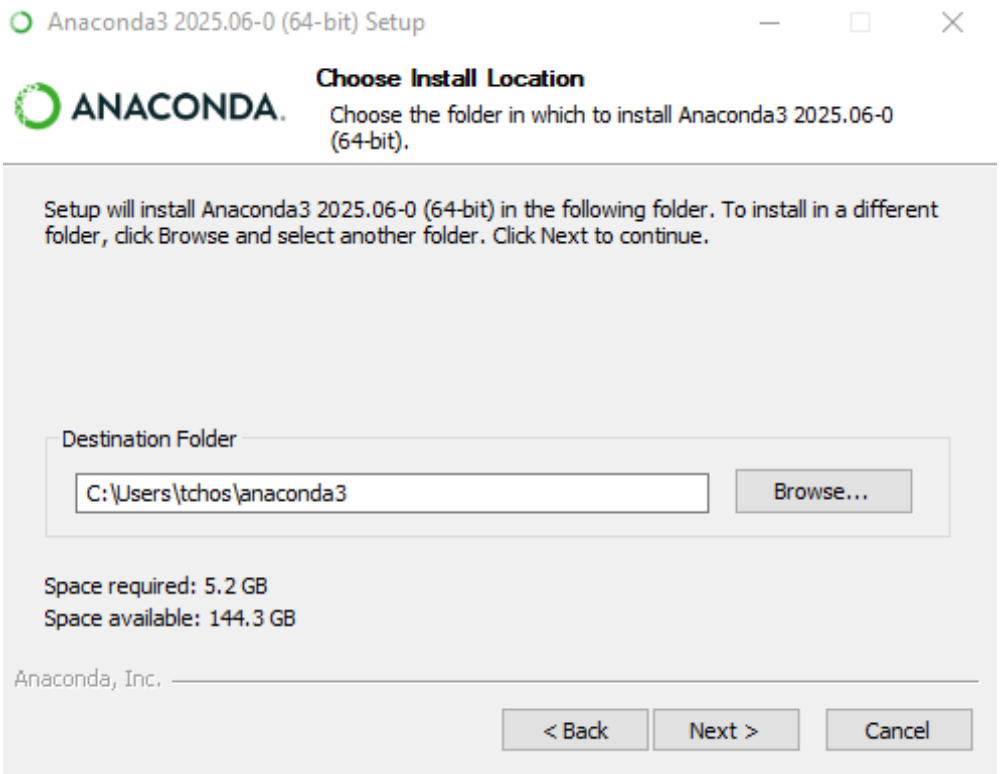


FIGURE 1.10 – Installation d'Anaconda (Suite 3)

6. Personnalisez vos options d'installation en cochant les mêmes options que sur la figure ci-dessous :

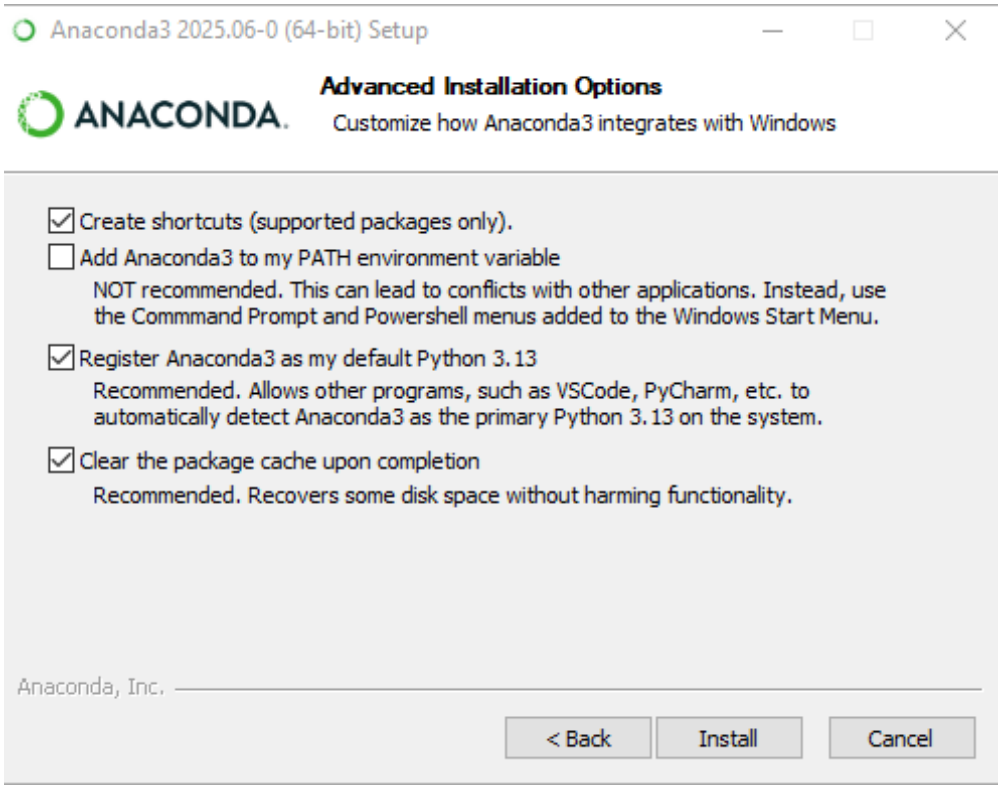


FIGURE 1.11 – Installation d'Anaconda (Suite 4)

7. Enregistrer Anaconda3 comme mon Python 3.12 par défaut - Sélectionné par défaut. Enregistre le paquet Python dans cette installation comme Python par défaut pour des programmes comme VSCode, PyCharm, et ainsi de suite. Cliquez ensuite sur Installer (Install).
8. Laissez l'installation se dérouler jusqu'à son terme :

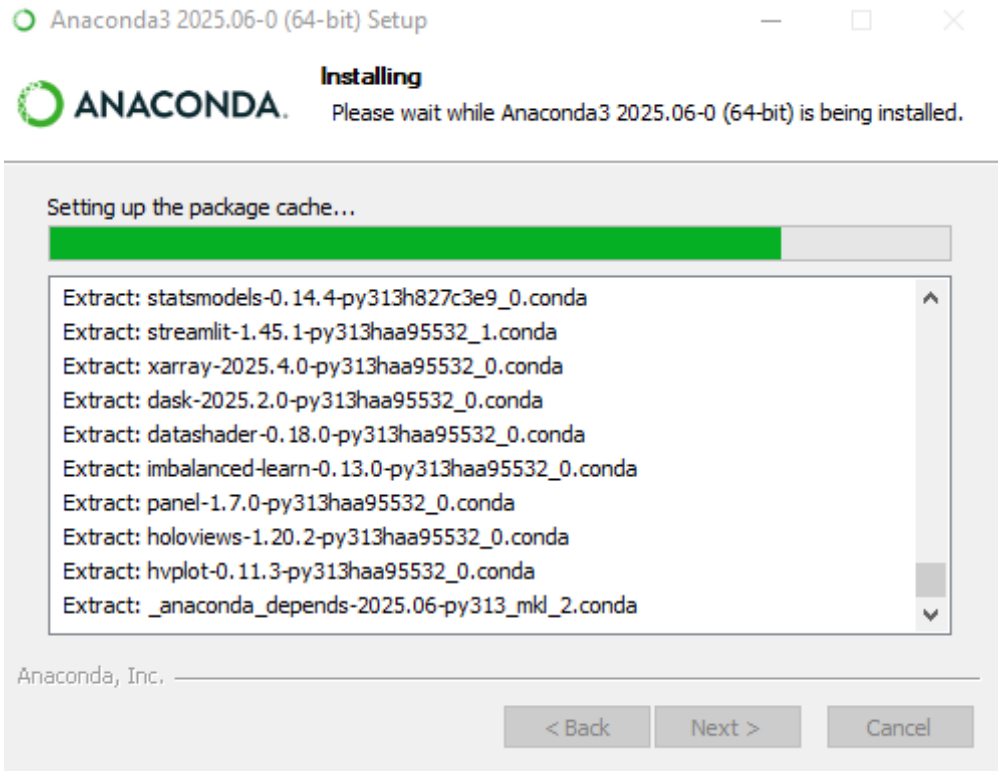


FIGURE 1.12 – Installation d'Anaconda (Suite 5)

9. Cliquez deux fois sur Suivant, puis cliquez sur Terminer pour fermer l'installateur :

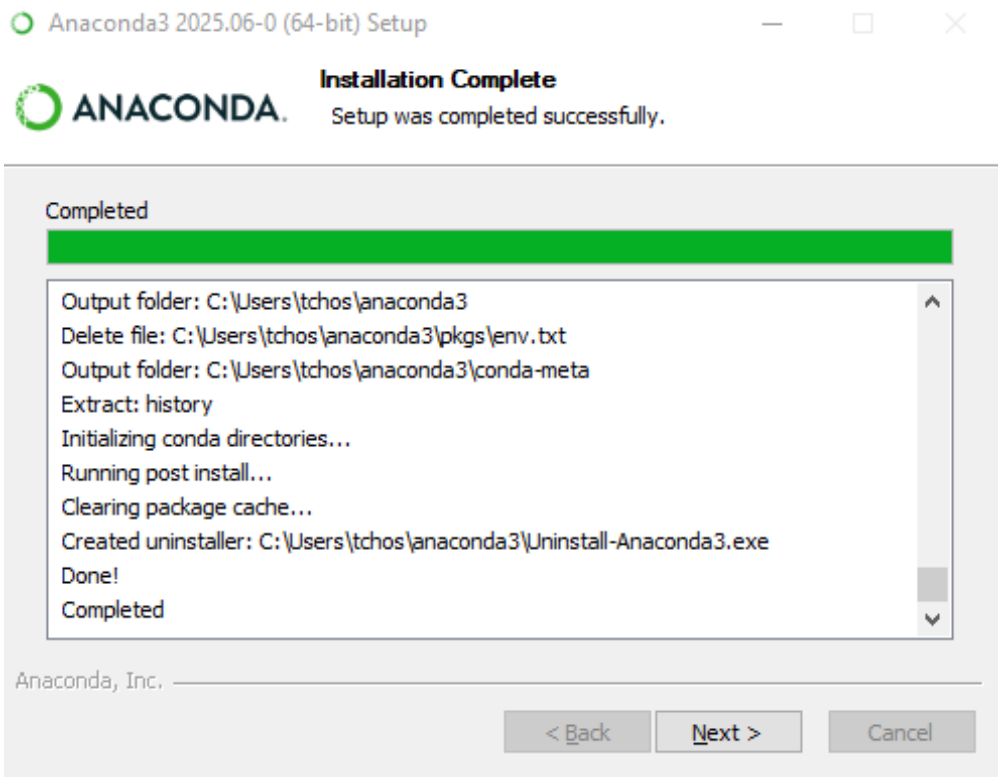


FIGURE 1.13 – Installation d'Anaconda (Suite 6)

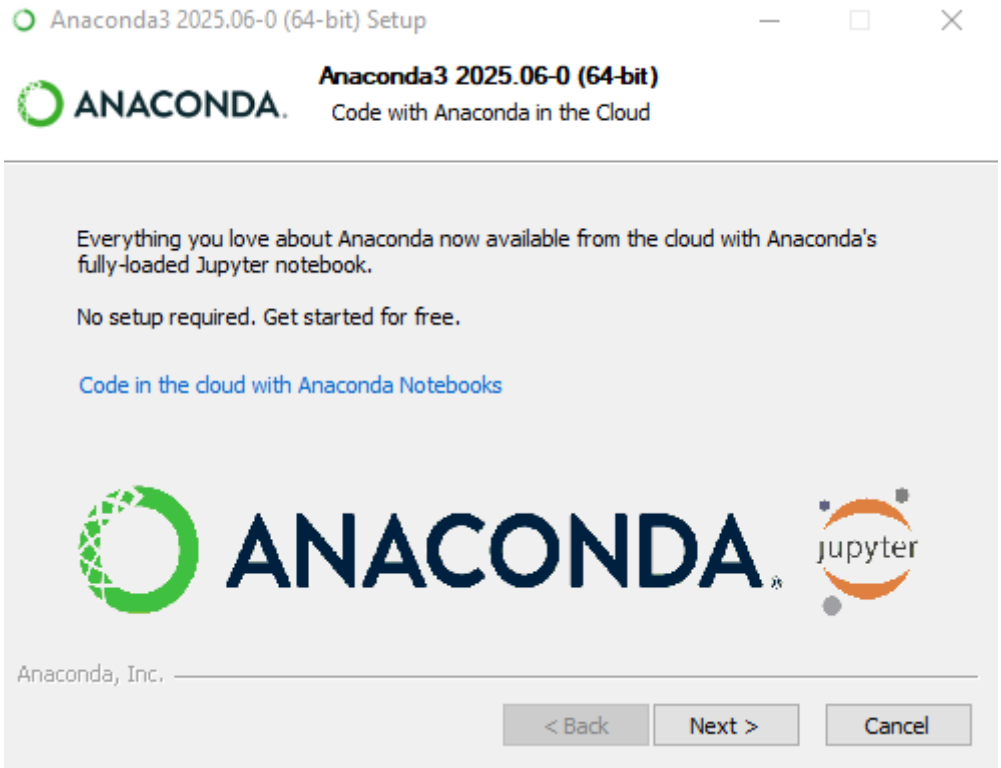


FIGURE 1.14 – Installation d'Anaconda (Suite 7)

10. Ca y est, Anaconda est désormais installé sur votre machine

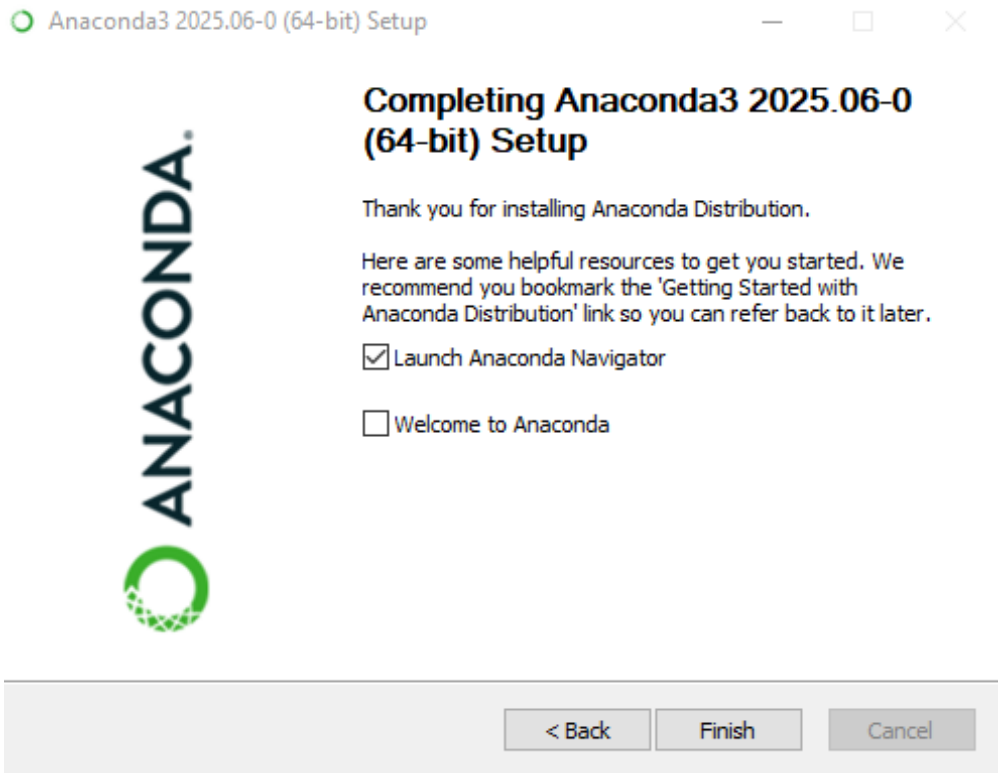


FIGURE 1.15 – Installation d'Anaconda (Suite et fin)

Installation d'Anaconda sous Linux

Suivez minutueusement les étapes suivantes :

1. Téléchargez le programme d'installation à partir du site Web d'Anaconda, naviguez jusqu'à arriver à ce niveau pour télécharger la version compatible à votre système :

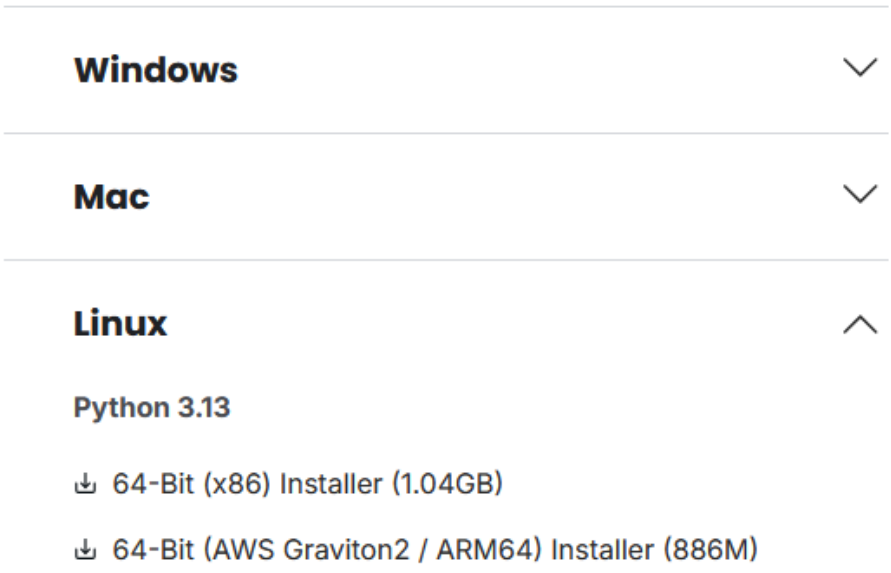


FIGURE 1.16 – Télécharger Anaconda pour Linux

2. Lancez l'installation du fichier télécharger via le terminal en vous rendans dans le dossier où se trouve le fichier source préalablement téléchargé et exécutez la commande ci-dessous :

```
parallels@ubuntu-linux-2404:~$ bash ~/Anaconda3-2024.10-1-Linux-aarch64.sh

Welcome to Anaconda3 2024.10-1

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

FIGURE 1.17 – Installation d'Anaconda sous Linux

3. Tapez la touche "Entrer" pour accepter l'installation puis sur la même touche "Entrer" pour accepter le dossier qu'il vous propose pour l'installation.
4. Laisser le processus se dérouler jusqu'à la fin comme dans le cas de l'installation sous Windows.

Il ne nous restera qu'à installer PyCharm.

1.2.3 Installation de PyCharm

PyCharm est un outil développé par la société JetBrains, spécialiste du développement d'IDEs pour différents langages de programmation.

Pour installer PyCharm Community sur Windows, téléchargez le programme d'installation sur le site de JetBrains, puis exécutez l'installateur et suivez les instructions, en sélectionnant les options souhaitées comme l'ajout du dossier bin au PATH, et en créant des raccourcis.

Instructions détaillées pour installer *PyCharm* sur votre système :

1. Trouvez le fichier d'installation téléchargé dans votre dossier de téléchargements et double-cliquez dessus.
2. Si une fenêtre d'avertissement apparaît, cliquez sur "Oui" pour lancer l'installation.
3. Dans l'assistant d'installation, cliquez sur "Next".
4. Choisissez un dossier de destination (le dossier par défaut est recommandé) puis cliquez sur "Next".
5. Dans la fenêtre des options d'installation :

   — Il est conseillé de cocher "Create Desktop Shortcut" pour créer un raccourci sur le bureau.
   — Il est également recommandé de cocher "Add 'bin' folder to the system PATH" pour faciliter l'utilisation de PyCharm en ligne de commande.

6. Cliquez sur "Next" puis "Install".
7. Une fois l'installation terminée, redémarrez votre ordinateur si vous y êtes invité pour finaliser le processus.
8. Lancez PyCharm depuis le raccourci sur le bureau.

Installer à l'aide de l'application Toolbox

L'application JetBrains Toolbox est l'outil recommandé pour installer les produits JetBrains.

Utilisez-le pour installer et gérer différents produits ou plusieurs versions d'un même produit, y compris les versions du programme d'accès anticipé (PAE), mettre à jour et faire reculer si nécessaire, et supprimer facilement tout outil.

L'application Toolbox maintient une liste de tous vos projets pour ouvrir rapidement n'importe quel projet dans la bonne IDE et la bonne version.

Pour installer l'application Toolbox, suivez pas à pas les étapes suivantes :

1. Téléchargez le .exe du programme d'installation à partir de la page Web de l'application Toolbox
2. Exécutez l'installateur et suivez les étapes de l'assistant.
3. Après avoir exécuté l'application Toolbox, cliquez sur l'icône Toolbox dans la zone de notification et sélectionnez le produit que vous souhaitez installer. Vous pouvez sélectionner et installer une version spécifique parmi les Versions disponibles.

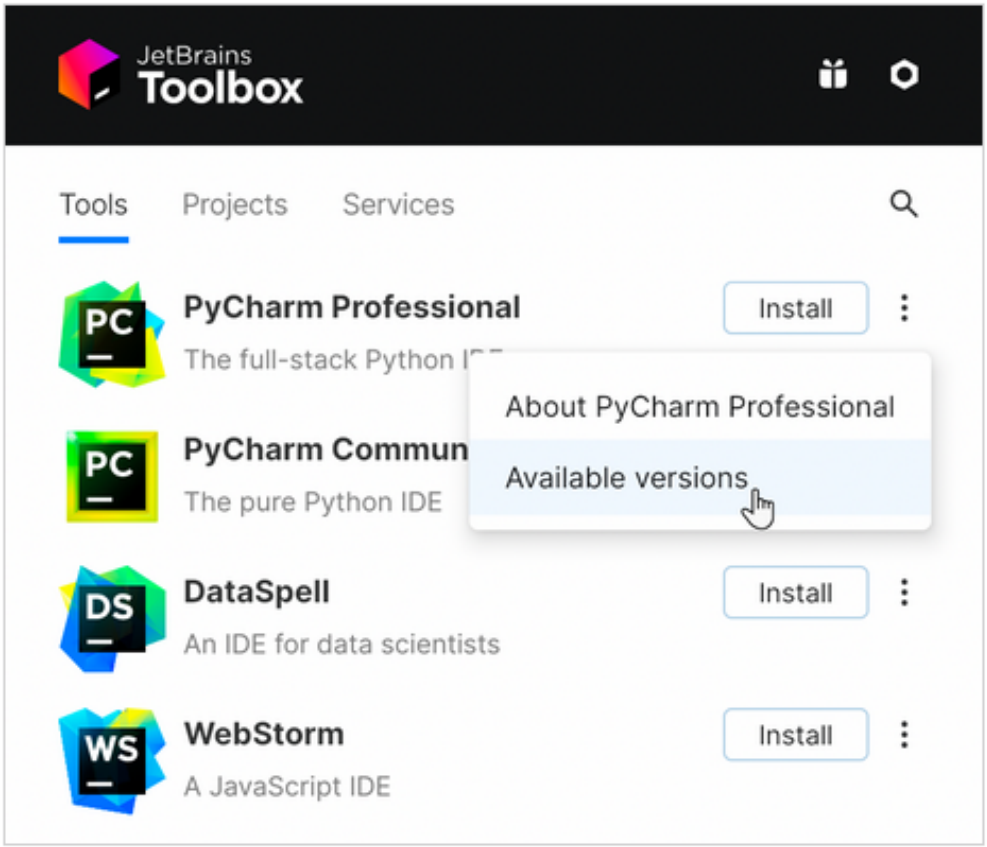


FIGURE 1.18 – Installation de PyCharm via ToolBox

Si vous avez installé *PyCharm* via l'application Toolbox, vous pouvez trouver le répertoire d'installation dans l'application : ouvrez les paramètres de l'instance IDE dans l'application Toolbox, développez Configuration et recherchez le champ Emplacement d'installation.

Remarque : L'installation d'un IDE via ToolBox nécessite une connexion à Internet.

1.3 Prise en main de nos environnements de travail

Dans le cadre de ce cours nous utiliserons ***Jupyter Notebook*** et ***PyCharm*** pour écrire nos programmes. Il est donc nécessaire que vous sachiez les utiliser.

1.3.1 Prise en main de Jupyter Notebook

Un notebook Jupyter est un document interactif qui combine code Python, texte explicatif et résultats dans une même interface web. Imaginez un cahier numérique où vous pouvez :

- Écrire et exécuter du code Python par petits blocs ;
- Documenter votre démarche avec du texte formaté ;
- Visualiser immédiatement les résultats et graphiques ;
- Partager facilement votre travail.

Cette approche révolutionne l'apprentissage et l'expérimentation en programmation, car vous avancez étape par étape, testez immédiatement vos idées, et gardez une trace de votre raisonnement.

Pourquoi débuter avec Jupyter ? Contrairement aux scripts Python classiques, les notebooks permettent d'expérimenter sans contrainte et de comprendre chaque étape de votre code.

Découvrons comment lancer Jupyter. Vous avez deux options principales.

Option 1 : Anaconda Navigator (recommandé débutants)

Anaconda Navigator est une interface graphique intuitive :

1. **Lancez** Anaconda Navigator depuis le menu Démarrer/Applications
2. **Cliquez** sur "Launch" sous ***Jupyter Notebook***
3. **Attendez** : Votre navigateur s'ouvre automatiquement
4. **Explorez** : Vous accédez à l'interface Jupyter

Option 2 : Terminal/Ligne de commande

Pour les plus aventureux, depuis le terminal :

```
# Créer un dossier pour vos notebooks (optionnel)
mkdir ~/Mes-Notebooks
cd ~/Mes-Notebooks

# Lancer Jupyter Notebook
jupyter notebook

# Résultat attendu :
# [I 10:30:15.123 NotebookApp] Serving notebooks from local directory
# [I 10:30:15.123 NotebookApp] Jupyter Notebook is running at:
# [I 10:30:15.123 NotebookApp] http://localhost:8888/?token=abc123...
```

FIGURE 1.19 – Démarrer Jupyter Notebook via l'invite de commandes

Jupyter Notebook s'ouvrira alors dans votre navigateur à l'adresse `http://localhost:8888`.

Astuce

À savoir : Vous pouvez aussi lancer JupyterLab, une interface plus puissante :

```
jupyter lab
```

Comprendre l'interface d'accueil

L'interface **Jupyter** ressemble à un explorateur de fichiers :

- Files : Vos dossiers et fichiers
- Running : Notebooks actuellement ouverts
- Clusters : Fonctionnalités avancées (ignorez pour l'instant)
- New : Bouton pour créer un nouveau notebook

Navigation : Cliquez sur les dossiers pour explorer, utilisez le bouton "New" pour créer.

Créer votre premier notebook

Étape 1 : Nouveau notebook

Dans l'interface Jupyter :

1. **Cliquez** sur le bouton "New" (en haut à droite)
2. **Sélectionnez "Python 3"** dans le menu déroulant
3. **Attendez** : Un nouvel onglet s'ouvre avec votre notebook vierge

Félicitations ! Vous venez de créer votre premier notebook.

Étape 2 : Nommer votre notebook Par défaut, votre notebook s'appelle "Untitled". Changeons cela :

1. **Cliquez** sur "Untitled" en haut de la page
2. **Tapez** : "Mon-Premier-Notebook"
3. **Validez** avec Entrée

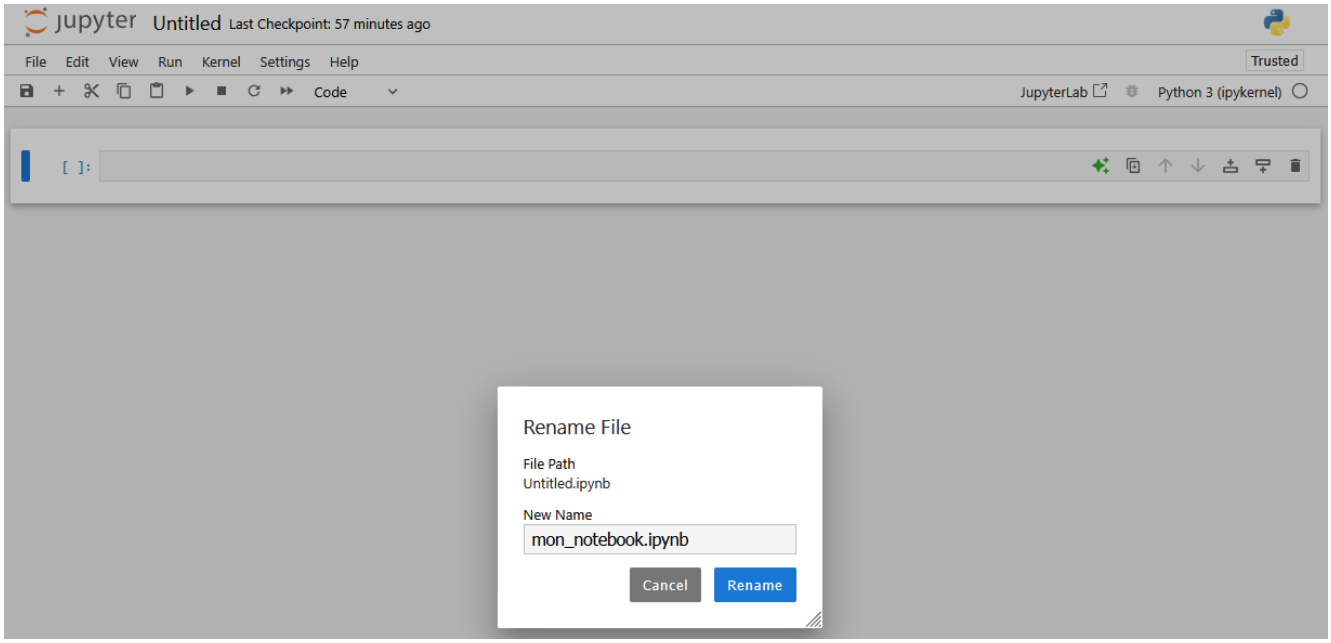
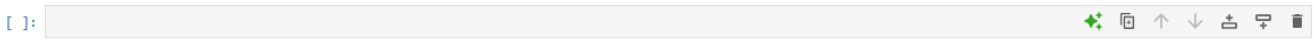


FIGURE 1.20 – Renommer un notebook

Sauvegarde automatique : Jupyter sauvegarde votre travail toutes les 2 minutes.

Étape 3 : Comprendre l'interface du notebook Votre **notebook** contient plusieurs éléments clés :

- **Cellule** : Zone rectangulaire où vous tapez du code ou du texte



- **Bouton Run** : Execute la cellule active

- **Mode édition** : Curseur visible, vous pouvez taper
- **Mode commande** : Cellule sélectionnée (bordure bleue), raccourcis actifs

Concept fondamental : Un notebook = séquence de cellules exécutables.

Vos premières cellules de code

Passons à la pratique ! Nous allons créer plusieurs cellules de code pour découvrir les bases.

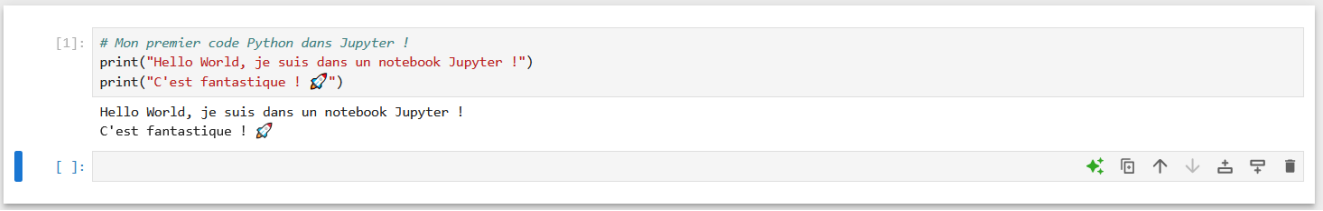
Cellule 1 : Hello World en Python Dans la première cellule (déjà présente), tapez :

```
# Mon premier code Python dans Jupyter !
print("Hello World, je suis dans un notebook Jupyter !")
print("C'est fantastique !!!")
```

Exécution :

— **Cliquez** sur le bouton **"Run"**
— Ou utilisez le raccourci **Shift + Entrée**

Magie ! Le code s'exécute et le résultat s'affiche immédiatement en dessous.



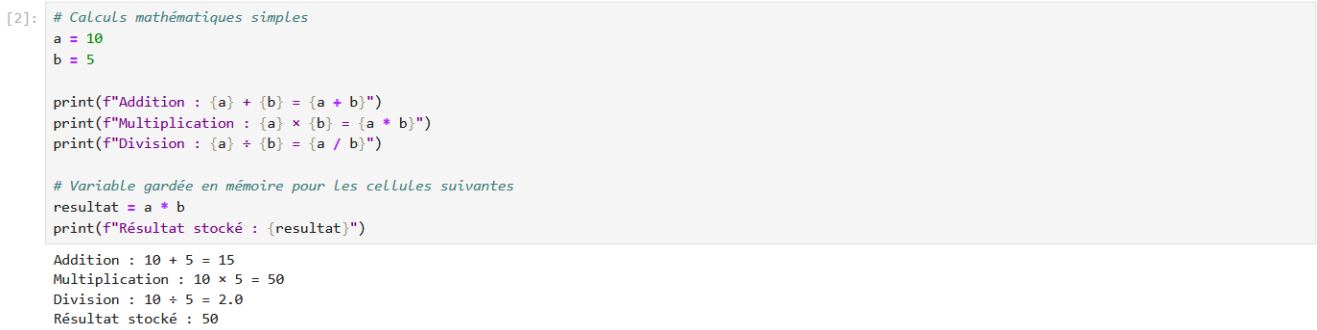
Cellule 2 : Calculs mathématiques Jupyter excelle pour les calculs interactifs. Créez une nouvelle cellule :

```
# Calculs mathématiques simples
a = 10
b = 5

print(f"Addition : {a} + {b} = {a + b}")
print(f"Multiplication : {a} × {b} = {a * b}")
print(f"Division : {a} ÷ {b} = {a / b}")

# Variable gardée en mémoire pour les cellules suivantes
resultat = a * b
print(f"Résultat stocké : {resultat}")
```

Résultat attendu :



Point important : Les variables restent en mémoire entre les cellules !

Cellule 3 : Utiliser la variable précédente Saisissez le code ci-après dans une nouvelle cellule :

```
# La variable 'resultat' de la cellule précédente est toujours accessible
print(f"Je récupère le résultat précédent : {resultat}")

# Je peux l'utiliser pour de nouveaux calculs
nouveau_calcul = resultat * 2
print(f"Nouveau calcul : {resultat} × 2 = {nouveau_calcul}")
```

Résultat attendu :

```
[3]: # La variable 'resultat' de la cellule précédente est toujours accessible
     print(f"Je récupère le résultat précédent : {resultat}")

     # Je peux l'utiliser pour de nouveaux calculs
     nouveau_calcul = resultat * 2
     print(f"Nouveau calcul : {resultat} × 2 = {nouveau_calcul}")

     Je récupère le résultat précédent : 50
     Nouveau calcul : 50 × 2 = 100
```

Ajouter du texte avec Markdown

Les notebooks ne contiennent pas que du code. Ajoutons des explications avec des cellules **Markdown**.

Créer une cellule Markdown

1. **Nouvelle cellule :** Cliquez sur "+" dans la barre d'outils
2. **Changer le type :** Menu déroulant "Code" → "Markdown"
3. **Taper du texte** formaté.

Voici un exemple de code que vous pouvez saisir dans une cellule "Markdown" :

Code Python :

```
# Mon Analyse de Données

Ceci est une **cellule Markdown** qui me permet d'expliquer mon travail.

## Objectifs
- Apprendre les bases de Jupyter
- Créer des graphiques simples
- Documenter mes découvertes

> **Note :** Le Markdown permet de structurer et embellir mes explications.

### Prochaines étapes
1. Analyser plus de données
2. Créer des graphiques avancés
3. Partager mon notebook
```

Exécution : Shift + Entrée transforme le texte brut en texte formaté élégant.

Mon Analyse de Données

Ceci est une **cellule Markdown** qui me permet d'expliquer mon travail.

Objectifs

- Apprendre les bases de Jupyter
- Créer des graphiques simples
- Documenter mes découvertes

> **Note importante :** Le Markdown permet de structurer et embellir mes explications.

Prochaines étapes

1. Analyser plus de données
2. Créer des graphiques avancés
3. Partager mon notebook

FIGURE 1.21 – Rendu du code "Markdown" après exécution

Vous ne connaissez pas le Markdown ? Pas de problème, vous pouvez l'apprendre ici.

L'autre environnement de développement avec lequel nous travaillerons et que nous avons installé à la section précédente, est **PyCharm**.

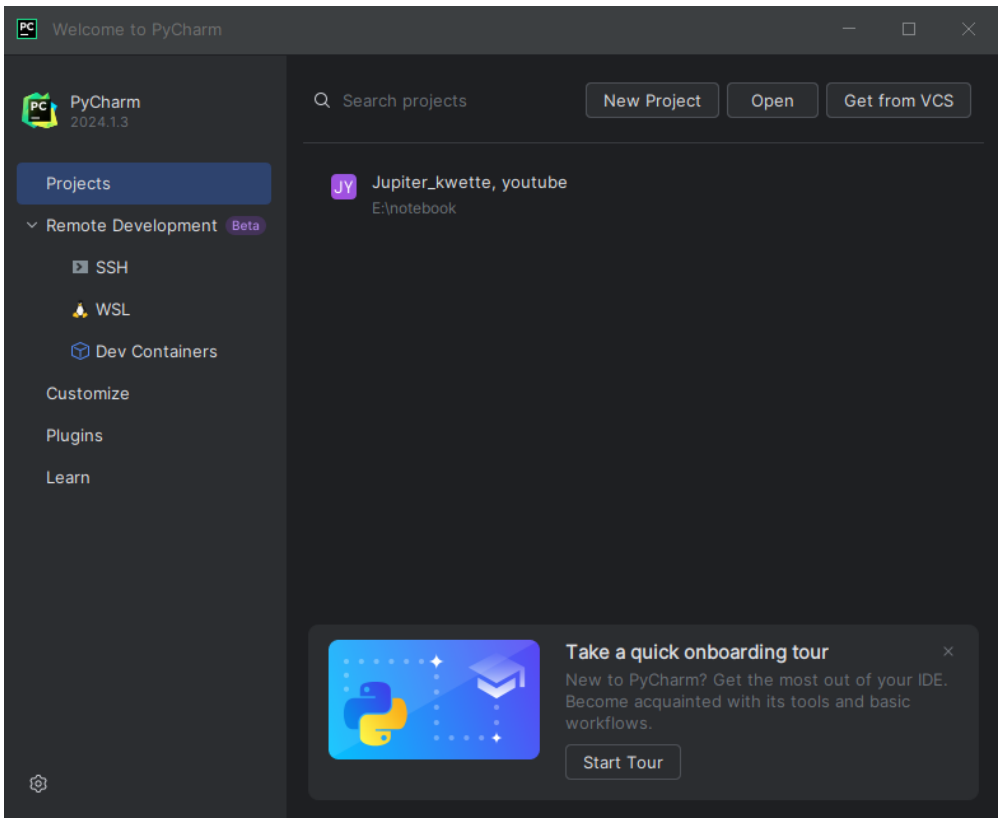
1.3.2 Prise en main de PyCharm

Nous allons maintenant prendre en main PyCharm. Je vous propose de commencer par démarrer l'IDE : l'exécutable se trouve dans le dossier $PYCHARM_HOME/bin/pycharm[.sh|.exe] (ou $PYCHARM_HOME représente le répertoire d'installation du logiciel).

Note :

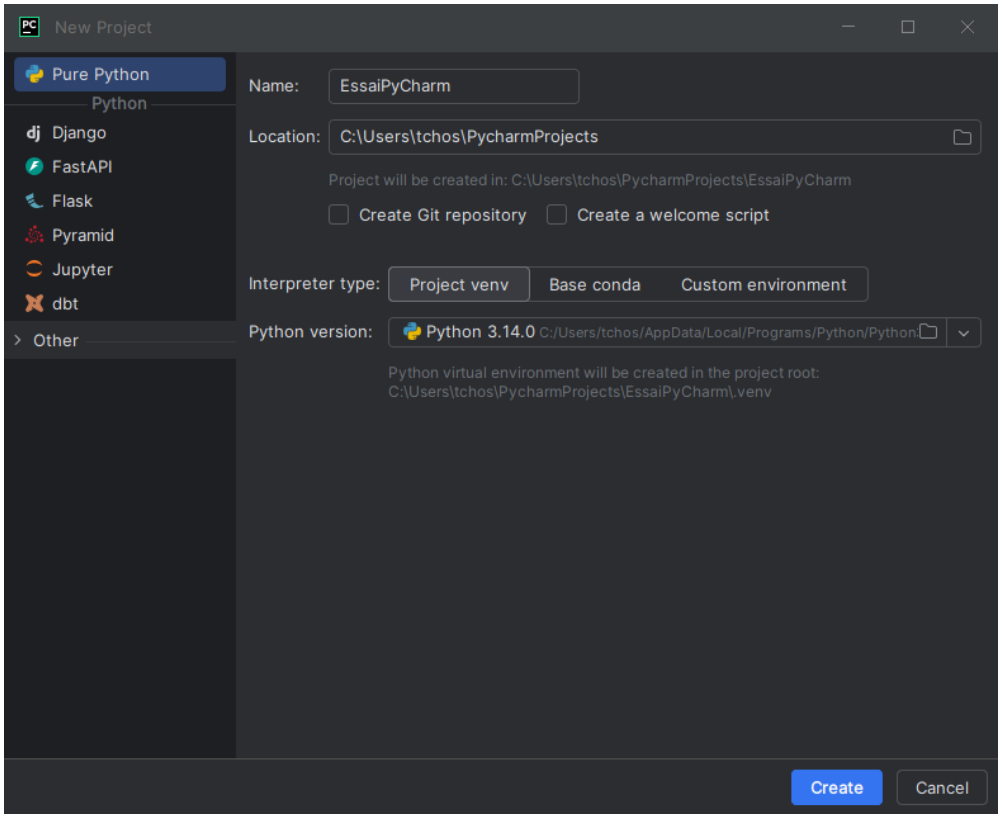
Sous Windows, vous pouvez bien entendu passer par le « **Menu Démarrer** » pour lancer l'outil.

Voici une capture de l'écran d'accueil initialement proposé.



Créer un projet PyCharm

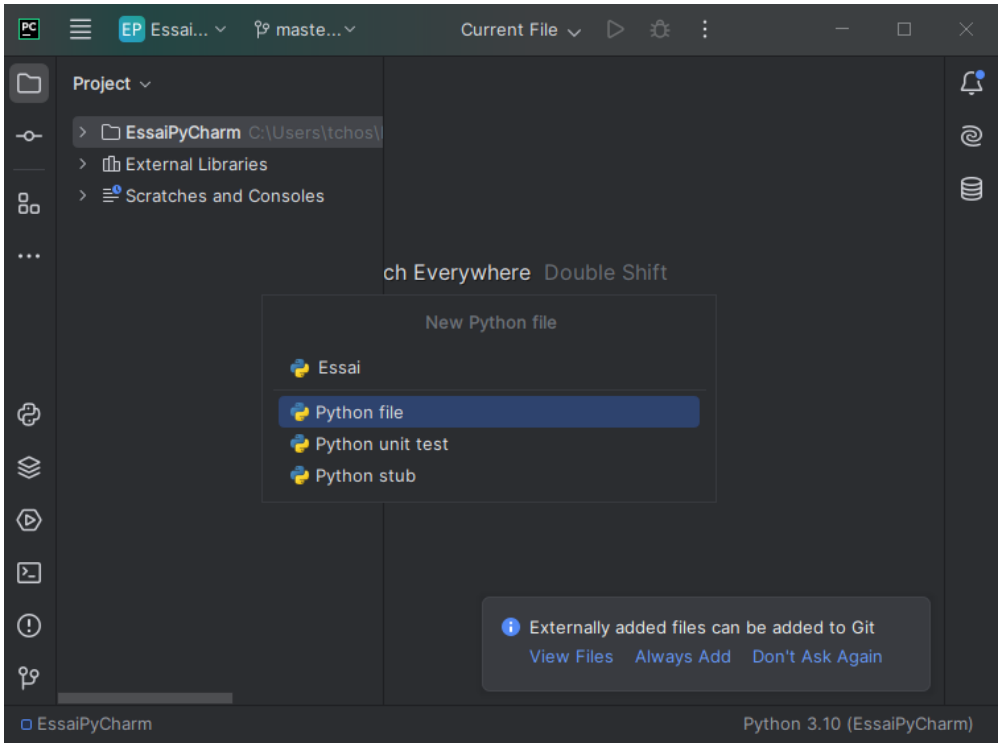
Pour créer un nouveau projet, vous devez cliquer sur le bouton « **Create New Projet** ». Un nouvel écran vous est alors proposé. Pensez à dérouler l'entrée « **Project Interpreter** » afin d'y voir l'ensemble des configurations proposées. Je vous conseille de rester avec une configuration de type « **New environment using VirtualEnv** ». Nous reviendrons sur ces notions d'environnements virtuels (venv) plus tard. Vérifiez juste que vous soyez bien configuré avec une version de Python 3.



Renseignez un nom (et sa localisation) de projet dans la zone d'édition proposée en haut de la fenêtre. Cliquez sur « **Create** » pour procéder à sa génération. Au bout de quelques secondes cette fenêtre doit apparaître.

Créer un module de code Python

En Python, un fichier de code est communément appelé « **un module** ». Pour créer un nouveau Fichier/Module de code Python, veuillez cliquer avec le bouton droit de la souris puis sélectionner « **New / Python File** ». Un popup doit apparaître : il faut y saisir en haut le nom du fichier à produire (l'extension du fichier est automatiquement ajoutée si vous l'omettez). En appuyant sur la touche « Entrée », le fichier **Essai.py** doit être produit.



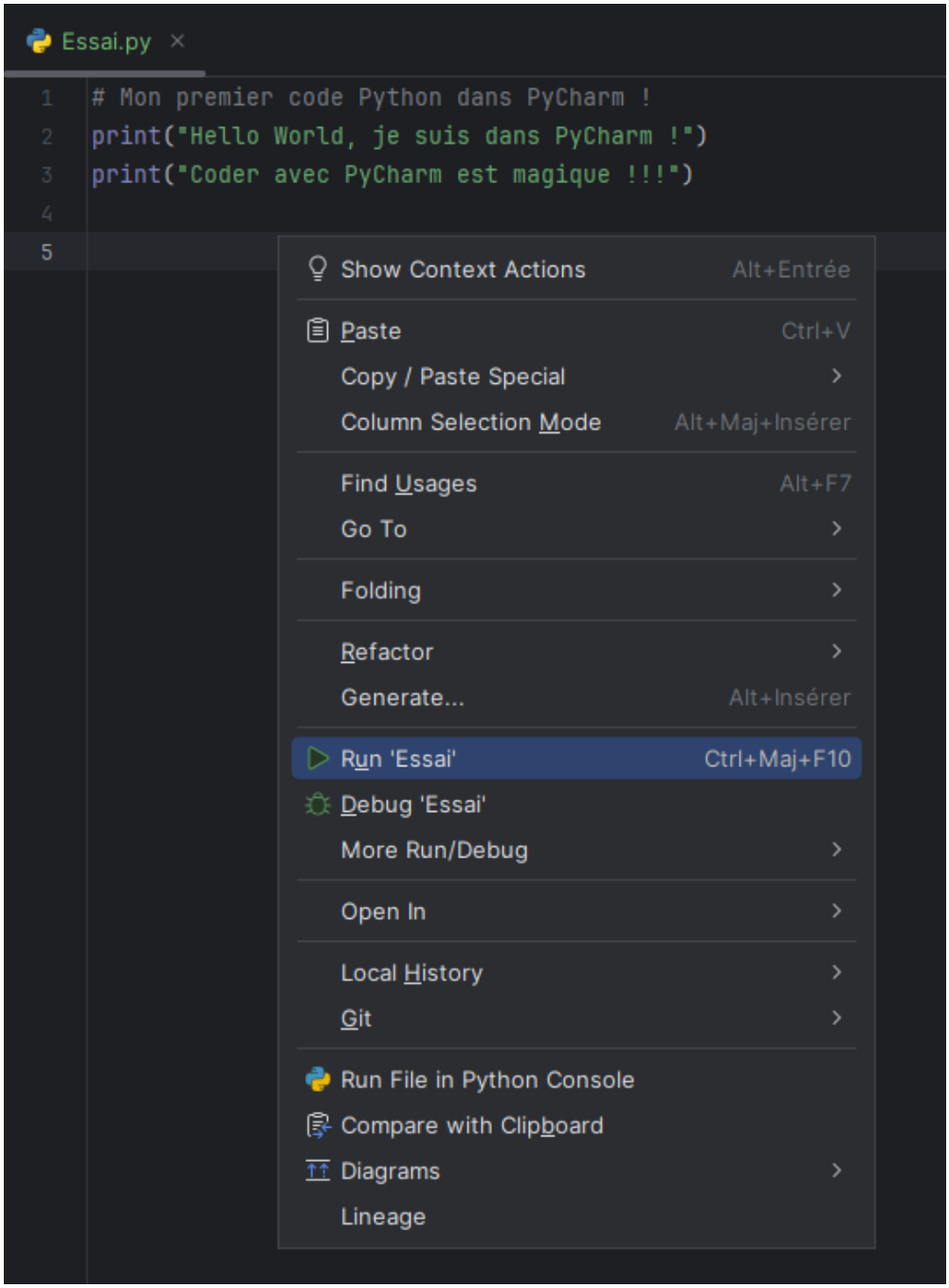
Editer votre script Python

Dans le but de tester l'éditeur PyCharm, je vous propose d'y saisir le programme suivant :

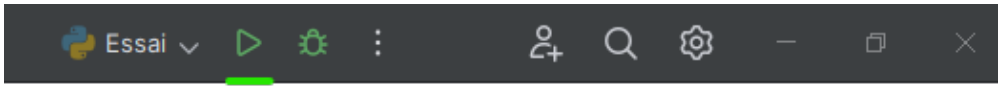
```
# Mon premier code Python dans PyCharm !
print("Hello World, je suis dans PyCharm !")
print("Coder avec PyCharm est magique !!!")
```

Lancer votre script Python

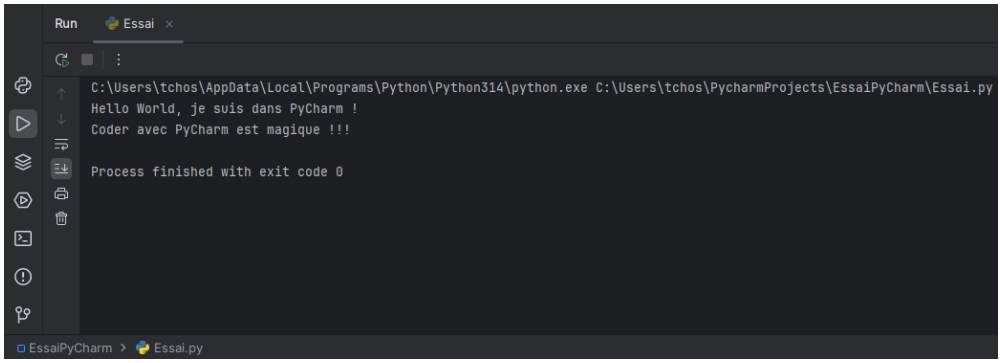
Une fois votre code édité, il ne reste plus qu'à le démarrer : pour ce faire, il faut cliquer avec le bouton droit de la souris dans le fichier en cours d'édition, puis sélectionner « **Run Essai** » comme le montre la capture d'écran ci-dessous. L'affiche du programme est réalisé dans la vue « **Run** » affichée en bas de la fenêtre de **PyCharm**.



On peut également exécuter son module python en cliquant sur le bouton en forme de triangle le vert comme indiqué sur la capture ci-dessous :



Après exécution le résultat vous est fourni dans le terminal intégré à **PyCharm**.



Maintenant que les outils qui vous permettront de coder sont désormais installés sur vos machines, votre apprentissage peut commencer. Rendez-vous au chapitre suivant.

Chapitre 2

Module 2 : Fondamentaux du langage Python

Objectifs :

— Comprendre la syntaxe de base et les types primitifs.
— Installer et configurer l'environnement (Python, pip, venv, IDE).

Contenu :

- Introduction à Python et son écosystème
- Hello World et les commentaires
- Les types natifs de données
- Interagir avec l'utilisateur
- Variables, opérateurs et expressions

Dans ce chapitre, vous apprendrez les concepts de bases de la programmation avec le langage python. Mais ne courons pas vite en besogne, il est utile pour vous apprenant de comprendre ce qu'est python et son écosystème. Mais comme dans tous l'apprentissage de tout langage de programmation, commençons par le « *Hello World* ».

2.1 Python et son écosystème

Python est aujourd'hui l'un des langages de programmation les plus populaires au monde. Adopté par les ingénieurs, scientifiques, chercheurs, entreprises technologiques et même les débutants, il se distingue par sa simplicité, sa lisibilité et sa polyvalence.

Créé par **Guido van Rossum** et publié en 1991, Python est conçu pour être intuitif, clair et proche du langage humain. Son nom ne vient pas du serpent, mais de la troupe humoristique *Monty Python*.

Philosophie : "*Write less, do more*" — *écrire moins de code pour faire plus.*

2.1.1 Les Forces de Python

Le tableau ci-dessous mets en lumière les atouts de python :

Atouts	Explications
Simplicité	Syntaxe claire, lisible, idéale pour débutants et experts
Polyvalence	Web, data science, IA, réseaux, cybersécurité, scripts, ...
Grande communauté	Millions d'utilisateurs, ressources, fora
Immense écosystème	Bibliothèques pour tous les domaines
Multiplateforme	Windows, macOS, Linux
Open source	Gratuit et évolutif

TABLE 2.1 – Atouts du langage python

2.1.2 Paradigmes de Programmation

Python supporte plusieurs paradigmes :

Paradigme	Exemple
Programmation procédurale	Scripts, automatisation
Programmation orientée objet (OOP)	Applications modulaires
Programmation fonctionnelle	map, filter, lambda, immutabilité
Programmation orientée données	Pandas, NumPy

TABLE 2.2 – Paradigmes de Programmation Supportés en python

Il s'adapte ainsi à divers styles de programmation.

2.1.3 Architecture et Fonctionnement

Python est un **langage interprété**. Le code source est exécuté par un interpréteur (CPython par défaut).

Le cycle d'exécution d'un programme python est le suivant :

1. Script Python (.py)
2. Conversion en bytecode
3. Exécution via la machine virtuelle Python (PVM)

Remarque : Python n'est pas compilé comme C++, mais **interprété**. Par ailleurs, mis à part **CPython** (Implémentation officielle et la plus utlisée), il existe d'autres variantes d'implémentations de python. Les plus populaires sont :

- **PyPy :** Plus rapide grâce au JIT.
- **Jython :** Intégré à la JVM.
- **IronPython :** Pour l'écosystème .NET.

2.1.4 Écosystème Python

Python doit sa puissance à son **immense bibliothèque standard** et ses **packages externes**. Nous allons vous exposer domaine par domaine, les bibliothèques les plus utilisées.

Les bibliothèques standards :

- Fichiers (os, shutil) ;
- Réseau (socket, requests) ;
- Sérialisation (json, pickle) ;
- Dates (datetime) ;
- Concurrence (threading, asyncio) ;
- Interfaces graphiques (tkinter).

Data Science & IA :

Python est le langage dominant en intelligence artificielle. Dans ce domaine, les bibliothèques les plus connues sont :

- **NumPy, Pandas :** Manipulation de données
- **Matplotlib / Seaborn / Plotly :** Visualisation
- **Scikit-learn :** Machine learning
- **TensorFlow / PyTorch :** Deep learning

Développement Web :

- **Django :** Applications web robustes.
- **Flask :** Micro-framework simple et rapide.
- **FastAPI :** API modernes ultra-rapides.

Automatisation & Scripting :

Python est excellent pour automatiser des tâches système (Renommer des fichiers, Automatiser des analyses, Scripts DevOps, Bots).

Les bibliothèques les plus utiles sont : **os, subprocess, pathlib, schedule**.

Cybersécurité & Pentesting :

Les **pentesters** adorent Python pour la rapidité de prototypage.

- **Scapy :** Manipulation de paquets réseau.
- **Paramiko :** SSH et automatisation.
- **Cryptography :** Chiffrement.
- **Pwntools :** Exploitation binaire.

Interfaces Graphiques :

- **Tkinter :** Interface native.
- **PyQt5 / PySide6 :** Interfaces avancées.
- **Kivy :** Apps mobiles / desktop.

Automation / IoT :

- OpenCV → Vision artificielle.
- ROS + Python → Robotique.
- MicroPython → Microcontrôleurs.

Python se distingue par sa simplicité, sa lisibilité et sa polyvalence mais il comporte quelques points faibles.

— Plus lent que C/C++ (langage interprété) ;
— Usage mémoire plus élevé ;
— Moins adapté au mobile natif.

Toutefois Python est un langage complet, moderne et incontournable. Sa simplicité, sa flexibilité et la richesse de son écosystème en font l'un des langages les plus influents pour l'avenir du développement informatique.

Apprendre Python, c'est investir dans un langage durable et indispensable.

Trêve de bavardage, il est temps de se jeter à l'eau.

2.2 Les bases en python

Pour cette section du cours, vous pouvez indépendamment utiliser *PyCharm* ou *Jupyter Notebook* pour écrire vos programmes.

2.2.1 Les types natifs de python

Ce qu'on appelle types natifs, ce sont des les types de données préexistants qui vous permette en programmation de représenter tout ce dont a besoin.

Avec Python (et en programmation en général), il existe plusieurs types natifs qui nous permettent de représenter toutes sortes de données. Parmi ces types natifs, les chaînes de caractères, les nombres et les booléens sont à la base de presque tout.

Les chaînes de caractère : str

Les chaînes de caractères permettent de représenter du texte. Elles sont définies par des guillemets simples ou doubles. Pour créer une chaîne de caractères, on doit entourer les caractères d'apostrophes (aussi appelées guillemets simples) ou de guillemets doubles :

```
#  variable_un et variable_deux sont de type str !
variable_un = 'apostrophes'
variable_deux = "guillemets"
```

Il n'y a **aucune différence entre les deux**, c'est une question de style !

Personnellement, je préfère utiliser les guillemets doubles pour la simple raison que ça permet d'insérer des / guillemets simples dans les phrases sans avoir besoin de les échapper avec un antislash.

Par exemple si je veux faire une chaîne de caractères pour indiquer mon âge :

```
#  age est une variable de type str !
age = "J'ai 26 ans"  # Avec guillemets doubles
age = 'J\'ai 26 ans'  # Avec guillemets simples
```

Vous pouvez également écrire des chaînes de caractères sur plusieurs lignes en utilisant trois guillemets au lieu d'un seul :

```
#  Vous pouvez utiliser des guillemets simples ou doubles.
longue_phrase = '''Bonjour,
bienvenue sur Docstring,
    on espère que vous appréciez
ce glossaire sur les chaînes
de caractères
'''
```

Les chaînes de caractères étant des séquences de caractères, on peut accéder à un élément en utilisant les crochets et la position de l'élément dans la chaîne de caractères :

```
site = 'docstring'

print(site[0])  # d
print(site[4])  # t
```

Vous pouvez accéder à plusieurs éléments en même temps grâce au *slicing* qui permet de préciser un intervalle de recherche avec un début, une fin et éventuellement un pas (un par un, deux par deux, trois par trois, etc). On doit séparer chaque valeur de l'intervalle par le symbole deux-points :

```
site = 'docstring'

print(site[0:3])  # 'doc'
print(site[3:])   # 'string'
print(site[1::2]) # 'osrn'
```

Attention ! ! ! La valeur de début de l'intervalle est **inclusive** alors quelle celle de fin est **exclusive** !

Modifier une chaîne de caractères Il n'est pas possible de directement modifier une chaîne de caractères. En effet, une chaîne de caractères est un objet immuable, ce qui signifie qu'on ne peut pas la modifier :

```
site = 'docstring'
site[2:] = 'py' # TypeError: 'str' object does not support item assignment
```

Astuce

Un objet immuable est un objet que l'on ne peut pas modifier. Parmi les objets immuables, on retrouve notamment :

- Les nombres entiers (int)
- Les nombres décimaux (float)
- Les chaînes de caractères (str)
- Les booléens (bool)
- Les tuples (tuple)

Les nombres

Avec python on fait la distinction entre 2 types de nombres : les nombres entiers et les nombres décimaux.

Les nombres entiers : int Un nombre entier est un nombre qui ne possède pas de décimales (chiffres après la virgule). En anglais, on appelle ces nombres des 'integer' (*int* en raccourci).

Quelques exemples de nombres entiers :

```
# print est une fonction native de python qui sert d'affichage d'une expression.
print(0)
print(-1)
print(6)
print(-9384)
```

```
print(-205)
print(100000)
```

Les nombres entiers peuvent être positifs ou négatifs. 0 est également considéré comme un nombre entier.

Pour créer un nombre entier, on peut également utiliser la classe *int* :

```
# Creattion de 2 entiers a et b en se servant de la classe int.
a = int(5)
b = int(-2)
```

Dans la pratique, la classe *int* est surtout utilisée pour convertir un objet en nombre entier :

```
# Conversion d'un float en int
a_float = 10.7542
a_int = int(a_float)
print(a_int)
```

Remarques :

- En convertissant un nombre décimal en nombre entier, on se retrouve uniquement avec les chiffres avant le point. Python n'arrondit donc pas le nombre, il se contente d'enlever les décimales.
- Vous ne pouvez passer à la classe *int* que des nombres décimaux, des chaînes de caractères qui contiennent des nombres et des booléens (qui sont un sous- type des nombres entiers).

Si vous essayez de convertir un autre type d'objet en nombre entier, vous aurez une erreur de type *ValueError* :

```
# Conversions valides
print(int(5.0))
print(int(True))
print(int(False))
print(int("1587"))
print(int("-541.254"))

# Erreurs de type ValueError ou TypeError
print(int(None))
print(int("Bonjour"))
print(int([1, 2, 3]))
```

Les nombres décimaux : float Un nombre décimal est un nombre qui peut s'écrire avec un nombre fini de chiffres après la virgule.

Avec Python, on utilise un point (notation anglaise) pour séparer les décimales et non une virgule. En anglais, on appelle ces nombres des 'floating point numbers' (*float* en raccourci). Vous retrouverez donc également l'appellation de 'nombres flottants'.

Quelques exemples de nombres décimaux :

```
print(0.0)
print(10.2)
print(3.14159265359)
print(1938.23423)
print(-205.344)
print(-1.0)
```

Un nombre est considéré comme un nombre décimal même si les chiffres après le point sont des 0. Ainsi, 0.0 est un nombre décimal. Nous pouvons le vérifier avec la fonction *isinstance* :

```
print(isinstance(0.0, float))  # True
print(isinstance(0.0, int))    # False
```

Pour créer un nombre décimal on peut également utiliser la classe *float* :

```
a = float(5.0)
b = float(10.45)
```

Remarques :

- Vous ne pouvez passer à la classe *float* que des nombres entiers, des chaînes de caractères qui contiennent des nombres et des booléens (qui sont un sous- type des nombres entiers).
- Si vous essayez de convertir un autre type d'objet en nombre décimal, vous aurez une erreur.

```
# Conversions valides
print(float(5))
print(float(True))
print(float(False))
print(float("10.4"))

# Erreurs de type ValueError ou TypeError
print(float(None))
print(float("Bonjour"))
print(float([1, 2, 3]))
```

Remarques :

- Vous ne pouvez passer à la classe *float* que des nombres entiers, des chaînes de caractères qui contiennent des nombres et des booléens (qui sont un sous- type des nombres entiers).
- Si vous essayez de convertir un autre type d'objet en nombre décimal, vous aurez une erreur.

Les booléens

Un booléen est un type natif de Python qui n'a que deux valeurs possibles : *True* ou *False*.

Les booléens sont un sous-type du type natif des nombres entiers :

```
issubclass(bool, int)  # True
```

Le booléen *True* est donc similaire au nombre *1* et le booléen *False* au nombre *0*.

Il est ainsi possible de réaliser des opérations mathématiques entre un booléen et un entier :

```
# opérations mathématiques entre un booléen et un entier
print(True + 1)
print(True - 1)
print(False + 1)
print(False - 1)
print(True * 5)
print(False * 5)
print(True ** 4)
```

Cela peut sembler inutile, mais c'est très pratique si vous devez par exemple établir un compteur qui se base sur le résultat d'une expression qui résulte sur un booléen. Par exemple, le code suivant :

```
utilisateur_inscrit = True
nombre_utilisateurs = 0
if utilisateur_inscrit:
    nombre_utilisateurs += 1 #

print(nombre_utilisateurs)
```

NB : Tous les objets de Python peuvent être évalués comme des booléens. Pour évaluer (ou convertir) un objet en tant que booléen, on utilise la classe *bool* :

```
print(bool("Bonjour"))
print(bool(1))
print(bool([]))
print(bool((1, 2, 3)))
print(bool({1: "un", 2: "deux"}))
```

En effet, tous les types natifs possèdent par défaut une valeur qui est évaluée comme *False*, toutes les autres étant évaluées comme *True*. Par exemple, tous les types natifs « **vides** » seront évalués comme *False* :

```
print(bool("Bonjour"))
print(bool(1))
```

```
print(bool([]))
print(bool((1, 2, 3)))
print(bool({1: "un", 2: "deux"}))
```

Pour d'autres objets, ce seront des valeurs spécifiques, comme le *0* pour les nombres entiers, *0.0* pour les nombres décimaux ou encore le mot clé *None* :

```
print(bool(0))
print(bool(0.0))
print(bool(None))
```

Toutes les autres valeurs de ces types retourneront *True* quand vous les passez à la classe *bool*.

NB : Il est même possible d'implémenter la méthode spéciale *bool* pour définir le comportement d'un objet avec la classe *bool* :

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

    def __bool__(self):
        return self.age >= 18

patrick = Person(prenom="Patrick", nom="Michaud", age=14)
john = Person(prenom="John", nom="Tremblay", age=20)

print(bool(patrick))  # False car Patrick a moins de 18 ans
print(bool(john))  # True car Patrick a plus de 18 ans
```

Il existe d'autres types natifs qui peuvent être construits à partir de ces 3 types de base. Ce sont :

- **Les types séquentiels** : listes ([1,"a",3]), tuples (("Oui", "Non")), range (n = rang(0,5)) ;
- **Les types d'ensemble** : set (Ensemble non ordonné, unique. Ex : s = 1, 2, 2, 3 devient 1, 2, 3), frozenset (Version immuable de set. Ex : fs = frozenset(1, 2, 3)) ;
- **Les types de correspondances ou de mappage** : les dictionnaires ("nom" : "Tchos", "age" : 50) ;
- **bytes** : données binaires immuables (Ex : b = b"Hello") ;
- **bytearray** : données binaires modifiables (Ex : ba = bytearray(5)) ;
- **memoryview** : vue sur un buffer mémoire (Ex : mv = memoryview(b"data")) ;
- **NoneType** : absence de valeur, représentée par `None`

Etant plus complexes, ils feront l'objet d'une étude particulière dans des chapitres qui leur sont réservés dans ce cours.

Le tableau ci-dessous récapitule les types de données reconnus en Python.

Catégorie	Types
Numériques	int, float, complex
Booléen	bool
Texte	str
Séquences	list, tuple, range
Mappage	dict
Ensembles	set, frozenset
Binaires	bytes, bytearray, memoryview
Spécial	NoneType

2.2.2 Les variables

Une variable un emplacement mémoire dont la valeur peut changer au cours de l'exécution d'un programme.

- Une variable possède un **nom**, on parle d'identifiant ;
- Une variable possède une **valeur** ;
- Une variable possède un **type** qui caractérise l'ensemble des valeurs que peut prendre la variable.

Il faut la voir un peu comme un sac à dos dans lequel vous rangez vos affaires.

```
sac_a_dos = ['stylo', 'cahier', 'gomme']
```

Vous pouvez ainsi y accéder quand vous le souhaitez sans avoir besoin de recréer à chaque fois l'objet auquel est associé la variable :

```
sac_a_dos = ['stylo', 'cahier', 'gomme']
print(sac_a_dos)
```

Vous pouvez redéfinir à n'importe quel moment une variable en y associant un nouvel objet :

```
sac_a_dos = ['stylo', 'cahier', 'gomme']
sac_a_dos = ['stylo', 'cahier', 'banane'] # pour le goûter
print(sac_a_dos)
```

Comment ça marche ? En Python, on n'assigne pas vraiment de valeurs aux variables. Au lieu de cela, Python donne la référence de l'objet (valeur) à la variable. La variable devient alors un **pointeur** vers cette référence. Pour illustrer ça, j'aime bien me le représenter avec le concept des prénoms : Des milliers de personnes portent le même prénom. Toutes ces personnes sont différentes et pourtant on peut les appeler par le même prénom.

Par exemple :

```
prenom = 'Paul'
autre_prenom = prenom
print(prenom)
print(autre_prenom)
```

Quand j'exécute ce code, Python interprète l'instruction *prenom = 'Paul'*, il crée un nouvel objet avec la valeur *'Paul'* et crée un lien entre la variable *prenom* et cet objet.

Sur la deuxième ligne, Python interprète *autre_prenom = prenom* et vérifie que cet objet n'existe pas déjà en mémoire. Comme c'est le cas, il ne va pas le recréer. Il va juste créer un nouveau lien entre cette variable *autre_prenom = prenom* et l'objet dont la valeur est *'Paul'*.

On peut utiliser la fonction id pour vérifier l'identifiant de l'espace mémoire dans lequel se trouve un objet associé à une variable :

```
prenom = 'Paul'
autre_prenom = prenom

# La fonction id retourne 2 pour les deux variables.
# Nous accédons donc au même objet (la chaîne de caractères 'Paul') avec deux
↪  variables différentes
print(id(prenom))        # 2
print(id(autre_prenom))  # 2

# Si on associe deux variables à deux objets différents, les id seront différents
↪  :
prenom = 'Patrick'
autre_prenom = 'Pierre'

# Les id sont différents, car nous accédons à deux objets différents, avec deux
↪  noms différents :
print(id(prenom))        # 3
print(id(autre_prenom))  # 4
```

La portée des variables

En Python, comme dans beaucoup d'autres langages, chaque variable est accessible dans un périmètre précis. On parle de *scope* en anglais.

Une variable définie à l'intérieur d'une fonction (dans l'espace local de la fonction) ne sera pas disponible à l'extérieur.

Par exemple :

```
def hello_paul():
    prenom = 'Paul'
    print('Hello {}'.format(prenom))

print(prenom)
```

À l'inverse une variable définie dans l'espace global (à l'extérieur de la fonction) sera disponible à l'intérieur de celle-ci :

```
prenom = 'Paul'

def hello_paul():
```

```
    print('Hello {}'.format(prenom))

hello_paul()
```

Assigner une valeur à une variable

Comme vous l'avez vu dans les exemples pris plus dans le cours, on utilise le symbole égal = pour assigner une valeur à une variable :

```
# Opération d'affectation d'une valeur à une variable
prenom = 'Paul'
age = 24
homme = True
```

Python est un langage **dynamique**, nous n'avons pas besoin de lui dire qu'un nombre est un nombre ou qu'une chaîne de caractères est une chaîne de caractères.

Les assignations multiples

Comme nous l'avons vu au début de cette sous-section, Python n'assigne pas vraiment de valeurs aux variables, il crée des liens entre des noms et des objets. De ce fait, on peut lier plusieurs variables au même objet de cette manière :

```
# La meme valeur est assignée à a, b, c
a = b = c = 750
id(a)  # 2591630728144
id(b)  # 2591630728144
id(c)  # 2591630728144
```

Les assignations parallèles

On peut aussi demander à Python d'assigner plusieurs variables à plusieurs objets différents de cette façon :

```
# Exemple 1:
a, b, c = 1, 2, 3
print(a)
print(b)
print(c)

# Exemple 2:
a, *b, c = (1, 2, 3, 4, 5, 6)
print(a)
print(b)
print(c)
```

```python
# Exemple 3:
a, b, c = {1: "un", 2: "deux", 3: "trois"}
print(a)
print(b)
print(c)

# Exemple 4:
i, c, t = "ICT" # Avec une chaîne de caractères
print(i)
print(c)
print(t)
```

Cette technique très précieuse car nous fait gagner énormément dans la réduction des lignes de codes, est appelée en python *L'unpacking*.

Les constantes

En théorie, une constante est un **type de variable qu'on ne peut pas modifier**. Avec Python, on les déclare toujours en majuscule :

```python
PI = 3.14
PRIMARY_COLOR = "#FF00FF"
```

Mais en pratique, avec Python, les constantes n'en sont pas vraiment car on peut leur assigner des valeurs différentes à n'importe quel moment !

C'est plus une convention que respectent les développeurs Python et qui permet de séparer visuellement les variables destinées à être modifiées de celles qui ne le sont pas.

Concaténer des variables

Pour concaténer des variables, on utilise l'opérateur d'addition (+) :

```python
prenom = 'Paul'
nom = 'Sims'

print(prenom + ' ' + nom)
```

En plus d'être un langage **dynamique** , Python est aussi un langage **fortement typé**. Cela signifie que vous ne pouvez pas mélanger des chaînes de caractères avec des nombres ou d'autres types et vice-versa.

Si vous écrivez :

```python
prenom = 'Paul'
age = 24

print(prenom + age) # TypeError
```

Dans ce cas-là vous devez convertir le type de votre variable pour pouvoir la concaténer. En l'occurrence, on peut convertir le nombre *24* en chaîne de caractères *'24'* grâce à la fonction *str()*

```
prenom = 'Paul'
age = 24

print(prenom + ' ' + str(age))
```

Règles et conventions de nommage

Il existe plusieurs conventions très suivies dans la communauté Python que vous retrouverez sous le nom de PEP8 (Les Python Enhancement Proposals, abrégées PEP, signifient en français : Propositions d'Amélioration de Python). Parmi ces conventions, on retrouve de nombreuses conventions de nommage qui indique les règles à suivre lorsque l'on crée des noms de variable. Ainsi, il est conseillé d'utiliser uniquement des lettres en minuscule et des tirets du bas pour séparer les mots.

Les règles de nommage d'une variable :

- Le nom d'une variable ne doit pas commencer par un chiffre ;
- Un nom de variable ne doit pas contenir d'espace ;
- Un nom de variable ne doit contenir que des caractères aplha-numériques et l'underscore (_) ;
- Certains mots sont réservés en python et e doivent en conséquence pas être utilisés pour définir des variables.

Pour vore culture, voici la liste des mots-clés principaux de python :

- **Logiques et conditions** : and, or, not, if, elif, else, in, is, assert
- **Boucles** : for, while, break, continue
- **Fonctions et classes** : def, return, class, yield, nonlocal, lambda, async, await
- **Gestion des exceptions** : try, except, finally, raise
- **Structure et contrôle** : with, pass, del, global
- **Opérateurs** : None, True, False

Voici quelques exemples de noms de variables qui respectent ces règles : *website_url*, *number_of_students*.

Voici quelques contre-exemples : *StudentID*, *Lastname*, *nom personne*, *6personne*, *Site-Web*, *#reponse*.

Votre script fonctionnera si vous ne suivez pas ces conventions. Mais elles sont très respectées dans la communauté Python et permettent d'avoir une uniformité très agréable parmi les scripts sur lesquels vous allez travailler.

Si vous respectez toutes ces conventions et que vous vous habituez à les utiliser, vous verrez qu'avec en plus la syntaxe épurée de Python, il vous sera très facile de lire des scripts d'autres développeurs.

2.2.3 Les types des objets et la conversion

Dans cette section, il s'agira de vous montrer comment changer le type d'une variable. Il y aura des situations qui nous pousseront à vouloir changer le type d'une variable :

— Pour utiliser les opérateurs mathématiques sur des variables ;
— Pour faire des comparaisons entre des variables.

Ce qu'il faut comprendre c'est que *50* est bien différent de *"50"* car l'un est un nombre et l'autre une chaîne de caractères et donc deux types de données différents. N'oubliez jamais que Python est un langage de programmation **dynamique** et **fortement typé**. Pour effectuer des opérations sur de variables, il faudrait qu'elles soient toutes du même type.

FIGURE 2.1 – Exemple d'addition entre 2 variables de type différent

Pour régler ce genre de problème, nous aurons besoin de changer le type de nos variables et c'est alors qu'entre en jeu la conversion.

Afficher le type d'une variable

Pour afficher le type d'une variable, Python vous offre la fonction *type()*.

```
a = 3
b = "14"
print(type(a)) # <class 'int'>
print(type(b)) # <class 'str'>
```

Les fonction de conversion int et str

La fonction *int* va permettre de convertir une chaîne de caractères (constituée uniquement des chiffres) en nombre afin de pourvoir utiliser ladite chaîne comme un nombre et d'y effectuer des opérations mathématiques.

```
a = 3
b = "14"
b = int(b)
print(a + b)
```

Pareillement, pour convertir un nombre entier en chaîne de caractères, on utilise la fonction *str*, ce qui est nécessaire pour effectuer des concaténations.

```
a = 2
b = 6
c = 3
# Pour concaténer des éléments ensemble on peut utiliser le symbole "+"
resultat = str(a) + " + " + str(b) + " + " + str(c)
```

2.2.4 Interagir avec l'utilisateur

Il est très courant dans un script d'avoir besoin à un moment ou un autre de demander des informations à l'utilisateur. On pourrait par exemple demander la nouvelle taille d'une image qu'on souhaite modifier, l'URL d'un site web ou encore un chemin de dossier dans lequel on veut écrire des données.

Pour demander des informations à un utilisateur dans un script en ligne de commande, on utilise avec Python la fonction *input* .

La fonction input accepte un argument qui correspond au texte que l'on souhaite afficher à l'utilisateur. Par exemple, si on souhaite demander l'âge de l'utilisateur :

```
input("Quel âge avez-vous ? ")
```

L'utilisateur peut ensuite écrire à la fin de la phrase affichée. La saisie de l'utilisateur est ensuite retournée par la fonction *input*.

NB : Depuis la version 3 de Python, la fonction input retourne systématiquement une chaîne de caractères. Ainsi, la variable age dans l'exemple ci-dessus ne sera pas considérée comme un nombre entier mais comme une chaîne de caractères contenant le nombre 25, ce que l'on peut vérifier avec la fonction type :

```
age = input("Quel âge avez-vous ? ")
print(type(age))
```

Si vous souhaitez convertir la variable *age* en nombre entier, il faudra donc utiliser la fonction de conversion *int* .

Dans la section suivante nous allons voir comment manipuler les objets de types chaînes de caractères.

2.2.5 Manipuler les chaînes de caractères

Avec python il existe beaucoup de façons de modifier des objets avec python. Pour cela on utilisera des fonctions sur un objet. Dans cette partie nous allons nous intéresser aux objet de type *chaînes de caractères*. On peut effectuer beaucoup d'opération sur des objets de type *chaînes de caractères*.

Voici ci-dessous une liste exhaustive de toutes les méthodes que tu peux utiliser sur une chaîne de caractères ainsi que leur utilité :

Méthodes Python pour manipuler les chaînes de caractères

Méthode	Description	Retour
capitalize()	Met en majuscule le premier caractère	`str`
casefold()	Minuscule agressive (comparaisons)	`str`
center()	Centre la chaîne	`str`
count()	Compte les occurrences	`int`
encode()	Encode la chaîne	`bytes`
endswith()	Test fin de chaîne	`bool`
expandtabs()	Remplace tab par espaces	`str`
find()	Index ou `-1` si absent	`int`
format()	Formate la chaîne	`str`
index()	Comme find mais erreur si absent	`int`
isalnum()	Alphanumérique ?	`bool`
isalpha()	Alphabétique ?	`bool`
isdecimal()	Décimal ?	`bool`
isdigit()	Chiffres ?	`bool`
isidentifier()	Identifiant Python valide ?	`bool`
islower()	Tout en minuscules ?	`bool`
isnumeric()	Numérique ?	`bool`
isprintable()	Imprimable ?	`bool`
isspace()	Espaces ?	`bool`
istitle()	Style titre ?	`bool`
isupper()	Tout en majuscules ?	`bool`
join()	Concatène un iterable	`str`
ljust()	Justifie à gauche	`str`
lower()	Minuscule	`str`
lstrip()	Supprime début	`str`
maketrans()	Table de traduction	`mapping`
partition()	Coupe en 3 parties	`tuple`
replace()	Remplace une sous-chaîne	`str`
rfind()	Index depuis la droite	`int`
rindex()	Comme rfind mais erreur si absent	`int`
rjust()	Justifie à droite	`str`
rpartition()	Coupe depuis droite	`tuple`
rsplit()	Split depuis la droite	`list`
rstrip()	Supprime fin	`str`
split()	Découpe	`list`
splitlines()	Découpe par lignes	`list`
startswith()	Test début	`bool`
strip()	Trim début/fin	`str`

swapcase()	Inverse casse	str
title()	Met chaque mot en capitales	str
translate()	Traduction	str
upper()	Majuscule	str
zfill()	Remplit avec zéros	str

Astuce

Pour afficher l'aide d'une méthode, vous pouvez utiliser la fonction *help* (par exemple *help(str.center)* pour afficher l'aide de la méthode *center*).

```
[20]: "1253".isdigit()

[20]: True

[22]: "Les étudiants de ICT".title()

[22]: 'Les Étudiants De Ict'

[24]: "Les étudiants de ICT".isalpha()

[24]: False

[26]: intab = "aeiou"
      outtab = "12345"
      trantab = str.maketrans(intab, outtab)

      string_original = "this is a test string"
      string_traduit = string_original.translate(trantab)

      print(string_traduit)

      th3s 3s 1 t2st str3ng

[28]: texte = "Python est puissant"
      resultat = texte.partition("est")
      print(resultat)

      ('Python ', 'est', ' puissant')

[30]: "123".zfill(9)

[30]: '000000123'
```

FIGURE 2.2 – Quelques exemples d'utilisation des méthodes python sur des chaînes de caractères

2.2.6 Les opérateurs en Python

Intéressons-nous maintenant aux opérateurs. Python offre plusieurs catégories d'opérateurs permettant d'effectuer des calculs, des comparaisons, des tests logiques, des manipulations de bits et des tests d'identité et d'appartenance.

Voici la liste complète des opérateurs utilisables en Python, classés par catégories avec exemples.

Opérateurs arithmétiques

Opérateur	Description	Exemple
+	Addition	`3 + 2 = 5`
-	Soustraction	`5 - 3 = 2`
*	Multiplication	`4 * 2 = 8`
/	Division	`7 / 2 = 3.5`
//	Division entière	`7 // 2 = 3`
%	Modulo (reste)	`7 % 2 = 1`
**	Puissance	`2 ** 3 = 8`

Opérateurs d'affectation

Opérateur	Signification
=	Affectation simple
+=	Addition puis affectation
-=	Soustraction puis affectation
*=	Multiplication puis affectation
/=	Division puis affectation
//=	Division entière puis affectation
%=	Modulo puis affectation
**=	Puissance puis affectation

Opérateurs de comparaison

Opérateur	Description	Exemple
==	Égal à	`5 == 5 → True`
!=	Différent de	`5 != 3 → True`
>	Supérieur à	`5 > 3 → True`
<	Inférieur à	`3 < 5 → True`
>=	Supérieur ou égal	`5 >= 5 → True`
<=	Inférieur ou égal	`3 <= 4 → True`

Opérateurs logiques

Opérateur	Description
and	Vrai si les deux conditions sont vraies
or	Vrai si au moins une condition est vraie
not	Inverse la valeur booléenne

Opérateurs d'appartenance

Opérateur	Description
in	Vrai si un élément appartient à une séquence
not in	Vrai si un élément n'appartient pas à une séquence

Opérateurs d'identité

Opérateur	Description
is	Vrai si les deux objets sont identiques (même adresse mémoire)
is not	Vrai si les deux objets sont différents

Opérateurs bit-à-bit

Opérateur	Description	Exemple
&	ET bit-à-bit	`5 & 3 = 1`
|	OU bit-à-bit	`5 | 3 = 7`
^	XOR bit-à-bit	`5 ^ 3 = 6`
~	NON bit-à-bit	`~5 = -6`
«	Décalage à gauche	`5 « 1 = 10`
»	Décalage à droite	`5 » 1 = 2`

La différence entres les opérateurs *is* et ==

C'est une erreur que font souvent la plupart des débutants dans leur apprentissage en python. En Python, == (test d'égalité des valeur) et *is* (test d'identité (référence mémoire)) sont tous les deux des opérateurs de comparaison, mais ils ne comparent pas la même chose.

L'opérateur == vérifie si les valeurs de deux variables sont égales. On compare le contenu tandis que l'opérateur *is* vérifie si deux variables pointent vers le même objet en mémoire, ici on compare l'identité des objets.

Le code ci-dessous, met en exergue la grosse différence qui existe entre les deux opérateurs.

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a == c)  # True  -> mêmes valeurs
print(a is c)  # False -> objets différents

print(id(a))
print(id(c))   # Pour confirmer que a pointe vers le même objet que c

print(a == b)  # True
print(a is b)  # True -> b pointe vers le même objet que a
print(id(b))   # Pour confirmer que b pointe vers le même objet que a
```

2.2.7 Le formatage des chaînes de caractères

Il existe plusieurs façons de faire de la concaténation dans une chaîne de caractères. Depuis la version *3.6* de Python, une des façons de faire les plus efficaces est d'utiliser les *f-string*.

Le terme de « formatage » de chaînes n'a rien à voir avec le formatage d'un disque dur. Par « formatage », on entend « **mise en forme** » et il existe de nombreuses façons de mettre en forme

des chaînes de caractères avec Python.

Au commencement, il y avait la concaténation

Avant de parler de formatage, on parle souvent de concaténation. La concaténation peut se faire très simplement avec l'opérateur mathématique + :

```
protocole = "https://"
nom_du_site = "Docstring"
extension = "fr"

url = protocole + "www." + nom_du_site + "." + extension
print(url)
```

Ça fonctionne, mais très rapidement vous allez observer deux problèmes :

— Ce n'est pas très agréable à lire.

— Vous ne pouvez concaténer que des objets du même type (car Python est un langage fortement typé).

Ainsi, le code suivant retournera une erreur car la variable age est un nombre entier :

```
age = 26
phrase = "J'ai " + age + " ans."
```

```
[32]: age = 26
      phrase = "J'ai " + age + " ans."

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 2
      1 age = 26
----> 2 phrase = "J'ai " + age + " ans."

TypeError: can only concatenate str (not "int") to str
```

Pour palier à cette erreur, il faut convertir la variable age en chaîne de caractères :

```
age = 26
phrase = "J'ai " + str(age) + " ans."
```

Et on retombe sur le premier problème : ça devient très vite difficile à lire et c'est le meilleur moyen de se retrouver avec des *TypeError*.

Il existe un autre moyen d'insérer des objets dans une chaîne de caractères qui permet de spécifier directement le type des données insérées. Pour ce faire, on utilise l'opérateur modulo (%) et on spécifie ensuite le format des données que l'on insère dans la chaîne de caractères (**Attention !!!** Cette façon de faire est obsolète mais on continue de la retrouver dans de nombreux scripts Python. Il est important de comprendre donc de quoi il en retourne.).

```
age = 26
phrase = "J'ai %d ans." %age  # "J'ai 26 ans."
phrase = "J'ai %f ans." %age  # "J'ai 26.000000 ans."
```

— En utilisant %d on se retrouve avec un nombre entier.
— En utilisant %f, la variable age, qui est au départ un nombre entier, se retrouve dans la chaîne de caractères sous forme de nombre décimal.

Cette façon de faire a été longtemps la méthode privilégiée pour formater des chaînes de caractères, raison pour laquelle vous la verrez souvent dans certains scripts.

Certains continuent même d'utiliser cette façon de faire alors qu'il existe maintenant des méthodes bien plus efficaces comme celles que nous allons voir dans la suite

La méthode format

Pour éviter les erreurs de type et insérer des objets dans une chaîne de caractères, on peut utiliser la méthode *format*. Cette méthode permet d'insérer des objets à l'intérieur 'emplacements spécifiés dans la chaîne de caractères par des accolades :

```
age = 26
phrase = "J'ai {} ans".format(age)
```

Ici, pas besoin de convertir la variable age en chaîne de caractères. La méthode format s'en charge pour nous et nous évitons ainsi le *TypeError*.

On peut également spécifier à l'intérieur des accolades un nom qu'on utilisera comme paramètre dans la méthode *format* pour obtenir une phrase encore plus claire :

```
age_de_lutilisateur = 26
phrase = "J'ai {age} ans".format(age=age_de_lutilisateur)
```

On peut mettre autant d'emplacements dans une chaîne de caractères que nécessaire. Il faudra cependant faire attention de passer autant d'éléments à la méthode format qu'il y a d'emplacements dans notre chaîne de caractères.

Par exemple, le code suivant fonctionne :

```
prenom = "Pierre"
age = 26
phrase = "Je m'appelle {} et j'ai {} ans.".format(prenom, age)
```

Mais le code ci-dessous retournera une erreur (*IndexError* : tuple index out of range) car il y a deux emplacements dans la chaîne de caractères mais un seul élément est passé à la méthode format :

```
prenom = "Pierre"
age = 26
phrase = "Je m'appelle {} et j'ai {} ans.".format(prenom)
```

On peut également spécifier à l'intérieur des accolades un indice, qui fera référence à la position de l'argument de la méthode format à utiliser. C'est très pratique si vous souhaitez réutiliser un argument :

```
prenom = "Pierre"
age = 26
langage = "Python"
phrase = "Je m'appelle {0} et j'ai {1} ans. {0} apprends le langage
↪  {2}.".format(prenom, age, langage)
# Je m'appelle Pierre et j'ai 26 ans. Pierre apprends le langage Python.
```

On peut également remettre des noms à l'intérieur des accolades afin d'obtenir quelque chose de plus agréable à lire :

```
prenom = "Pierre"
age = 26
langage = "Python"
phrase = "Je m'appelle {prenom} et j'ai {age} ans. {prenom} apprends le langage
↪  {langage}.".format(prenom=prenom, age=age, langage=langage)
```

NB : Le nom que vous mettez à l'intérieur des accolades n'a pas besoin d'être le même que la variable que vous passez à la méthode format. On aurait très bien pu faire :

```
prenom = "Pierre"
age = 26
langage = "Python"
phrase = "Je m'appelle {user} et j'ai {age} ans. {user} apprends le langage
↪  {learning}.".format(user=prenom, age=age, learning=langage)
```

La méthode format permet donc d'obtenir un code beaucoup plus facile à lire et évite de s'embêter avec les fonctions de conversion mais il y a mieux encore.

Les f-string

Depuis la version *3.6* de Python, une fonctionnalité très intéressante a fait son apparition : les ***f-string***.

Pour utiliser les ***f-string***, il suffit de rajouter la lettre ***f*** devant votre chaîne de caractères :

```
phrase = f"Je suis étudiant en ICT."
```

On peut ensuite, comme avec la méthode *format*, insérer des données dans la chaîne de caractères avec les accolades. La différence majeure réside dans le fait qu'on peut écrire du code Python directement dans les accolades. On retrouve donc la flexibilité de la méthode format mais avec quelque chose d'encore plus facile à lire.

```
phrase = f"Je suis étudiant en ICT."
```

Reprenons l'exemple avec les variable *age* et *prenom* :

```
prenom = "Pierre"
age = 26
langage = "Python"
print(f"Je m'appelle {prenom} et j'ai {age} ans. {prenom} apprends le langage
↪  {langage}")
# Je m'appelle Pierre et j'ai 26 ans. Pierre apprends le langage Python.
```

Je m'appelle Pierre et j'ai 26 ans. Pierre apprends le langage Python :

```
prenom = "Pierre"
age = 26
langage = "Python"
print(f"Je m'appelle {prenom.upper()} et j'ai {age} ans. {prenom} apprends le
↪  langage {langage.title()}")
# Je m'appelle Pierre et j'ai 26 ans. Pierre apprends le langage Python.
```

Les caractères spéciaux Vous l'avez peut-être vu venir... On utilise des accolades pour indiquer des emplacements à l'intérieur d'une chaîne de caractères. Comment faire alors pour insérer des accolades ? Facile, il suffit de les tripler !

```
prenom = "Pierre"
print(f"Je me nomme {{prenom}}") # Je me nomme {prenom}
print(f"Je me nomme {{{prenom}}}") # Je me nomme {Pierre}
```

Quelques options de formatage

Il est possible de spécifier des options de formatage avancées directement à l'intérieur des accolades.

Les options de formatages sont séparées des données par le symbole des deux-points :

```
f"{valeur:formatage}"
```

Voici une liste non exhaustive d'options de formatage que j'utilise très souvent.

'Padder' un nombre - Vous pouvez rajouter des 0 avant le nombre indiqué (ici 12) pour remplir la chaîne de caractères jusqu'à atteindre le nombre de caractères indiqués (ici 4) :

```
f"{12:04d}"  # '0012'
```

Ici, Python va donc rajouter 2 zéros par la gauche pour arriver à un total de 4 caractères (0012). Très pratique quand vous travaillez avec des séquences.

Formater une date Pour une liste plus complète des options de formatage disponibles pour les objets de type date, je vous conseille ce site très bien fait.

```
from datetime import datetime
print(f'datetime(2020, 8, 7, 9, 0):Le %Y-%m-%d à %H:%M}') # 'Le 2020-08-07 à
↪  09:00'
```

Tronquer une chaîne de caractères - Utile pour tous les endroits où vous avez un nombre limité de caractères (les descriptions de vidéos YouTube, articles de blogs etc).

```
phrase = """Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
↪  eiusmod tempor incididunt ut labore et dolore magna aliqua.
            Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
             ↪  nisi ut aliquip ex ea commodo consequat.
            Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
             ↪  dolore eu fugiat nulla pariatur.
            Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
             ↪  officia deserunt mollit anim id est laborum."""

print(f"{phrase:.60}") # 'Lorem ipsum dolor sit amet, consectetur adipiscing elit,
↪  sed'
```

A présent, ce chapitre est terminé. Ce fut long mais les bases sont très importantes à maîtriser. D'ailleurs nous n'en avons pas fini avec les bases de python (la suite se trouve dans le prochain chapitre du cours) mais nous allons nous arrêter ici et bosser sur les TP du chapitre.

2.3 TP

- Écrire un programme qui lit le nom et l'âge de l'utilisateur et affiche un message personnalisé.

```
age = input("Veuillez entrez votre âge svp : ")
nom = input("Veuillez entrez votre nom svp : ")
print(f"Bonjour Mrs/Mme, vous vous appelez {nom} et vous avez {age} ans.
↪  Merci.")
```

- Calculatrice scientifique basique : Dans ce projet, vous devez créer une calculatrice en ligne de commande qui demande à l'utilisateur de saisir deux nombres et qui affiche ensuite le résultat de l'addition de ces deux nombres.

```
a = int(input("Entrer le premier nombre :"))
b = int(input("Entrer le second nombre :"))

print(f"{a} + {b} = {a + b}")
```

2.4 Exercices

EXO 1 : Dans cet exercice, vous devez créer des objets de différents types natifs. Les valeurs des objets ne sont pas importantes, vous pouvez ainsi mettre n'importe quelle valeur dans les chaînes de caractères, nombres ou booléens, l'important est juste de respecter le type demandé.

```
# Créer un objet de type chaîne de caractères
chaine =
# Créer un objet de type nombre entier
nombre_entier =
# Créer objet de type nombre décimal
nombre_decimal =
# Créer un objet de type booléen
boolean =
```

EXO 2 : Dans cet exercice, vous devez modifier le code pour qu'il ne produise plus d'erreurs et qu'il affiche le texte suivant :

- Bonjour tout le monde !
- Je m'appelle Patrick.
- Je dis souvent "la vie est un long fleuve tranquille" quand on s'appelle Patrick.
- Oui, je suis très philosophe.

```
# Créer un objet de type chaîne de caractères
print('Bonjour tout le monde !")
print('Je m'appelle Patrick.')
print("Je dis souvent "la vie est un long fleuve tranquille" quand on s'appelle
↪  Patrick.")
print(Oui, je suis très philosophe.)
```

EXO 3 : Dans cet exercice, vous devez corriger les valeurs des variables a, b, c et d pour que le script ne produise plus d'erreurs.

- La variable a doit ainsi contenir le nombre -2,5.
- Les variables b et c respectivement les booléens vrai et faux.
- La variable d la chaîne de caractères Bonjour tout le monde.

```
a = -2..5
b = true
c = falce
d = Bonjour tout le monde
```

EXO 4 : Soit les chaînes de caractères suivantes :

```
h = 'Happy birthday'
t = 'to you'
```

```
p = 'Prenom'
```

Imprimer la chanson *Happy birthday* par concatenation de ces chaînes (aller à la ligne à la fin de chaque couplet).

EXO 5 : Écrire un script qui compte le nombre de caractères blancs " " contenus dans une chaîne entée par l'utilisateur. Par exemple, si la chaîne de caractères est s="Bonjour le monde !", on devra obtenir 3.

EXO 6 : Que vaut l'expression ci-dessous en python ?

```
len("lo"+("la"*5+" ")*4)
```

EXO 7 : Considérons les affectations :

```
a = "six"
b = a
c = " > "
d = "ty"
e = "1"
```

Modifier les variables en utilisant exclusivement les valeurs de a, b, c, d, e de sort à ce que l'expression a+c+e+b affiche sixty > 11 > six.

EXO 8 : Affecter la variable year avec l'année courante et birthyear avec l'année de votre naissance (on le demandera à l'utilisateur). Afficher la phrase "Je suis né en xxxx donc cette année j'aurai xx ans" où xx sera calculé automatiquement.

EXO 9 : Soit CT, CC, TP respectivement les notes de contrôle terminal, de contrôle continu et de travaux pratiques d'un ECUE. La note finale est calculée selon la formule :

$$0.3\text{TP} + \max\{0.7\text{CT}\,;\, 0.5\text{CT} + 0.2\text{CC}\}$$

Écrire un script qui calcule la note finale dans les cas suivants (vérifier les résultats !) :

- TP=10, CT=10, CC=10 ;
- TP=10, CT=10, CC=20 ;
- TP=10, CT=10, CC=20 ;
- TP=20, CT=10, CC=10 ;
- TP=20, CT=10, CC=20 ;
- TP=20, CT=20, CC=10.

EXO 10 : Pour tout nombre entier donné par l'utilisateur, écrire un script python pour calculer le nombre de chiffres qui le composent. Voici un exmple d'exécution de votre script :

```
Entrez un entier naturel : 35678
Nombre de chiffres : 5
```

EXO 11 : Affecter à la variable *secTOT* un nombre de secondes. Calculer le nombre d'heures, de minutes et de secondes correspondants. Exemple de output pour secTOT=12546 :

```
"12546 secondes correspondent à 3 h 29 m 6 s"
```

EXO 12 : Dans cet exercice, le but est de compter le nombre de phrases présentes dans le texte contenu dans la variable *lorem*.

```
lorem = """Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
↪  tempor incididunt ut labore et dolore magna aliqua.
                  Ut enim ad minim veniam, quis nostrud exercitation ullamco
                  ↪  laboris nisi ut aliquip ex ea commodo consequat.
                  Duis aute irure dolor in reprehenderit in voluptate velit esse
                  ↪  cillum dolore eu fugiat nulla pariatur.
                  Excepteur sint occaecat cupidatat non proident, sunt in culpa
                  ↪  qui officia deserunt mollit anim id est laborum."""

resultat = ?
```

EXO 13 : Le but de cet exercice et de remettre en ordre alphabétique les prénoms présents dans la chaîne de caractères.

```
chaine = "Pierre, Julien, Anne, Marie, Lucien"
```

Vous devez créer une variable *chaine_en_ordre* qui, à la fin de l'exercice, doit contenir la chaîne de caractères suivante :

```
Anne, Julien, Lucien, Marie, Pierre
```

EXO 14 : Dans cet exercice, vous devez recréer l'URL du site *https ://https ://www.eurosport.fr/football/* grâce aux différents variables

```
# Ne modifiez pas les variables ci-dessous
protocole = "https://"
nom_du_site = "eurosport"
extension = "fr"
page = "football"

# Modifiez le code à partir d'ici
```

```
URL = ?
```

EXO 15 : Avec constat = "Monsieur Tchos, vous dactylographiez bien mieux que votre vieil ami Pirlo", que donneront les commandes suivantes ?

```
s = "Monsieur Tchos, vous dactylographiez bien mieux que votre vieil ami Pirlo"

# Donnez le résultat de chaque commande ci-dessous:
len(s) ; s.count('e') ; s.count('ie')
s.find('i') ; s.find('i',40) ; s.find('i',20,30) ; s.rfind('i')
s.index('i') ; s.index('i',40) ; s.index('i',20,30) ; s.rindex('i')
"ab;.bc;.cd".replace(';.','-'); " abcABC ".strip();
```

Chapitre 3

Module 3 : Programmation procédurale et Structures de données

Objectifs :

— Structurer le code de manière procédurale.
— Manipuler les structures de données internes à Python.

Contenu :

- Les structures conditionnelles et les boucles.
- Les structures de données (Listes, tuples, sets, dictionnaires).
- Fonctions et portée des variables.
- Modules et packages.

Un programme consiste en une suite d'instructions structurées et exécutées par un ordinateur. Cette définition est le cœur de ce chapitre. Nous allons ici écrire nos premiers scripts Python en utilisant les bases que nous avons vues dans le chapitre précédant mais en modifiant le déroulement des opérations en introduisant les exécutions conditionnelles et en boucle avant d'affronter les structure de données et les fonctions.

3.1 Les structures conditionnelles

Les structures conditionnelles permettent d'ajouter de la logique dans l'exécution du code. Elles permettent d'exécuter une portion de code si une certaine condition est remplie, et une autre portion si cette même condition n'est pas remplie. Grâce à des conditions, vous pouvez contrôler votre code et exécuter une action A plutôt qu'une action B.

3.1.1 La structure conditionnelle if simple

En Python, on écrit une structure conditionnelle grâce au mot clé *if*.

```
if condition:
    code
    ...

suite du code
...
```

Python exécute le code qui se trouve à l'intérieur de votre structure conditionnelle uniquement si la **condition** est vraie (*True*). Si c'est *False*, le code n'est pas exécuté. En Python, *None* et *0* sont des valeurs interprétées comme *False*.

Autre chose, vous devez *ajouter deux points (:)* après la condition lors de la déclaration du *if*, sans quoi votre code ne fonctionnera pas. De même, vous devez **indenter** votre code à l'intérieur de votre structure conditionnelle pour qu'il soit correctement interprété par Python.

La première ligne indentée marque le début du bloc de code et cela s'arrête à la première ligne qui ne l'est plus.

Dans l'exemple ci-dessous, nous souhaitons vérifier qu'un utilisateur est majeur.

```
age = 24
MAJORITE = 21

if age >= MAJORITE:
    print("Je suis majeur !")

print(f"J'ai {age} ans !")
```

Je définis une variable *age* qui correspond à l'âge de l'utilisateur fictif puis une constante *MAJORITE* qui est assignée à la valeur *21*, l'âge légal de la majorité au Cameroun.

Je démarre ma structure conditionnelle avec le mot-clé *if* suivi de la condition *age >= MAJORITE* et de *deux-points :*

Si la variable *age* est égale à 21 ou plus, j'indente mon code et j'affiche le message « *Je suis majeur !* » . Dans tous les cas, le programme se termine par l'affichage de l'âge de l'utilisateur.

3.1.2 La structure conditionnelle if / else

Dans l'exemple précédent, notre code faisait quelque chose de particulier uniquement si l'utilisateur était majeur. Oui, mais qu'en est-il s'il est mineur ?

Le mot clé *else* permet d'exécuter une condition si les conditions précédentes ne sont pas vérifiées. On doit l'écrire au même niveau d'indentation que le *if*, finir par d*deux-points :* puis indenter notre code.

```
age = 15
MAJORITE = 21

if age >= MAJORITE:
```

```
    print("Je suis majeur !")
else:
    print("Je suis mineur !")

print(f"J'ai {age} ans !")
```

3.1.3 La structure conditionnelle if / elif / else

Le mot-clé *elif* (pour « **else if** », en français « **sinon si** ») permet d'ajouter des conditions supplémentaires à une structure conditionnelle. On peut ajouter autant de **elif** que nécessaire.

Si la condition testée dans le *if* est *False* alors on testera celles du/des *elif*. Si toutes ces conditions retournent *False* alors on exécutera le code se trouvant dans le *else*.

```
age = 15
MAJORITE = 21

if age > MAJORITE:
    print("Je suis majeur !")
elif age == MAJORITE:
    print("Tout juste majeur depuis aujourd'hui")
else:
    print("Je suis mineur..")

print(f"J'ai {age} ans !")
```

3.1.4 Imbriquer des structures conditionnelles

Vous pouvez imbriquer une structure conditionnelle dans une autre structure, puis dans une autre, et encore une autre, etc...On appelle ça de l'**imbrication** en français (« **nesting** » en anglais).

Il suffit de déclarer votre nouvelle structure à l'intérieur de la précédente avec une indentation.

Vous imbriquez ainsi, grâce à l'indentation, plusieurs structures les unes dans les autres, comme des poupées russes :

```
age = 15
MAJORITE = 21

if age >= MAJORITE:
    print("Je suis majeur !")
    if age == MAJORITE:
        print("Tout juste majeur depuis aujourd'hui")
else:
    print("Je suis mineur !")

print(f"J'ai {age} ans !")
```

Il est conseillé de ne pas trop abuser de cette imbrication pour que votre code reste lisible et facile à suivre.

3.1.5 Syntaxe compacte d'une alternative

L'opérateur ternaire en Python est une expression conditionnelle qui permet d'écrire des instructions *if-else* simples sur une seule ligne pour affecter une valeur à une variable.

Syntaxe

Sa syntaxe est *valeur_si_vrai si condition sinon valeur_si_faux*. Il est utile pour des conditions concises afin d'améliorer la lisibilité du code.

La structure est la suivante :

Syntaxe :

variable = valeur_si_vrai if condition else valeur_si_faux

Voici un exemple concret pour illustrer son utilisation :

```
age = 20
statut = "Majeur" if age >= 21 else "Mineur"
print(statut)
```

Dans cet exemple, si age est supérieur ou égal à **21**, la variable statut sera définie comme "*Majeur*". Sinon, elle sera définie comme "*Mineur*".

Avantages de l'opérateur ternaire

- **Compacité** : Il permet de condenser une logique if-else simple en une seule ligne, ce qui rend le code plus court.
- **Lisibilité** : Pour des conditions simples, il peut rendre l'intention du code plus claire.

Quand l'utiliser et quand l'éviter

- **Utilisation conseillée** : Il est idéal pour des affectations de variables basées sur une condition unique et simple.
- **Limitation** : Pour des conditions complexes, des blocs de code multiples ou plusieurs *elif*, il est préférable d'utiliser une instruction *if-else* classique pour une meilleure lisibilité.

3.2 Les boucles

La boucle est une structure de programmation ou une fonction qui répète une instruction ou une séquence d'instructions jusqu'à ce qu'une condition donnée se réalise. Une boucle permet ainsi d'indiquer à l'ordinateur, suivant le langage de programmation utilisé (Python notamment), d'exécuter les mêmes algorithmes ou les mêmes instructions jusqu'à ce que la condition d'arrêt de la boucle soit atteinte. Dans le code, la boucle pose d'abord une condition d'entrée, spécifie ensuite l'instruction itérative et pose enfin la condition de sortie pour mettre fin à l'itération.

L'algorithme comprend également une variable compteur sur laquelle se basera l'itération. Quand la boucle est lancée, l'ordinateur vérifie ainsi, avant chaque réalisation d'instruction, que la fonction itérative est toujours valide compte tenu des données d'entrée. Si oui, l'action adéquate indiquée par l'algorithme est exécutée, sinon la boucle prend fin. Pour traiter un même problème de logique informatique, utiliser une boucle ou plusieurs boucles sur Python génère des algorithmes avec le moins de lignes de code possible, qui restent tout aussi efficaces que des algorithmes plus longs.

Python propose deux instructions particulières pour construire des boucles : l'instruction *for ... in ...* et l'instruction *while* que nous allons découvrir tout de suite.

3.2.1 La boucle while

La boucle *while* est utilisée pour exécuter du code tant qu'une **certaine condition est vérifiée**. Cette boucle est très utile lorsqu'on ne sait pas combien de fois nous devons itérer.

Syntaxe de la boucle while

Une boucle *while* s'écrit de cette manière :

```
while condition:
    code
    ....
```

Lorsque votre script rencontre une boucle *while*, il vérifie que la condition renvoie *True*. Tant que cette condition retourne *True*, il exécute le code contenu à l'intérieur de la boucle sans interruption ! À chaque itération, il vérifie la condition et ne sort de la boucle que si elle renvoie *False*.

Par exemple :

```
i = 0
while i < 10:
    print('Salut')
    i += 1
```

Ici, tant que i est inférieur à 10, on affiche '*Salut*'.

Il faut noter **deux choses très importantes** :

- **On déclare la variable qui nous sert de condition avant la boucle while** ⇒ On assigne la valeur O à la variable i juste avant d'entrer dans la boucle.

- **On met à jour notre condition à chaque nouvelle itération** ⇒ On incrémente i à la fin de ma boucle. Ce faisant, on évite de rentrer dans une **boucle infinie**. Eh oui, d'après vous, que se passe-t-il si je n'incrémente pas i à la fin de la boucle ? Mon script part en boucle infinie et mon programme ne se terminera jamais à moins que je l'arrête manuellement ! Faites-y très attention !

while...else

Comme on le verra avec les boucles *for*, il est possible de définir un comportement par défaut grâce au mot-clé *else*. Le code à l'intérieur du *else* sera exécuté si la condition de votre boucle *while* renvoie *False*.

```
i = 0
while i < 3:
    print('Salut')
    i += 1
else:
    print('Au revoir')
```

Dans l'exemple ci-dessus, on affiche '*Salut*' tant que $i < 3$ ('*Salut*' sera donc affiché 3 fois). À la 4e itération, i est supérieur à 3, on passe donc dans le *else* et on affiche '*Au revoir*'.

3.2.2 La boucle for

Avant toute chose, il est nécessaire de vous présenter 2 notions très utilisées en python : les **conteneurs** et les **itérables**.

Définition :

Un **conteneur** désigne un type de données permettant de stocker un ensemble d'autres données, en ayant ou non, suivant le type du conteneur, une notion de classement ordonné entre ces données.

Définition :

Un **itérable** désigne un type de conteneur que l'on peut parcourir élément par élément ou qui est capable de fournir des séquences de valeurs à la demande.

Avec Python, on utilise la boucle *for* pour parcourir des objets itérables comme les listes, les tuples ou encore les chaînes de caractères. C'est une boucle commune à presque tous les langages de programmation.

Parcourir ces *itérables et conteneurs* avec la boucle *for* vous permettra d'accéder tour à tour à chaque valeur afin de la traiter dans le corps de la boucle.

Syntaxe de la boucle for

Une boucle *for* s'écrit de la manière suivante :

```
for value in object:
    code
    ....
```

La variable *value* prend la valeur de chaque élément contenu dans *object* à chaque itération. La boucle *for* continue jusqu'à ce qu'on ait atteint le dernier élément de *object*.

En pratique, cela donne :

```
fruits = ['pomme', 'orange', 'ananas', 'pastèque', 'cerise', 'banane', 'kiwi']

for fruit in fruits:
    print(fruit)
```

Vous pouvez tester et voir le résultat par vous-mêmes.

Comme pour les structures conditionnelles *if..else*, vous pouvez définir un comportement par défaut pour votre boucle for grâce au mot-clé *else*.

for ... else

Ce mot clé est optionnel, mais cela vous permet d'exécuter du code lorsque la boucle *for* est terminée :

```
fruits = ['pomme', 'orange', 'ananas', 'pastèque', 'cerise', 'banane', 'kiwi']

for fruit in fruits:
    print(fruit)
else:
    print('Plus de fruits dans le panier')
```

Voici d'autres exemples d'utilisation de la boucle *for* avec différents types d'itérables :

```
for i in range(3):
    print(i)

prenom = "John"
for lettre in prenom:
    print(lettre)

for (pk, name) in [(1, "Patrick"), (2, "John"), (3, "Marie")]:
    print(pk, name)
```

3.2.3 Les instructions break, continue et pass

L'utilisation des boucles *for* et *while* en Python vous permet d'automatiser et de répéter des tâches de manière efficace.

Mais parfois, un facteur externe peut influencer le fonctionnement de votre programme. Lorsque cela se produit, vous pouvez souhaiter que votre programme quitte complètement une boucle, saute une partie d'une boucle avant de continuer ou ignore ce facteur externe. Vous pouvez effectuer ces actions avec les instructions *break*, *continue*, et *pass*.

L'instruction break

En Python, l'instruction *break* vous offre la possibilité de sortir d'une boucle lorsqu'une condition externe est déclenchée. Vous placerez l'instruction *break* dans le bloc de code sous votre instruction de boucle, généralement après une instruction conditionnelle *if*.

```
nombre = 0
for nombre in range(10):
    if nombre == 5:
        break # break ici
    print(f'Le Nombre est {nombre}')
print('En dehors de la boucle')
```

Dans ce petit programme, la variable nombre est initialisée à *0*. Ensuite, une instruction *for* construit la boucle tant que la variable nombre est inférieure à *10*.

Au sein de la boucle *for*, il y a une déclaration if qui présente la condition que si la variable nombre est équivalente à l'entier *5*, la boucle s'interrompra alors grâce à l'instruction *break*.

Dans la boucle se trouve également une instruction *print()* qui s'exécutera à chaque itération de la boucle *for*. Elle se situe après l'instruction *break* et nous permettra d'illustrer le comportement de l'exemple. Pour savoir quand nous sommes hors de la boucle, nous avons inclus une instruction *print()* finale en dehors de la boucle *for*.

Lorsque nous exécutons ce code, nous obtenons la sortie suivante :

```
Le Nombre est 0
Le Nombre est 1
Le Nombre est 2
Le Nombre est 3
Le Nombre est 4
En dehors de la boucle
```

Cela montre qu'une fois que l'entier nombre est évalué comme équivalent à *5*, la boucle se rompt, car le programme est invité à le faire avec l'instruction *break*.

l'instruction *break* fait sortir un programme d'une boucle. Dans le cas où vous n'avez plus besoin d'itérer sur les éléments restant, sortir de la boucle à l'aide de *break* vous permettra d'optimiser votre programme en vous faisant gagner du temps, car la boucle n'aura pas à parcourir les éléments restants.

L'instruction continue

L'instruction *continue* vous donne la possibilité de sauter l'exécution d'une partie du code d'une boucle quand une **condition externe est déclenchée**, puis de passer à l'itération suivante pour terminer le reste de la boucle. C'est-à-dire que l'itération actuelle de la boucle sera **interrompue** et le programme reviendra au sommet de la boucle pour passer à l'itération suivante.

L'instruction *continue* sera dans le bloc de code sous l'instruction de boucle, généralement après une instruction conditionnelle *if*.

En utilisant la même boucle for que dans la section précédente, nous utiliserons une *continue* plutôt qu'une instruction *break* :

```
nombre = 0
for nombre in range(10):
    if nombre == 5:
        continue # continue ici
    print(f'Le Nombre est {nombre}')
print('En dehors de la boucle')
```

La différence entre l'utilisation de l'instruction *continue* plutôt qu'une instruction *break* est que notre code continuera malgré la perturbation lorsque la variable nombre est évaluée comme équivalente à *5*. Passons en revue notre sortie :

```
Le Nombre est 0
Le Nombre est 1
Le Nombre est 2
Le Nombre est 3
Le Nombre est 4
Le Nombre est 6
Le Nombre est 7
Le Nombre est 8
Le Nombre est 9
En dehors de la boucle
```

Vous remarquerez ici que le nombre *5* n'apparaît pas dans la sortie, mais que la boucle a quand même continué après ce point en affichant des lignes pour les nombres *6* à *10* avant de quitter la boucle. Vous pouvez utiliser l'instruction *continue* pour éviter un code conditionnel profondément imbriqué ou pour optimiser une boucle en éliminant les cas fréquents que vous voudriez rejeter.

L'instruction *continue* fait qu'un programme ignore certains facteurs qui apparaissent dans une boucle, mais continue ensuite à travers le reste de la boucle.

L'instruction pass

Lorsqu'une condition externe est déclenchée, l'instruction *pass* vous permet de gérer la condition sans que la boucle soit affectée de quelque manière que ce soit ; tout le code continuera à être lu à moins qu'une *break* ou une autre instruction ne se produise. Comme pour les autres instructions, l'instruction *pass* sera dans le bloc de code sous l'instruction de boucle, généralement après une instruction conditionnelle *if*.

En utilisant toujours le même bloc de code, remplaçons l'instruction *break* ou *continue* par une instruction *pass* :

```
nombre = 0
for nombre in range(10):
    if nombre == 5:
        pass # pass ici
    print(f'Le Nombre est {nombre}')
print('En dehors de la boucle')
```

L'instruction *pass* qui se produit après l'instruction *if* conditionnelle indique au programme de continuer à exécuter la boucle et d'ignorer le fait que la variable nombre est réévaluée comme équivalente à *5* au cours de l'une de ses itérations.

Nous allons exécuter le programme et considérer le résultat :

```
Le Nombre est 0
Le Nombre est 1
Le Nombre est 2
Le Nombre est 3
Le Nombre est 4
Le Nombre est 5
Le Nombre est 6
Le Nombre est 7
Le Nombre est 8
Le Nombre est 9
En dehors de la boucle
```

En utilisant l'instruction *pass* dans ce programme, nous remarquons que le programme s'exécute exactement comme il le ferait s'il n'y avait pas d'instruction conditionnelle dans le programme. L'instruction *pass* indique au programme de ne pas tenir compte de cette condition et de continuer à exécuter le programme comme d'habitude.

L'instruction *pass* peut créer des classes minimales ou agir comme un espace réservé lorsque vous travaillez sur un nouveau code et réfléchissez à un niveau algorithmique avant de définir des détails.

Il n'y a pas grand-chose à rajouter à part que les instructions *break*, *continue* et *pass* vous permettront d'utiliser des boucles for et des boucles while plus efficacement dans votre code.

3.3 Les structures de données

Les structures de données sont des moyens spécifiques d'organiser et de stocker des données afin qu'elles puissent être consultées et travaillées de manière efficace. Les structures de données définissent la relation entre les données et les opérations pouvant être effectuées dessus.

Le choix d'une structure de données dépend du problème que nous résolvons et du type de données dont nous disposons. Une structure de données inappropriée peut entraîner des durées de fonctionnement (inutilement) longues, des pertes de mémoire et un gaspillage de stockage.

Les structures de données principales en Python sont les listes, les tuples, les dictionnaires, les chaînes de caractères et les ensembles. Les listes, tuples et chaînes de caractères sont des structures

de **type séquence**, tandis que les dictionnaires sont des structures de **type correspondance**. Les ensembles ne **contiennent pas de doublons** et sont souvent utilisés pour des opérations mathématiques.

3.3.1 Les listes

Une liste est simplement une structure de données muable et ordonnée dans laquelle vous pouvez stocker n'importe quel type d'objet. Chaque objet contenu dans une liste est appelé un élément.

Les listes en Python sont des collections ordonnées et modifiables d'éléments, créées en plaçant des valeurs séparées par des virgules entre crochets []. Elles peuvent contenir des éléments de types de données différents et sont un type de donnée mutable, ce qui signifie qu'on peut les modifier après leur création en ajoutant, supprimant ou changeant des éléments. On accède aux éléments via leur indice, qui commence à 0.

Enfin, et c'est vraiment le plus intéressant, une liste est une séquence de données ! Cela signifie que tu peux itérer sur cette séquence, avec une boucle for par exemple.

Pour créer une liste, utilisez des crochets [] et séparez les éléments par des virgules.

Caractéristiques principales des listes

Voici les caractéristiques principales des listes :

- **Ordonnées** : Les éléments conservent leur ordre d'insertion.
- **Modifiables (mutables)** : Vous pouvez changer les éléments après la création de la liste.
- **Flexibles** : Une liste peut contenir des éléments de types différents et des doublons.
- **Itérables** : Vous pouvez à l'aide d'une boucle *for* parcourir le contenu d'une liste.

Différents éléments possibles

En Python, les listes sont simples à manipuler et intuitives, ce qui en fait un choix privilégié pour les développeurs de tous niveaux. Voici une liste non-exhaustive des opérations que vous pouvez effectuer avec une liste

Créer une liste vide - Pour créer une liste sans élément, utilisez simplement des crochets vides *[]* :

```
# Créer une liste vide
ma_liste = []
```

Cela initialise une liste vide, prête à être remplie. Une liste vide peut être utile dans de nombreux contextes, par exemple lorsque l'on souhaite collecter des éléments au fur et à mesure dans une boucle.

Créer une liste avec des valeurs initiales - Vous pouvez également initialiser une liste avec des valeurs prédéfinies :

```
ma_liste = [10, 20, 30, 40, 50]
```

Dans cet exemple, *ma_liste* contient cinq éléments de type entier. Cette méthode est idéale pour les cas où vous connaissez déjà les éléments à ajouter.

Créer une liste avec des éléments de types différents - Une liste en Python peut contenir des éléments de types variés, ce qui permet d'imbriquer des données de différents types dans une même collection :

```
ma_liste = [1, "texte", True, 3.14]
```

Ici, la liste contient un entier, une chaîne de caractères, un booléen et un nombre à virgule flottante. Cette flexibilité permet de regrouper des données diverses dans une seule variable.

Générer automatiquement des valeurs avec range() - Si vous avez besoin d'une séquence de nombres, vous pouvez utiliser la fonction range(), souvent en combinaison avec list(). Par exemple, pour créer une liste de nombres de 0 à 9 :

```
ma_liste = list(range(10))
```

La fonction *range()* génère une séquence de nombres que *list()* convertit en liste.

Accéder aux éléments d'une liste - Une fois qu'une liste est créée, vous devez savoir comment accéder à ses éléments pour les utiliser ou les modifier. Les listes Python utilisent l'**indexation** pour accéder à chaque élément.

Indexation de base Les listes sont des **séquences indexées** en Python, ce qui signifie que chaque élément possède une position dans la liste, appelée index. L'indexation commence à *0*, donc le premier élément d'une liste est situé à l'index *0*, le deuxième à l'index *1* et ainsi de suite.

```
ma_liste = [10, 20, 30, 40, 50]
print(ma_liste[0])  # Affiche 10
print(ma_liste[2])  # Affiche 30
```

Utilisation de l'indexation négative - Python permet aussi d'accéder aux éléments d'une liste en utilisant des **indices négatifs**, ce qui est particulièrement utile pour accéder aux éléments depuis la fin de la liste. L'index *-1* correspond au dernier élément, *-2* à l'avant-dernier et ainsi de suite.

```
print(ma_liste[-1])  # Affiche 50
print(ma_liste[-2])  # Affiche 40
```

Extraire une sous-liste avec le slicing Le **slicing** permet de sélectionner une partie de la liste (une sous-liste). Cette technique est particulièrement utile pour obtenir une série d'éléments situés à des **indices consécutifs**.

```
# liste[début:fin]
sous_liste = ma_liste[1:4]
print(sous_liste)  # Affiche [20, 30, 40]
```

Utilisation du pas (step) dans le slicing - En ajoutant un troisième argument au slicing, vous pouvez spécifier un pas pour sauter des éléments.

```
# liste[début:fin:pas]
sous_liste = ma_liste[0:5:2]
print(sous_liste)  # Affiche [10, 30, 50]
```

Avec le **slicing** ma_liste[0 :5 :2], on sélectionne les éléments de la liste en sautant un élément sur deux.

Ajouter et supprimer des éléments - Python offre différentes méthodes pour **ajouter** ou **supprimer** des éléments d'une liste, rendant la modification de listes très flexible.

Ajouter un élément avec append() - La méthode **append()** ajoute un élément à la fin de la liste :

```
ma_liste.append(60)
print(ma_liste)  # Affiche [10, 20, 30, 40, 50, 60]
```

Insérer un élément à une position spécifique avec insert() - Pour insérer un élément à une position précise, utilisez **insert(index, element)** :

```
ma_liste.insert(1, 15)
print(ma_liste)  # Affiche [10, 15, 20, 30, 40, 50, 60]
```

Ajouter plusieurs éléments avec extend() : Pour ajouter plusieurs éléments en une fois, *extend()* est idéal :

```
ma_liste.extend([70, 80])
print(ma_liste)  # Affiche [10, 15, 20, 30, 40, 50, 60, 70, 80]
```

Supprimer un élément avec remove() - La méthode *remove()* supprime la **première occurrence d'un élément** :

```
ma_liste.remove(15)
print(ma_liste)  # Affiche [10, 20, 30, 40, 50, 60, 70, 80]
```

Supprimer un élément avec pop() - *pop()* supprime un élément à une position donnée et le retourne. Si aucun index n'est précisé, il retire et retourne le dernier élément :

```
dernier_element = ma_liste.pop()
print(dernier_element)  # Affiche 80
print(ma_liste)         # Affiche [10, 20, 30, 40, 50, 60, 70]
```

Supprimer un élément avec del - L'instruction *del* permet de supprimer un élément par son index, ou même un groupe d'éléments avec le **slicing** :

```
del ma_liste[0]
print(ma_liste)  # Affiche [20, 30, 40, 50, 60, 70]

del ma_liste[1:3]
print(ma_liste)  # Affiche [20, 50, 60, 70]
```

Modifier les éléments d'une liste - Les listes en Python sont **modifiables** (ou **mutables**), ce qui signifie que vous pouvez changer la valeur des éléments existants.

Modifier un élément spécifique - Pour modifier un élément, accédez à sa position avec son index et assignez-lui une nouvelle valeur :

```
ma_liste[1] = 45
print(ma_liste)  # Affiche [20, 45, 60, 70]
```

Modifier plusieurs éléments avec le slicing - Vous pouvez utiliser le slicing pour modifier plusieurs éléments à la fois :

```
# liste[début:fin]
ma_liste[1:3] = [35, 55]
print(ma_liste)  # Affiche [20, 35, 55, 70]
```

Ajouter ou supprimer des éléments avec le slicing - Le slicing permet également d'ajouter ou de supprimer des éléments en une opération :

```
ma_liste[2:2] = [40, 50]
print(ma_liste)  # Affiche [20, 35, 40, 50, 55, 70]
```

Ici, *40* et *50* sont insérés à l'index *2*.

Fonctions intégrées utiles pour les listes - Python propose de nombreuses fonctions intégrées pour manipuler les listes de manière efficace. Ces fonctions permettent d'obtenir des informations sur une liste ou d'effectuer des opérations courantes comme le **comptage**, le **tri** ou l'**inversion** des éléments.

Longueur de la liste avec len() - La fonction *len()* est utilisée pour obtenir le nombre d'éléments présents dans une liste, ce qui peut être pratique dans des boucles ou pour vérifier si une liste est vide :

```
ma_liste = [10, 20, 30, 40, 50]
print(len(ma_liste))  # Affiche 5 car la liste contient cinq éléments.
```

Trouver la position d'un élément avec index() - La méthode *index()* permet de trouver la **première position d'un élément spécifique** dans une liste. Si l'élément n'est pas trouvé, Python génère une erreur *ValueError* :

```
position = ma_liste.index(30)
print(position)  # Affiche 2 car l'élément 30 se trouve à l'index 2 dans la liste.
```

Vérifier la présence d'un élément avec in - Vous pouvez utiliser l'opérateur ***in*** pour vérifier si un élément existe dans une liste. Cela renvoie *True* si l'élément est trouvé et *False* sinon :

```
print(20 in ma_liste)  # Affiche True
print(60 in ma_liste)  # Affiche False
```

Compter les occurrences d'un élément avec count() - La méthode ***count()*** permet de **compter le nombre d'occurrences** d'un élément spécifique dans la liste. Cela peut être utile pour analyser des données ou trouver les doublons :

```
ma_liste = [10, 20, 20, 30, 20]
print(ma_liste.count(20))  # Affiche 3 car 20 apparaît trois fois dans ma_liste.
```

Trier une liste avec sort() - La méthode ***sort()*** trie les éléments d'une liste dans l'ordre croissant par défaut, mais elle peut également trier en ordre décroissant en utilisant l'argument ***reverse=True*** :

```
ma_liste = [30, 10, 20, 50, 40]
ma_liste.sort()
print(ma_liste)  # Affiche [10, 20, 30, 40, 50]

ma_liste.sort(reverse=True)
print(ma_liste)  # Affiche [50, 40, 30, 20, 10]
```

Le tri est fait en place, ce qui signifie que la liste d'origine est modifiée. Si vous souhaitez conserver la liste initiale, utilisez la fonction ***sorted()*** pour créer une nouvelle liste triée.

Inverser une liste avec reverse() - La méthode ***reverse()*** permet d'inverser l'ordre des éléments dans une liste. Contrairement à ***sort()***, cette méthode ne trie pas mais retourne simplement les éléments dans l'ordre opposé :

```
ma_liste.reverse()
print(ma_liste)  # Affiche [10, 20, 30, 40, 50] dans l'ordre inverse
```

Boucles et itérations sur une liste - Parcourir les éléments d'une liste est une opération fréquente en Python, notamment pour appliquer des opérations à chaque élément. Cette section vous montre comment utiliser les boucles pour **itérer** sur les éléments d'une liste.

Parcourir une liste avec une boucle for - La manière la plus simple et la plus courante de parcourir une liste est d'utiliser une boucle ***for***. Elle vous permet d'accéder à chaque élément de la liste :

```
ma_liste = [10, 20, 30, 40, 50]
for element in ma_liste:
    print(element)
```

Dans cet exemple, chaque élément de ***ma_liste*** est affiché un par un. C'est la méthode privilégiée pour des opérations simples.

Utiliser enumerate() pour obtenir les index et les éléments - Parfois, vous avez besoin de l'index de chaque élément en plus de sa valeur. Dans ce cas, utilisez la fonction ***enumerate()*** qui vous permet de récupérer à la fois l'**index** et l'**élément** :

```
for index, element in enumerate(ma_liste):
    print(f"Index {index} : {element}")
```

enumerate() renvoie des couples ***index***, ***valeur*** pour chaque élément de la liste, ce qui est pratique dans les cas où la position de l'élément est importante pour le traitement.

Parcourir une liste avec une boucle while - Une autre méthode pour parcourir une liste est d'utiliser une boucle **while**. Elle peut être utile si vous voulez parcourir la liste jusqu'à ce qu'une condition spécifique soit remplie :

```
index = 0
while index < len(ma_liste):
    print(ma_liste[index])
    index += 1
```

Ici, nous utilisons un compteur ***index*** pour accéder aux éléments de ***ma_liste*** jusqu'à ce que tous les éléments aient été parcourus.

Listes imbriquées (listes de listes) - Les listes imbriquées sont des listes qui contiennent d'autres listes comme éléments. Elles permettent de créer des structures de données plus complexes,

comme des matrices ou des tableaux multidimensionnels.

Créer une liste imbriquée - Pour créer une liste imbriquée, il suffit d'inclure des sous-listes au sein d'une liste principale :

```
ma_liste = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Ici, ma_liste est une liste de trois sous-listes, chacune contenant trois éléments.

Accéder aux éléments d'une liste imbriquée - Pour accéder aux éléments d'une liste imbriquée, vous utiliserez deux niveaux d'indexation. Le premier index sélectionne la sous-liste, le second sélectionne l'élément dans cette sous-liste.

```
print(ma_liste[0])      # Affiche [1, 2, 3]
print(ma_liste[0][1])   # Affiche 2
```

Dans cet exemple, ***ma_liste[0]*** renvoie la première sous-liste ***[1, 2, 3]***, et ***ma_liste[0][1]*** renvoie le deuxième élément de cette sous-liste, soit *2*.

Parcourir une liste imbriquée avec des boucles Pour parcourir tous les éléments d'une liste imbriquée, utilisez des **boucles imbriquées** : une première boucle pour parcourir les sous-listes et une seconde pour parcourir les éléments de chaque sous-liste.

```
for sous_liste in ma_liste:
    for element in sous_liste:
        print(element)
```

Chaque ***sous_liste*** est parcourue dans la boucle externe et chaque ***element*** dans la boucle interne.

Utiliser des compréhensions de liste avec des listes imbriquées - Les **compréhensions de liste** peuvent être imbriquées pour manipuler des listes imbriquées de manière concise. Par exemple, pour aplatir une liste imbriquée en une liste simple :

```
ma_liste_imbriquee = [[1, 2, 3], [4, 5], [6, 7, 8]]
aplatie = [element for sous_liste in ma_liste_imbriquee for element in sous_liste]
print(aplatie)  # Affiche [1, 2, 3, 4, 5, 6, 7, 8]
```

Dans cet exemple, chaque **element** est extrait de chaque ***sous_liste*** pour créer une liste aplanie, ***aplatie***.

Les compréhensions de liste - Les **compréhensions de liste** permettent de créer de nouvelles listes de manière concise et lisible. Elles sont particulièrement utiles pour effectuer des transformations ou des filtrages rapides.

Syntaxe de base des compréhensions de liste - La syntaxe de base d'une compréhension de liste consiste en une ***expression*** suivie d'une boucle ***for***. Par exemple, pour créer une liste des carrés des éléments d'une liste existante :

```
ma_liste = [1, 2, 3, 4, 5]
carres = [x ** 2 for

 x in ma_liste]
print(carres)  # Affiche [1, 4, 9, 16, 25]
```

Utiliser des conditions dans les compréhensions de liste - Les compréhensions de liste peuvent inclure des **conditions** pour filtrer les éléments :

```
pairs = [x for x in ma_liste if x % 2 == 0]
print(pairs)  # Affiche [2, 4]
```

Ici, seuls les éléments pairs de ***ma_liste*** sont inclus dans la nouvelle liste ***pairs***.

Compréhensions de liste imbriquées - Les compréhensions de liste imbriquées permettent de manipuler les listes imbriquées de manière concise. Par exemple, pour obtenir les carrés des éléments d'une liste imbriquée :

```
ma_liste_imbriquee = [[1, 2], [3, 4]]
carres = [[x ** 2 for x in sous_liste] for sous_liste in ma_liste_imbriquee]
print(carres)  # Affiche [[1, 4], [9, 16]]
```

Chaque sous-liste est parcourue pour calculer les carrés des éléments.

Méthodes avancées sur les listes - Enfin, Python propose des méthodes avancées pour copier, fusionner et transformer les listes de manière efficace, notamment avec ***map()*** et ***filter()***.

Copier une liste - Pour créer une copie d'une liste, vous pouvez utiliser ***copy()*** ou le ***slicing*** *[:]*. Cela évite de modifier la liste d'origine par inadvertance :

```
ma_liste = [1, 2, 3]
copie = ma_liste.copy()
```

Fusionner des listes Pour fusionner deux listes, utilisez l'opérateur *+* ou la méthode ***extend()*** :

```
fusion = liste1 + liste2
```

Utiliser map() et filter() - ***map()*** applique une fonction à chaque élément d'une liste, tandis que ***filter()*** sélectionne les éléments répondant à une condition :

```
ma_liste = [1, 2, 3]
carres = list(map(lambda x: x ** 2, ma_liste))
pairs = list(filter(lambda x: x % 2 == 0, ma_liste))
```

Bonnes pratiques pour travailler avec les listes

Travailler avec des listes en Python de manière efficace implique de suivre certaines **bonnes pratiques** pour garantir un code optimisé, lisible et évitant les erreurs courantes. Ces pratiques vous aident à écrire un code plus sûr et plus performant.

Privilégiez les compréhensions de liste pour les opérations simples - Les **compréhensions de liste** sont non seulement plus concises, mais souvent plus rapides que les boucles traditionnelles pour créer ou transformer des listes. Par exemple, pour obtenir une liste de carrés :

```
carres = [x ** 2 for x in range(10)]
```

Cette syntaxe est plus claire et plus efficace que de créer une liste avec une boucle ***for***.

Évitez de modifier une liste pendant son parcours - La modification d'une liste pendant son parcours peut causer des comportements inattendus, en particulier lors de la suppression d'éléments. Si vous avez besoin de transformer une liste, il est souvent préférable de créer une nouvelle liste plutôt que de modifier la liste d'origine :

```
# Mauvaise pratique
for element in ma_liste:
    if element < 10:
        ma_liste.remove(element)

# Bonne pratique : utiliser une compréhension de liste pour filtrer
nouvelle_liste = [element for element in ma_liste if element >= 10]
```

Utilisez les copies de liste pour préserver l'intégrité de la liste d'origine - Si vous devez conserver la liste d'origine intacte tout en la modifiant temporairement, créez une copie à l'aide de ***copy()*** ou du ***slicing [:]*** :

```
ma_liste = [1, 2, 3]
copie = ma_liste[:]  # Ou bien ma_liste.copy()
copie.append(4)
print(ma_liste)  # Affiche [1, 2, 3]
print(copie)     # Affiche [1, 2, 3, 4]
```

Cette pratique est particulièrement importante lorsque vous passez des listes à des fonctions, car Python passe les listes par référence, ce qui peut entraîner des modifications inattendues de la liste originale.

Attention aux listes imbriquées et aux copies profondes - Si vous travaillez avec des listes imbriquées, il est important de savoir que ***copy()*** ou le ***slicing [:]*** ne créent qu'une **copie peu profonde (shallow copy)**. Cela signifie que les sous-listes ne sont pas copiées indépendamment. Pour créer une **copie profonde (deep copy)** d'une liste imbriquée, utilisez le module texacopy et sa fonction deepcopy() :

```
import copy

ma_liste_imbriquee = [[1, 2], [3, 4]]
copie_profonde = copy.deepcopy(ma_liste_imbriquee)
copie_profonde[0][0] = 9
print(ma_liste_imbriquee)  # Affiche [[1, 2], [3, 4]]
print(copie_profonde)      # Affiche [[9, 2], [3, 4]]
```

La **copie profonde** crée des duplicatas de toutes les sous-listes, préservant ainsi la liste d'origine.

Optimisez les opérations coûteuses - Certaines opérations comme le **tri (sort())** ou l'**inversion (reverse())** modifient la liste en place. Si vous souhaitez obtenir une version triée ou inversée sans modifier la liste d'origine, utilisez les fonctions sorted() et reversed() :

```
ma_liste = [3, 1, 4, 2]
tri_inverse = sorted(ma_liste, reverse=True)
print(ma_liste)       # Affiche [3, 1, 4, 2]
print(tri_inverse)    # Affiche [4, 3, 2, 1]
```

En utilisant sorted() et reversed(), vous pouvez obtenir une nouvelle version triée ou inversée sans toucher à la liste originale.

Limitez les conversions entre listes et autres types de données - Python vous permet de convertir des listes en d'autres types, comme des tuples ou des ensembles, mais chaque conversion utilise des ressources. Limitez les conversions inutiles et assurez-vous que la conversion est vraiment nécessaire pour optimiser les performances de votre code :

```
ma_liste = [1, 2, 2, 3]
ensemble = set(ma_liste)  # Conversion pour supprimer les doublons
print(ensemble)  # Affiche {1, 2, 3}
```

Dans cet exemple, la conversion est justifiée pour supprimer les doublons, mais une conversion inutile peut ralentir le programme.

Utilisez join() pour assembler des listes de chaînes de caractères - Si vous travaillez avec des listes de chaînes de caractères que vous devez assembler, join() est beaucoup plus performant que de faire des concaténations successives. Par exemple :

```
mots = ["Python", "est", "génial"]
phrase = " ".join(mots)
print(phrase)  # Affiche "Python est génial"
```

Les list comprehension

On peut construire une list à partir d'une boucle for et de la méthode append() :

```
cubes = []
for i in range(10):
    cubes.append(i**3)
print(cubes)    # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Cependant Python possède une syntaxe plus concise et plus élégante, les **list comprehension** :

```
cubes = [ i**3 for i in range(10)]
print(cubes)    # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Attention !!! La variable utilisée à l'intérieur de la **list comprehension** est locale à celle ci :

```
i = 99
cub
print(cubes)    # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Il est possible d'ajouter un prédicat à une **list comprehension** :

```
liste = [ i**3 for i in range(10) if i%2 == 0]
print(liste)     # [0, 8, 64, 216, 512]
```

ou d'imbriquer des boucles, qui sont exécutées dans l'ordre d'apparition dans la **list comprehension** :

```
liste = [i+str(j) for i in 'ab' for j in range(3)]
print(liste)     # ['a0', 'a1', 'a2', 'b0', 'b1', 'b2']
```

On peut également utiliser une clause *else* dans le prédicat. Dans ce cas, le test est placé avant *for* :

```
marks = [8, 12, 7, 15, 16, 13, 4, 14, 9, 15]
liste = [ 'pass' if x >=10 else 'failed' for x in marks]
print(liste)
# ['failed', 'pass', 'failed', 'pass', 'pass', 'pass', 'failed', 'pass',
↪  'failed', 'pass']
```

En appliquant ces bonnes pratiques, vous pourrez manipuler les listes en Python de manière optimale. Les listes sont un outil puissant et en suivant ces conseils, vous réduirez les risques d'erreurs tout en maximisant les performances de votre code. Les listes permettent une gestion flexible des données et ce guide vous donne toutes les clés pour les maîtriser avec efficacité.

En conclusion

Les listes en Python sont un élément central de la programmation, offrant une combinaison unique de flexibilité et de simplicité. Que vous travailliez sur des projets de petite envergure ou que vous manipuliez des données complexes, les listes sont incontournables pour structurer et gérer les informations.

TP : Gestion d'une Liste de Courses

Énoncé : Cet exercice vous guide pas à pas dans la création d'un programme Python qui permet de gérer une liste de courses. Vous allez apprendre à utiliser les opérations de base sur les listes, les boucles, les conditions et les fonctions pour créer une petite application en ligne de commande.

Objectif : Créer un programme Python interactif permettant d'ajouter, de supprimer, de modifier et de visualiser les articles d'une liste de courses.

Étape 1 : Initialiser la Liste

1. Créez une liste vide nommée *liste_courses* pour stocker les articles.
2. Demandez à l'utilisateur s'il souhaite commencer avec une liste d'articles. S'il répond "oui", invitez-le à entrer des articles séparés par des virgules. Utilisez la méthode appropriée pour ajouter ces articles à *liste_courses*.
3. Affichez la liste initiale des articles, qu'ils aient été ajoutés ou non.

Solution :

```
# Initialisation de la liste de courses
liste_courses = []

# Demande d'initialisation
initial_input = input("Souhaitez-vous commencer avec une liste d'articles ?
↪  (oui/non) ")
if initial_input.lower() == "oui":
    articles = input("Entrez les articles de départ, séparés par des virgules : ")
    liste_courses = articles.split(",")
    liste_courses = [article.strip() for article in liste_courses]  # Nettoyage
    ↪  des espaces

print("Liste initiale :", liste_courses)
```

Étape 2 : Ajouter un Article

1. Demandez à l'utilisateur de saisir un article à ajouter à la liste.
2. Ajoutez cet article à la liste *liste_courses*.
3. Affichez un message confirmant que l'article a bien été ajouté et montrez la liste mise à jour.

Solution :

```
# Ajouter un article
def ajouter_article(liste):
    article = input("Entrez l'article à ajouter : ")
    liste.append(article)
    print(f"L'article '{article}' a été ajouté à la liste.")
    print("Liste mise à jour :", liste_courses)

ajouter_article(liste_courses)
```

Étape 3 : Supprimer un Article

1. Demandez à l'utilisateur quel article il souhaite retirer de la liste.
2. Vérifiez que cet article est bien présent dans *liste_courses*. Si c'est le cas, supprimez-le. Sinon, affichez un message indiquant que l'article n'est pas dans la liste.
3. Affichez la liste mise à jour après chaque suppression.

Solution :

```
# Supprimer un article
def supprimer_article(liste):
    article = input("Entrez l'article à supprimer : ")
    if article in liste:
        liste.remove(article)
        print(f"L'article '{article}' a été supprimé de la liste.")
    else:
        print(f"L'article '{article}' n'est pas dans la liste.")
    print("Liste mise à jour :", liste_courses)

supprimer_article(liste_courses)
```

Étape 4 : Modifier un Article

1. Demandez à l'utilisateur quel article il souhaite modifier.
2. Vérifiez que cet article est bien présent dans liste_courses. S'il est présent, demandez à l'utilisateur de saisir la nouvelle valeur pour cet article et mettez la liste à jour.
3. Affichez la liste actualisée après chaque modification.

Solution :

```
# Modifier un article
def modifier_article(liste):
    article = input("Entrez l'article à modifier : ")
    if article in liste:
        nouvel_article = input("Entrez la nouvelle valeur : ")
        index = liste.index(article)
        liste[index] = nouvel_article
        print(f"L'article '{article}' a été modifié en '{nouvel_article}'.")
```

```
    else:
        print(f"L'article '{article}' n'est pas dans la liste.")
    print("Liste mise à jour :", liste_courses)

modifier_article(liste_courses)
```

Étape 5 : Afficher la Liste

1. Ajoutez une option pour permettre à l'utilisateur d'afficher la liste complète de ses courses.
2. Formatez l'affichage pour que chaque article soit précédé de son numéro (ex. "1. Pain", "2. Lait", etc.).

Solution :

```
# Afficher la liste des courses
def afficher_liste(liste):
    print("Votre liste de courses :")
    for i, article in enumerate(liste, start=1):
        print(f"{i}. {article}")

afficher_liste(liste_courses)
```

Étape 6 : Menu Interactif

1. Ajoutez un menu principal qui affiche les options suivantes :

   — "1 : Afficher la liste des courses"
   — "2 : Ajouter un article"
   — "3 : Supprimer un article"
   — "4 : Modifier un article"
   — "5 : Quitter"

2. Faites en sorte que le programme répète ce menu après chaque action, jusqu'à ce que l'utilisateur choisisse de quitter.

Solution :

```python
# Menu principal
def menu():
    print("\nMenu :")
    print("1 : Afficher la liste des courses")
    print("2 : Ajouter un article")
    print("3 : Supprimer un article")
    print("4 : Modifier un article")
    print("5 : Quitter")

    choix = input("Entrez le numéro de votre choix : ")
    return choix

while True:
    choix = menu()
    if choix == "1":
        afficher_liste(liste_courses)
    elif choix == "2":
        ajouter_article(liste_courses)
    elif choix == "3":
        supprimer_article(liste_courses)
    elif choix == "4":
        modifier_article(liste_courses)
    elif choix == "5":
        print("Merci d'avoir utilisé le programme !")
        break
    else:
        print("Option invalide. Veuillez choisir une option du menu.")
```

Étape 7 : Gérer les Erreurs d'Entrée

1. Assurez-vous que votre programme peut gérer les erreurs d'entrée courantes :

   — Si l'utilisateur essaie de supprimer ou de modifier un article qui n'est pas dans la liste, affichez un message d'erreur.
   — Si l'utilisateur entre une option de menu invalide, affichez un message d'erreur et affichez de nouveau le menu.

Solution : Dans ce code, les erreurs courantes sont déjà gérées dans chaque fonction (supprimer_article et modifier_article vérifient la présence de l'article avant d'effectuer des modifications et le menu inclut une vérification des options).

Étape 8 : Rendre le Code Plus Lisible avec des Fonctions

1. Créez des fonctions pour chaque action (par exemple : afficher_liste(), ajouter_article(), supprimer_article(), etc.).
2. Modifiez votre programme pour qu'il utilise ces fonctions, ce qui le rendra plus lisible et organisé.

Solution : Chaque étape est déjà encapsulée dans des fonctions (ajouter_article(), supprimer_article(), modifier_article() et afficher_liste()), rendant le programme plus structuré et facile à lire.

3.3.2 Les tuples

Après avoir vu les **listes** en détail, nous allons maintenant nous pencher sur les **tuples** en Python. Les **tuples sont une autre structure de données en Python** qui est similaire aux listes, mais avec quelques différences clés.

Qu'est-ce que sont les tuples ?

Les tuples en Python sont des collections ordonnées et immuables de valeurs. Ils sont similaires aux listes, mais les tuples sont immuables, ce qui signifie que leurs éléments ne peuvent pas être modifiés une fois qu'ils ont été définis.

Pourquoi utiliser des tuples ?

Les tuples sont utiles dans de nombreux cas, notamment lorsque vous avez besoin d'une collection de valeurs qui ne doit pas être modifiée. Les tuples sont également plus rapides que les listes, car ils sont immuables et ne nécessitent pas de gestion de la mémoire supplémentaire.

Créer un tuple en Python

La création de tuples en Python est simple et repose sur l'utilisation de parenthèses et de virgules pour rassembler les éléments. Contrairement aux listes, **les tuples sont immuables**, ce qui signifie qu'ils ne peuvent pas être **modifiés** après leur création. Cette caractéristique en fait un excellent choix pour stocker des données fixes ou des éléments que vous ne voulez pas modifier accidentellement au cours du programme.

Pour créer un tuple, il suffit de mettre les éléments entre parenthèses et de les séparer par des virgules. Voici quelques exemples simples de tuples en Python :

```
# Création d'un tuple de plusieurs éléments
mon_tuple = (1, 2, 3, 4)
print(mon_tuple)  # Output : (1, 2, 3, 4)

# Création d'un tuple contenant des types variés
tuple_mixte = (1, "Python", 3.14, True)
print(tuple_mixte)  # Output : (1, 'Python', 3.14, True)
```

La syntaxe du tuple est très proche de celle de la liste, à la différence près que les tuples utilisent des parenthèses *()* alors que les listes utilisent des crochets *[]*.

Python permet également de créer des tuples sans utiliser les parenthèses. Si vous écrivez une suite de valeurs séparées par des virgules, Python les interprétera automatiquement comme un tuple.

```
tuple_sans_parentheses = 10, 20, 30
print(tuple_sans_parentheses)  # Output : (10, 20, 30)
```

Cependant, pour des raisons de clarté, il est recommandé d'utiliser des parenthèses, surtout lorsque le tuple est plus complexe ou contient plusieurs types d'éléments.

Pour un tuple contenant un seul élément, il est essentiel de mettre une virgule après cet élément, même si cela peut sembler contre-intuitif. Sans cette virgule, Python ne reconnaît pas la structure comme un tuple et interprète simplement l'élément comme son type de base.

```
# Tuple avec un seul élément
tuple_seul_element = (42,)
print(tuple_seul_element)  # Output : (42,)

# Sans virgule, ce n'est pas un tuple
pas_un_tuple = (42)
print(pas_un_tuple)  # Output : 42
```

Cette particularité est importante pour éviter les erreurs lorsque vous créez des tuples à élément unique. La virgule informe Python que l'élément est dans un tuple et non isolé.

Vous pouvez aussi utiliser la fonction tuple() pour créer un tuple à partir d'autres types **itérables**, comme une liste ou une chaîne de caractères. C'est particulièrement utile lorsque vous souhaitez convertir un type **mutable** en une **structure immuable** :

```
# Créer un tuple à partir d'une liste
liste_exemple = [1, 2, 3, 4]
tuple_a_partir_liste = tuple(liste_exemple)
print(tuple_a_partir_liste)  # Output : (1, 2, 3, 4)

# Créer un tuple à partir d'une chaîne
chaine = "Python"
tuple_a_partir_chaine = tuple(chaine)
print(tuple_a_partir_chaine)  # Output : ('P', 'y', 't', 'h', 'o', 'n')
```

Accéder aux éléments d'un tuple

Une fois qu'un tuple est créé, il est souvent nécessaire d'accéder aux éléments qu'il contient pour les manipuler ou les utiliser dans le code. Comme les tuples sont des séquences indexées, chaque élément possède une position unique, appelée ***index***, qui permet d'y accéder facilement. Nous verrons les différentes techniques pour accéder aux éléments d'un tuple en Python.

Accéder aux éléments par index - L'indexation est la méthode la plus courante pour accéder aux éléments d'un tuple. Chaque élément d'un tuple est repéré par un index, qui commence à 0 pour le premier élément et augmente de un pour chaque élément suivant. L'utilisation de l'index est simple et directe :

```
mon_tuple = (10, 20, 30, 40, 50)
print(mon_tuple[0])  # Output : 10
print(mon_tuple[2])  # Output : 30
```

Indexation négative - Python offre la possibilité d'utiliser des **index négatifs** pour accéder aux éléments en partant de la fin du tuple. L'index *-1* représente le dernier élément, *-2* l'avant-dernier et ainsi de suite. Cette méthode est pratique lorsque vous souhaitez accéder aux derniers éléments sans connaître la longueur exacte du tuple.

```
mon_tuple = (10, 20, 30, 40, 50)
print(mon_tuple[-1])  # Output : 50 c'est le dernier élément du tuples.
print(mon_tuple[-3])  # Output : 30 c'est le troisième élément en partant de la
↪  fin.
```

Extraction de sous-tuples (slicing) - L'extraction de sous-tuples (ou slicing) permet de récupérer une portion spécifique d'un tuple en utilisant une syntaxe basée sur les indices. La syntaxe pour le slicing est *tuple[start :stop :step]* :

- *start* est l'indice de début (inclus).
- *stop* est l'indice de fin (exclu).
- *step* est le pas entre chaque élément.

Voici quelques exemples pour illustrer cette fonctionnalité :

```
mon_tuple = (10, 20, 30, 40, 50)

# Extraire les trois premiers éléments
print(mon_tuple[0:3])  # Output : (10, 20, 30)

# Extraire du deuxième au quatrième élément
print(mon_tuple[1:4])  # Output : (20, 30, 40)

# Extraire tous les éléments avec un pas de 2
print(mon_tuple[::2])  # Output : (10, 30, 50)
```

Le slicing est très flexible et permet d'obtenir des sous-parties d'un tuple sans devoir le modifier. Notez que comme les tuples sont immuables, le slicing retourne toujours un nouveau tuple.

Accéder aux éléments dans un tuple imbriqué - Les tuples imbriqués sont des tuples qui contiennent d'autres tuples. Pour accéder à un élément spécifique dans un tuple imbriqué, il suffit de combiner plusieurs index pour naviguer dans les différentes couches de la structure :

```
tuple_imbrique = (1, 2, (3, 4, 5), (6, 7), 8)

# Accéder à l'élément 3 dans le sous-tuple (3, 4, 5)
```

```
print(tuple_imbrique[2][0])  # Output : 3

# Accéder à l'élément 7 dans le sous-tuple (6, 7)
print(tuple_imbrique[3][1])  # Output : 7
```

Erreurs courantes d'indexation - Étant donné que les tuples sont immuables, il est impossible de modifier un élément directement via son index. Toute tentative de modification d'un élément d'un tuple entraîne une erreur. Voici un exemple d'erreur courante liée à l'indexation :

```
mon_tuple = (1, 2, 3, 4)

# Essayer de modifier un élément
mon_tuple[0] = 10  # Error : TypeError: 'tuple' object does not support item
↪  assignment
```

Python renverra une **TypeError** pour indiquer que l'objet tuple ne peut pas être modifié. Si vous devez modifier des valeurs, vous devrez créer un nouveau tuple avec les valeurs souhaitées.

Les opérations de base sur les tuples

Les **tuples** en Python, bien qu'**immutables**, permettent d'effectuer plusieurs opérations pratiques pour manipuler et interroger leur contenu. Dans ce chapitre, nous explorerons les opérations courantes, les fonctions intégrées et les techniques pour utiliser efficacement les tuples dans vos programmes.

Concaténation de tuples - La **concaténation** permet de combiner plusieurs tuples pour en former un nouveau. L'opérateur + est utilisé pour joindre deux tuples en un seul, sans modifier les tuples d'origine.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple_concatenation = tuple1 + tuple2
print(tuple_concatenation)  # Output : (1, 2, 3, 4, 5, 6)
```

Répétition d'un tuple - L'opérateur * permet de répéter un tuple plusieurs fois pour en créer un plus grand contenant des éléments identiques en série. Cette opération est utile pour créer des structures régulières ou des motifs répétitifs.

```
tuple_repetition = (1, 2) * 3
print(tuple_repetition)  # Output : (1, 2, 1, 2, 1, 2)
```

Vérification de l'appartenance avec in - L'opérateur ***in*** est utilisé pour vérifier si un **élément spécifique** est présent dans un tuple. Cette opération renvoie True si l'élément est trouvé, sinon False.

```
tuple_elements = (1, 2, 3, 4, 5)

# Vérifier si 3 est dans le tuple
print(3 in tuple_elements)  # Output : True

# Vérifier si 6 est dans le tuple
print(6 in tuple_elements)  # Output : False
```

L'opérateur ***in*** est pratique pour effectuer des vérifications rapides dans des structures de données immuables comme les tuples.

Utilisation des fonctions intégrées len(), min() et max() Python propose plusieurs fonctions intégrées pour travailler avec les tuples, notamment :

- **len()** : renvoie le nombre d'éléments dans le tuple.
- **min()** : renvoie la plus petite valeur présente dans le tuple.
- **max()** : renvoie la plus grande valeur présente dans le tuple.

Ces fonctions permettent d'obtenir des informations essentielles sur les tuples.

```
mon_tuple = (10, 20, 30, 40)

# Nombre d'éléments dans le tuple
print(len(mon_tuple))  # Output : 4

# Valeur minimale
print(min(mon_tuple))  # Output : 10

# Valeur maximale
print(max(mon_tuple))  # Output : 40
```

Ces fonctions sont particulièrement utiles pour traiter des tuples numériques ou pour obtenir rapidement des valeurs clés.

Compter les occurrences d'un élément avec count() - La méthode count() permet de compter le nombre de fois qu'un élément spécifique apparaît dans un tuple. C'est pratique pour analyser les données contenues dans un tuple sans avoir à écrire de boucle.

```
tuple_compte = (1, 2, 3, 2, 4, 2)

# Compter le nombre de fois que 2 apparaît
print(tuple_compte.count(2))  # Output : 3
```

Trouver l'index d'un élément avec index() - La méthode index() permet de trouver la première occurrence d'un élément dans un tuple et retourne l'indice correspondant. Si l'élément n'est pas présent, elle déclenche une erreur ValueError.

```
tuple_index = (10, 20, 30, 40)

# Trouver l'index de l'élément 30
print(tuple_index.index(30))  # Output : 2
```

Attention, si l'élément est absent, il est conseillé de vérifier son existence avec in avant d'utiliser index() pour éviter des erreurs.

Comparaison de tuples - Les tuples peuvent être comparés avec les opérateurs de **comparaison standard** (==, !=, <, >, <=, >=). Python compare les tuples en analysant chaque élément dans l'ordre et dès qu'il trouve une différence, il détermine le résultat de la comparaison.

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)

# Comparer les tuples
print(tuple1 == tuple2)  # Output : False
print(tuple1 < tuple2)   # Output : True
```

Dans cet exemple, *tuple1* est considéré comme étant plus petit que *tuple2* car l'élément *3* dans *tuple1* est inférieur à *4* dans *tuple2*.

Conversion d'un tuple en liste - Bien que les tuples soient immuables, il est possible de les **convertir en liste** pour les manipuler plus librement. La conversion se fait en utilisant la fonction list() et permet de créer une liste avec les mêmes éléments.

```
mon_tuple = (1, 2, 3, 4)
liste = list(mon_tuple)
print(liste)  # Output : [1, 2, 3, 4]
```

Cette méthode est utile si vous avez besoin de modifier les éléments d'un tuple temporairement, puis de les reconstituer dans un tuple si nécessaire.

Les tuples nommés (namedtuples) - Les **tuples nommés** (ou **namedtuples**) sont une extension des tuples standards en Python, permettant de donner un nom aux éléments d'un tuple pour améliorer la lisibilité et la maintenabilité du code. Ils sont particulièrement utiles lorsqu'on souhaite représenter des données structurées de manière immuable, tout en facilitant l'accès aux éléments par des noms explicites au lieu d'indices.

Les **namedtuples** font partie du module collections de Python et sont souvent utilisés pour créer des structures de données légères et faciles à utiliser, idéales pour remplacer les objets simples ou les tuples classiques lorsque des noms d'attributs sont nécessaires.

Créer un namedtuple - Pour créer un **namedtuple**, on utilise la fonction namedtuple() du module collections. Il faut d'abord importer **namedtuple**, puis définir la structure en lui donnant un nom et une liste d'attributs. Voici comment créer un **namedtuple** de base :

```python
from collections import namedtuple

# Création d'un namedtuple appelé Point avec deux attributs : x et y
Point = namedtuple("Point", ["x", "y"])

# Créer une instance de Point
p = Point(10, 20)
print(p)  # Output : Point(x=10, y=20)
```

Dans cet exemple, *Point* est un **namedtuple** avec deux attributs, x et y. L'objet p est une instance de Point contenant les valeurs 10 pour x et 20 pour y.

Accéder aux éléments d'un namedtuple - Une fois le namedtuple créé, vous pouvez accéder à ses éléments de la même manière qu'avec un tuple standard, en utilisant des **indices**, mais aussi en utilisant des **noms d'attributs** pour plus de clarté.

```python
# Accès aux éléments par nom
print(p.x)  # Output : 10
print(p.y)  # Output : 20

# Accès aux éléments par indice
print(p[0])  # Output : 10
print(p[1])  # Output : 20
```

Accéder aux éléments par nom rend le code plus lisible, car $p.x$ et $p.y$ sont plus explicites que *p[0]* et *p[1]*.

Utilisation pratique des namedtuples - Les namedtuples sont particulièrement utiles pour représenter des données structurées sans avoir à utiliser des classes complètes. Par exemple, ils peuvent être utilisés pour représenter des points géométriques, des dates, des coordonnées GPS, ou même des enregistrements de bases de données.

Exemple : Représentation d'une personne

```python
# Création d'un namedtuple pour représenter une personne
Personne = namedtuple("Personne", ["nom", "prenom", "age"])

# Créer une instance de Personne
personne = Personne("Dupont", "Alice", 30)
print(personne.nom)     # Output : Dupont
print(personne.prenom)  # Output : Alice
print(personne.age)     # Output : 30
```

Dans cet exemple, le namedtuple *Personne* est utilisé pour stocker les informations d'une personne de manière structurée.

Convertir un namedtuple en dictionnaire - Il peut être utile de convertir un namedtuple en dictionnaire pour des traitements spécifiques. Les namedtuples disposent de la méthode _asdict() qui retourne un OrderedDict avec les noms d'attributs comme clés.

```
# Conversion d'un namedtuple en dictionnaire
personne_dict = personne._asdict()
print(personne_dict)  # Output : {'nom': 'Dupont', 'prenom': 'Alice', 'age': 30}
```


```
# Conversion d'un namedtuple en dictionnaire
personne_dict = personne._asdict()
print(personne_dict)  # Output : {'nom': 'Dupont', 'prenom': 'Alice', 'age': 30}
```


Cette méthode est pratique pour manipuler les données du namedtuple comme un dictionnaire, par exemple pour des opérations de formatage ou pour l'exporter en JSON.

Remplacer les valeurs dans un namedtuple avec $_{r}eplace()$ – Bien que les namedtuples soient immuables, la méthode _replace() permet de créer une nouvelle instance en modifiant un ou plusieurs attributs. Cette opération est utile lorsque vous avez besoin de créer une version modifiée sans altérer l'original.

```
# Remplacer l'âge dans le namedtuple
personne_modifiee = personne._replace(age=31)
print(personne_modifiee)  # Output : Personne(nom='Dupont', prenom='Alice',
↪  age=31)
```

Dans cet exemple, personne_modifiee est une nouvelle instance de Personne avec l'âge mis à jour, sans modifier l'objet personne original.

Utilisation des tuples pour le retour de multiples valeurs

En Python, les fonctions peuvent retourner plusieurs valeurs en utilisant un tuple. Cela est particulièrement utile pour transmettre plusieurs résultats ou informations d'une fonction sans nécessiter de structures complexes. Lorsque vous retournez plusieurs valeurs dans une fonction, Python les regroupe automatiquement dans un tuple, simplifiant ainsi le code et améliorant sa lisibilité.

Dans cette partie, nous allons voir comment retourner plusieurs valeurs avec un tuple, comment décomposer ces valeurs (ou les "déballer") lors de leur réception et examiner quelques exemples d'utilisation pratiques.

```
# Fonction qui retourne plusieurs valeurs
def calculs(a, b):
    somme = a + b
    produit = a * b
    return somme, produit  # Retourne un tuple (somme, produit)

resultat = calculs(5, 3)
print(resultat)  # Output : (8, 15)
```

Déballer les valeurs d'un tuple lors du retour - Lorsque vous recevez un tuple contenant plusieurs valeurs, vous pouvez les déballer en affectant chaque valeur à une variable séparée. Cela rend le code plus clair et rend chaque valeur immédiatement accessible.

```
# Déballer les valeurs retournées
somme, produit = calculs(5, 3)
print(somme)    # Output : 8
print(produit)  # Output : 15
```

Ici, *somme* et *produit* reçoivent respectivement les valeurs 8 et 15 du tuple retourné par la fonction, ce qui évite d'avoir à accéder aux valeurs en utilisant des indices.

Cas d'usage des tuples pour le retour de valeurs multiples - L'utilisation des tuples pour retourner plusieurs valeurs est particulièrement utile dans des situations où une fonction doit effectuer plusieurs calculs ou vérifier plusieurs conditions. Voici quelques cas pratiques :

Exemple 1 : Retourner des statistiques

Une fonction qui calcule les statistiques de base sur une liste de nombres peut retourner plusieurs valeurs utiles (moyenne, minimum et maximum) sous forme de tuple.

```
def statistiques(data):
    moyenne = sum(data) / len(data)
    minimum = min(data)
    maximum = max(data)
    return moyenne, minimum, maximum

# Utiliser la fonction et déballer les résultats
data = [10, 20, 30, 40, 50]
moyenne, minimum, maximum = statistiques(data)
print("Moyenne:", moyenne)    # Output : Moyenne : 30.0
print("Min:", minimum)        # Output : Min : 10
print("Max:", maximum)        # Output : Max : 50
```

Dans cet exemple, la fonction statistiques() retourne trois valeurs sous forme de tuple, ce qui permet de capturer toutes les informations utiles sans nécessiter de structures de données supplémentaires.

Exemple 2 : Validation de données

Une fonction peut également retourner plusieurs informations après avoir validé des données, comme un booléen indiquant la validité et un message d'erreur si les données ne sont pas valides.

```
def valider_utilisateur(nom, age):
    if not nom:
        return False, "Nom requis"
    if age < 0:
        return False, "Âge invalide"
    return True, "Utilisateur valide"

# Déballer les valeurs pour récupérer le statut et le message
statut, message = valider_utilisateur("Alice", 25)
print(statut)   # Output : True
print(message)  # Output : Utilisateur valide
```

Dans cet exemple, valider_utilisateur() retourne un tuple avec deux valeurs : un booléen indiquant

si l'utilisateur est valide et un message descriptif. Ce modèle de retour est pratique pour des vérifications ou validations dans lesquelles plusieurs résultats doivent être retournés.

TP sur les tuples

Dans cet exercice, vous allez construire pas à pas un programme en Python pour gérer les informations des étudiants d'une classe en utilisant des tuples. L'objectif est de pratiquer la manipulation des tuples en tant que structures de données immuables et d'apprendre comment organiser des informations lorsque celles-ci ne sont pas censées être modifiées après leur création.

Scénario : Vous êtes responsable de la gestion des informations des étudiants d'une classe. Chaque étudiant possède des informations immuables telles que son nom, son prénom, son âge et sa note finale. Vous allez suivre chaque étape pour construire votre programme en utilisant des tuples pour chaque étudiant et un dictionnaire pour stocker tous les étudiants.

Étapes de l'exercice

1. **Créer un tuple pour un étudiant**

   — Créez un tuple pour un étudiant contenant les informations suivantes : nom, prénom, âge et note finale. Remplissez-le avec un exemple de votre choix.

   Solution :

```
# Exemple de tuple pour un étudiant
etudiant1 = ("Durand", "Alice", 20, 85)
```

2. **Créer une liste d'étudiants sous forme de dictionnaire**

   — Créez un dictionnaire vide appelé classe pour stocker les informations de plusieurs étudiants.
   — Ajoutez le tuple de l'étudiant que vous avez créé dans la première étape en lui assignant un identifiant unique (comme "Etudiant1").

   Solution :

```
# Création de la classe et ajout du premier étudiant
classe = {}
classe["Etudiant1"] = etudiant1
```

3. **Ajouter plusieurs étudiants à la classe**

   — Créez au moins deux autres tuples pour représenter d'autres étudiants avec les mêmes informations (nom, prénom, âge, note finale).
   — Ajoutez-les au dictionnaire en utilisant des identifiants uniques (comme "Etudiant2", "Etudiant3", etc.).

Solution :

```
# Exemple de deux autres étudiants
etudiant2 = ("Martin", "Bob", 22, 78)
etudiant3 = ("Leclerc", "Claire", 19, 92)

# Ajout des étudiants à la classe
classe["Etudiant2"] = etudiant2
classe["Etudiant3"] = etudiant3
```

4. Afficher les informations de tous les étudiants

   — Parcourez le dictionnaire ***classe*** pour afficher les informations de chaque étudiant. Utilisez une boucle ***for*** pour afficher chaque étudiant sous la forme :

```
Etudiant1:
Nom : [nom]
Prénom : [prénom]
Âge : [âge]
Note finale : [note finale]
```

Solution :

```
# Parcourir et afficher les informations de tous les étudiants
for identifiant, etudiant in classe.items():
    print(f"{identifiant}:")
    print(f"  Nom : {etudiant[0]}")
    print(f"  Prénom : {etudiant[1]}")
    print(f"  Âge : {etudiant[2]}")
    print(f"  Note finale : {etudiant[3]}")
```

5. **Rechercher un étudiant par nom**

   — Demandez à l'utilisateur de saisir le nom d'un étudiant.
   — Parcourez classe pour vérifier si un étudiant avec ce nom existe.
   — Si l'étudiant est trouvé, affichez toutes ses informations. Sinon, affichez un message indiquant que l'étudiant n'a pas été trouvé.

Solution :

```
# Recherche d'un étudiant par nom
nom_recherche = input("Entrez le nom de l'étudiant que vous
↪ recherchez : ")
trouve = False
for etudiant in classe.values():
    if etudiant[0].lower() == nom_recherche.lower():
        print(f"Nom : {etudiant[0]}")
```

```
            print(f"Prénom : {etudiant[1]}")
            print(f"Âge : {etudiant[2]}")
            print(f"Note finale : {etudiant[3]}")
            trouve = True
            break

    if not trouve:
        print("Étudiant non trouvé dans la classe.")
```

6. Mettre à jour la note d'un étudiant

   — Demandez à l'utilisateur de saisir l'identifiant d'un étudiant (par exemple, "Etudiant1").
   — Puis demandez-lui de saisir une nouvelle note.
   — Mettez à jour la note finale en recréant le tuple correspondant pour l'étudiant dans le dictionnaire ***classe***.

Solution :

```
# Mise à jour de la note d'un étudiant
identifiant = input("Entrez l'identifiant de l'étudiant à modifier
↪ (par exemple, 'Etudiant1') : ")
if identifiant in classe:
    nouvelle_note = int(input("Entrez la nouvelle note finale : "))
    # On recrée le tuple avec la nouvelle note en gardant les autres
    ↪ informations
    ancien_tuple = classe[identifiant]
    classe[identifiant] = (ancien_tuple[0], ancien_tuple[1],
    ↪ ancien_tuple[2], nouvelle_note)
    print("La note a été mise à jour.")
else:
    print("Identifiant non trouvé.")
```

7. **Supprimer un étudiant de la classe**

   — Demandez à l'utilisateur de saisir l'identifiant d'un étudiant à supprimer (par exemple, "Etudiant2").
   — Supprimez l'étudiant correspondant du dictionnaire ***classe***.

Solution :

```
# Suppression d'un étudiant
identifiant = input("Entrez l'identifiant de l'étudiant à supprimer
↪ (par exemple, 'Etudiant2') : ")
if identifiant in classe:
    classe.pop(identifiant)
    print(f"{identifiant} a été supprimé de la classe.")
else:
```

```
        print("Étudiant non trouvé.")
```

8. Afficher les étudiants ayant une note supérieure à une valeur donnée

   — Demandez à l'utilisateur de saisir une note minimale.
   — Parcourez ***classe*** et affichez les informations de tous les étudiants ayant une note finale supérieure ou égale à cette valeur.

Solution :

```
# Afficher les étudiants ayant une note supérieure à une certaine
↪ valeur
note_minimale = int(input("Entrez la note minimale : "))
for identifiant, etudiant in classe.items():
    if etudiant[3] >= note_minimale:
        print(f"{identifiant}: {etudiant[1]} {etudiant[0]}, Âge:
        ↪ {etudiant[2]}, Note finale: {etudiant[3]}")
```

Cet exercice vous a permis d'explorer la manipulation des tuples et leur utilisation pour stocker des informations immuables. En créant et gérant une classe d'étudiants avec des tuples, vous avez appris à organiser et traiter des données sans modifier leur structure initiale.

Conclusion

La mise en pratique des tuples en Python permet de structurer et manipuler efficacement des données immuables. Que ce soit pour retourner plusieurs valeurs, stocker des données fixes, ou organiser des collections de données plus complexes, les tuples offrent une simplicité et une sécurité d'utilisation qui les rendent indispensables dans de nombreux cas. En les utilisant dans vos scripts, vous constaterez rapidement leur avantage en termes de performance et de lisibilité du code.

3.3.3 Les sets

Les *set* sont des containers mutables non ordonnés qui possèdent les caractéristiques des ensembles mathématiques. En particulier, ils ne disposent pas d'**éléments dupliqués**. Ils se définissent à partir des opérateurs { et } ou du constructeur *set*.

Les *set* représentent donc un ensemble de valeur non ordonnées et sans doublons de valeurs. Dans un set, chaque élément doit être **unique** et **immuable** , c'est-à- dire qu'il ne peut pas être modifié.

On peut donc stocker des **chaînes de caractères**, des nombres et des **tuples** dans un *set* mais pas de **listes** ou de **dictionnaires** !

En revanche, le set en lui-même peut être modifié ! On peut y ajouter des objets et aussi en supprimer.

Les sets sont souvent utilisés pour contrôler les **doublons** et effectuer des opérations sur les ensembles comme les unions, les différences, les intersections, etc.

Nous allons vous présenter la liste des opérations que vous pouvez effectuer avec les *sets*.

Créer un set vide

Pour définir un set vide, il faut passer par le constructeur de la classe *sets*

```
# Création d'un set vide
empty_set = set()
print(type(empty_set))
len(empty_set)
```

la syntaxe empty_set = {} définie un dictionnaire et non un set. Il est donc obligatoire de passer par le constructeur de la classe.

Créer un set et l'initialiser

Lorsque vous définissez un set avec cette fonction, vous devez passer un objet itérable comme une liste, un tuple ou une chaîne de caractères :

```
s = set() # Set vide
s = set(['spam', 'eggs', 'knights', 'ham'])
print(s)    # {'ham', 'knights', 'spam', 'eggs'}
s = set(('spam', 'eggs', 'knights', 'ham'))
print(s)    # {'ham', 'knights', 'spam', 'eggs'}
s = set('eggs')
print(s)    # {'e', 's', 'g'}
```

Comme vous le constatez, vous pouvez passer une liste en argument alors que c'est un objet **muable**. Cependant, la fonction *set()* retourne bien un set dans lequel la liste a été décomposée en chaînes de caractères, qui elles, sont bien des objets **immuables**.

L'autre manière de faire est d'utiliser les accolades {} :

```
s = {item1, item2, item3}
s = {'spam', 'eggs', 'knights'}
print(s)    # {'spam', 'eggs', 'knights'}
```

Chaque objet passé entre les accolades devient un item distinct du set , que cet objet soit itérable ou non. C'est ce qui différencie les deux méthodes de création d'un set ! Par exemple :

```
s = set('eggs')
print(s)    # {'e', 's', 'g'} => Itérable où chaque élément est évalué
s = {'eggs'}
print(s)    # {'eggs'}        => Reste un élément distinct du set
```

```
[1]: mon_ensemble = {10, "coconut"}
     element = max(mon_ensemble)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 mon_ensemble = {10, "coconut"}
----> 2 element = max(mon_ensemble)

TypeError: '>' not supported between instances of 'str' and 'int'
```

Les opérations sur les ensembles

Connaître la taille d'un ensemble - Pour connaître le nombre d'éléments d'un ensemble, il faut utiliser la fonction len() (qui est la contraction de length) :

```
mon_ensemble = {10, 20, 30, 40}
taille_ensemble = len(mon_ensemble)
print(taille_ensemble) # affiche 4
```

Savoir si un élément est présent dans un ensemble - Pour savoir si un élément est présent dans un ensemble, on peut utiliser le mot-clé *in* :

```
mon_ensemble = {1, 2, 3}
print(2 in mon_ensemble)        # True
print(12 in mon_ensemble)       # False
print(12 not in mon_ensemble)   # True
```

Comparer deux ensembles - On peut utiliser tous les opérateurs de comparaison entre deux ensembles.

```
mon_ensemble = {1, 2, 3}
print(mon_ensemble == {1, 2, 3})        # True
print(mon_ensemble != {})               # True
print(mon_ensemble < {1, 2, 3, 4})      # True
print(mon_ensemble > {2})               # True
```

Pour que deux ensembles soient égaux, il faut qu'ils contiennent les mêmes éléments.

Connaître le plus grand élément d'un ensemble - Pour connaître le plus grand élément d'un ensemble, il faut utiliser la fonction *max()* :

```
mon_ensemble = {10, 20, 30, 40}
element = max(mon_ensemble)
print(element)  # affiche 40
```

Cela suppose néanmoins que tous les éléments d'un ensemble sont comparables et donc de type similaire sinon on obtient une erreur de type *TypeError*.

Connaître le plus petit élément d'un ensemble - Pour connaître le plus petit élément d'un ensemble, il faut utiliser la fonction *min()* :

```
mon_ensemble = {"a", "b", "c"}
element = min(mon_ensemble)
print(element)  # affiche a
```

Comme pour la méthode *max()*, cela suppose que tous les éléments d'un ensemble sont **comparables** et donc de **type similaire**.

Quelques méthodes pour les ensembles

copy() : Copie un ensemble pour créer un nouvel ensemble.

```
mon_ensemble = {10, 20, 30}
nouvel_ensemble = mon_ensemble.copy()
```

Pour rappel, on ne peut pas modifier les éléments d'un set car ils sont **immuables** ! Du coup, Python met à notre disposition deux méthodes (add() et update()) pour ajouter des éléments à l'intérieur d'un set !

add() : Ajoute un élément à un ensemble.

```
s = {'spam', 'eggs', 'knights'}
s.add('ham')
print(s) # {'ham', 'knights', 'spam', 'eggs'}
```

update(iterable) : Met à jour le set avec les éléments contenus dans l'itérable.

```
s = {'spam', 'eggs', 'knights'}
s.update(['ham', 'lumberjack', 'bones'])
print(s)    # {'ham', 'bones', 'lumberjack', 'spam', 'eggs', 'knights'}
```

Retirer certains éléments d'un set - On peut retirer les éléments d'un set grâce aux méthodes discard et remove.

La méthode discard ne modifie pas le set si l'élément à supprimer n'est pas présent dans le set à l'inverse de remove qui lèvera une exception :

```
s = {'ham', 'bones', 'lumberjack', 'spam', 'eggs', 'knights'}
print(s) # {'ham', 'bones', 'lumberjack', 'spam', 'eggs', 'knights'}

s.discard('bones')
print(s) # {'ham', 'lumberjack', 'spam', 'eggs', 'knights'}
```

```
s.remove('eggs')
print(s) # {'ham', 'lumberjack', 'spam', 'knights'}

s.discard('anything')
print(s) # {'ham', 'lumberjack', 'spam', 'knights'}

s.remove('anything') # KeyError: 'anything'
```

clear() : Vous pouvez aussi vider complètement un set avec la méthode clear.

```
s = {'ham', 'bones', 'lumberjack', 'spam', 'eggs', 'knights'}
s.clear()
print(s)     # set()
```

NB :Même si le set a été créé en utilisant les accolades, la méthode clear() retourne set() et non {} (car des accolades vides correspondent à un dictionnaire).

union(e) : Crée un ensemble représentant l'union entre deux ou plusieurs ensembles. Une union entre A et B crée un set avec tous les éléments de A et de B.

Pour créer cette union, on utilise le symbole |(on peut aussi utiliser la méthode union :

```
A = {1, 2, 3}
B = {2, 3, 4}

print(A | B)       # {1, 2, 3, 4}
print(A.union(B))  # {1, 2, 3, 4}
```

intersection(e) : Crée un ensemble représentant l'intersection entre deux ou plusieurs ensembles. Une intersection entre A et B crée un set avec tous les éléments communs de A et de B.

Pour créer cette intersection, on utilise le symbole & (on peut aussi utiliser la méthode intersection) :

```
A = {3, 4, 5, 6, 7}
B = {6, 7, 8, 9, 10}

print(A & B)              # {6, 7}
print(A.intersection(B))  # {6, 7}
```

difference(e) : Crée un ensemble représentant la différence entre deux ou plusieurs ensembles. Une différence de A sur B crée un set avec tous les éléments uniquement compris dans A mais pas dans B.

Pour créer cette différence, on utilise le symbole -(on peut aussi utiliser la méthode difference)

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

print(A - B)            # {1, 2, 3}
print(A.difference(B))  # {1, 2, 3}
```

symmetric_difference(e) : Crée un ensemble contenant les éléments présents dans un des ensembles mais pas dans les deux. Une **différence symétrique** entre A et B crée un set avec tous les éléments compris dans A et dans B mais pas dans les deux à la fois.

Pour créer cette différence, on utilise le symbole

(on peut aussi utiliser la méthode symmetric_difference) :

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

print(A ^ B)                      # {1, 2, 3, 6, 7, 8}
print(A.symmetric_difference(B))  # {1, 2, 3, 6, 7, 8}
```

Les "set comprehension"

Comme pour les *listes*, on peut construire un *set* de façon concise avec un set comprehension

```
mon_set = {x*y for x in {1,2,3} for y in {2,3,4} if x != y}
print(mon_set)  # {2, 3, 4, 6, 8, 12}
```

Les Frozenset

Les *frozenset* ont exactement les mêmes caractéristiques qu'un *set* classique à la différence qu'**ils ne peuvent plus être modifiés une fois créés**.

On les définit en utilisant la fonction *frozenset()* :

```
s = frozenset(['bones', 'lumberjack', 'spam', 'eggs', 'knights'])

print(s)              # frozenset({'knights', 'spam', 'lumberjack', 'bones',
↪  'eggs'})
print(len(s))         # 5
print(s.add('bones'))  # AttributeError: 'frozenset' object has no attribute
↪  'add'
```

Ensemble, liste et tuple

Les ensembles, listes et tuples sont des collections d'éléments, il est possible de passer de l'un à l'autre grâce aux méthode set, list, tuple.

```
ma_liste = [1, 1, 2, 2]
print(type(ma_liste))
mon_ensemble = set(ma_liste)
print(type(mon_ensemble))
mon_tuple = tuple(mon_ensemble)
print(type(mon_tuple))
```

Astuce

Comme un ensemble n'est pas ordonné, si vous créez une liste ou un tuple à partir d'un ensemble, vous n'avez **aucune garantie sur l'ordre des éléments** dans la liste ou le tuple qui sera créé.
De la même manière, si vous parcourez un ensemble avec une instruction for, vous n'avez aucune garantie sur l'ordre de parcours des éléments. Dit autrement, un ensemble n'est pas une structure de données recommandée quand il existe une relation d'ordre entre les éléments.

3.3.4 Les dictionnaires

Les dictionnaires sont des collections d'**objets non-ordonnées**. Un dictionnaire est composé d'éléments et chaque élément se compose d'une paire ***clé : valeur***.

Dans d'autres langages de programmation, on parle de **tableaux associatifs** ou de **hashs**.

Comme les listes, les dictionnaires sont des **objets muables** et **dynamiques**. Ils peuvent être modifiés et s'étendre selon vos besoins.

Un dictionnaire peut contenir des objets de n'importe quel type et même inclure d'autres dictionnaires.C'est grâce à ces caractéristiques que les dictionnaires sont souvent utilisés pour créer des structures de données complexes et polyvalentes où plusieurs éléments sont imbriqués les uns dans les autres ! ils offrent un moyen flexible de structurer et d'accéder aux données. Les dictionnaires sont particulièrement adaptés lorsque vous avez besoin d'une organisation facile d'accès, car chaque clé est associée à une valeur unique. Cette particularité en fait un choix privilégié pour de nombreuses applications, que ce soit pour le stockage de configurations, la gestion de bases de données ou la manipulation d'objets complexes.

Nous allons tour à tour exposer l'ensemble des opérations que vous pourrez effectuer sur et avec les dictionnaires.

Création d'un dictionnaire

La façon la plus simple de créer un dictionnaire est d'ouvrir des accolades *{}* et d'y insérer des paires de clés et de valeurs.

Pour écrire une paire, il faut respecter la syntaxe suivante : *clé : valeur*.

Chaque paire doit être séparée de l'autre par une virgule.

Les valeurs peuvent être de n'importe quel type alors que les clés doivent obligatoirement être de type immuable (2.2.1 : comme une chaîne ou un entier) !

```
# Création d'un dictionnaire simple
personne = {
    "nom": "Alice",
    "âge": 30,
    "ville": "Paris"
}
```

Dans cet exemple, "nom", "âge" et "ville" sont les **clés** et leurs **valeurs respectives** sont "Alice", 30 et "Paris". Vous pouvez utiliser des types variés pour les valeurs, comme des listes, d'autres dictionnaires, ou même des tuples si nécessaire.

Vous pouvez aussi créer un dictionnaire vide et y ajouter des éléments par la suite :

```
# Dictionnaire vide
dictionnaire_vide = {}

# Ajout de paires clé-valeur
dictionnaire_vide["clé1"] = "valeur1"
dictionnaire_vide["clé2"] = "valeur2"
```

Vous pouvez aussi créer un dictionnaire grâce la fonction *dict* :

```
d = dict() # {}

d = dict({
    'spam': 'eggs',
    'knights': 'lumberjack',
    'bacon': 'sausage'
})
print(d)
```

Une clé doit être unique, les doublons ne sont pas autorisés. Si cela arrive, cela ne va pas créer d'erreur mais votre seconde clé écrasera la première.

```
d = {
        'spam': 'eggs',
        'knights': 'lumberjack', # 1ère clé 'knights'
        'bacon': 'sausage',
        'knights': 'ham' # 2ème clé 'knights' qui va écraser la première
}

print(d)
```

Ne n'oubliez pas, une clé ne peut pas être un objet muable :

```
d = {
    ['spam']: 'eggs' # TypeError: unhashable type: 'list'
}
```

Accéder aux valeurs dans un dictionnaire

Alors qu'on accède aux éléments contenus dans une liste ou un tuple grâce à leurs indices (car ce sont des structures ordonnés), pour les dictionnaires, une fois un dictionnaire créé, accéder à ses valeurs est très simple. Il suffit d'utiliser la **clé** correspondante entre crochets

pour obtenir la valeur associée.

Exemple d'accès à une valeur :

```
personne = {
    "nom": "Alice",
    "âge": 30,
    "ville": "Paris"
}

# Accéder à la valeur de la clé "nom"
nom_personne = personne["nom"]
print(nom_personne)  # Affiche : Alice
```

Dans cet exemple, en utilisant la clé "nom", j'obtiens la valeur "Alice" associée à cette clé.

Gérer l'absence de clé avec get()

Si vous essayez d'accéder à une clé qui n'existe pas, Python renvoie une erreur. Pour éviter cela, vous pouvez utiliser la méthode *get()*, qui retourne None si la clé n'est pas trouvée, ou une valeur par défaut que vous spécifiez.

Exemple avec *get()* :

```
# Utilisation de get pour une clé qui n'existe pas
profession = personne.get("profession", "Inconnue")
print(profession)  # Affiche : Inconnue
```

Ici, comme la clé "profession" n'existe pas dans le dictionnaire, *get()* renvoie "Inconnue" au lieu de générer une erreur. Cette méthode est particulièrement utile pour éviter des exceptions et pour rendre votre code plus robuste lorsqu'il s'agit d'accéder à des valeurs dans un dictionnaire.

Ajouter des éléments

Les dictionnaires en Python sont des structures dynamiques, ce qui signifie que vous pouvez facilement y ajouter de nouvelles paires clé-valeur ou modifier des valeurs existantes.

Ajouter un nouvel élément : Pour ajouter un nouvel élément dans un dictionnaire, il suffit de spécifier une nouvelle clé entre crochets [] et d'y assigner une valeur. Voici un exemple d'ajout :

```
personne = {
    "nom": "Alice",
    "âge": 30
}

# Ajout d'une nouvelle clé-valeur
personne["ville"] = "Paris"
print(personne)  # Affiche : {'nom': 'Alice', 'âge': 30, 'ville': 'Paris'}
```

Dans cet exemple, j'ajoute une nouvelle entrée "ville" avec la valeur "Paris" dans le dictionnaire personne.

Modifier un élément existant

Pour modifier la valeur d'une clé existante, il suffit de la réassigner avec la nouvelle valeur souhaitée.

Exemple de modification :

```
# Modification de la valeur de la clé "âge"
personne["âge"] = 31
print(personne)  # Affiche : {'nom': 'Alice', 'âge': 31, 'ville': 'Paris'}
```

Ici, j'ai mis à jour la valeur associée à la clé "âge" en la remplaçant par 31. Avec cette flexibilité, vous pouvez adapter le contenu de vos dictionnaires au fur et à mesure que les données évoluent dans votre programme.

Supprimer des éléments d'un dictionnaire

Python offre plusieurs méthodes pour supprimer des éléments d'un dictionnaire, qu'il s'agisse de supprimer une clé spécifique ou de vider complètement le dictionnaire.

Supprimer un élément avec pop() : La méthode pop() permet de supprimer une clé spécifique et de renvoyer sa valeur. Si la clé n'existe pas, une erreur est générée, sauf si vous spécifiez une valeur par défaut.

Exemple d'utilisation de pop() :

```
personne = {
    "nom": "Alice",
    "âge": 30,
    "ville": "Paris"
}

# Suppression de la clé "ville"
ville = personne.pop("ville")
```

```
print(ville)        # Affiche : Paris
print(personne)     # Affiche : {'nom': 'Alice', 'âge': 30}
```

Dans cet exemple, j'ai supprimé la clé "ville" du dictionnaire et pop() a renvoyé la valeur "Paris".

Supprimer un élément avec del : Vous pouvez aussi utiliser l'instruction del pour supprimer un élément spécifique. Contrairement à pop(), del ne renvoie pas la valeur supprimée.

Exemple d'utilisation de del :

```
# Suppression de la clé "âge"
del personne["âge"]
print(personne)     # Affiche : {'nom': 'Alice'}
```

Ici, del a supprimé la clé "âge" et sa valeur associée du dictionnaire.

Vider complètement un dictionnaire avec clear() : Si vous souhaitez supprimer tous les éléments d'un dictionnaire, utilisez clear(). Cela vide le dictionnaire sans le supprimer lui-même.

Exemple d'utilisation de clear() :

```
# Vider le dictionnaire
personne.clear()
print(personne)     # Affiche : {}
```

Le dictionnaire personne est maintenant vide, prêt à être réutilisé ou rempli à nouveau.

Méthodes courantes sur les dictionnaires

Les dictionnaires Python disposent de plusieurs méthodes pratiques pour manipuler et explorer leur contenu. Voici quelques-unes des méthodes les plus utiles que vous utiliserez souvent.

Obtenir les clés avec keys() : La méthode keys() renvoie un objet contenant toutes les clés du dictionnaire, ce qui est pratique pour parcourir ou vérifier la présence d'une clé.

Exemple d'utilisation de keys() :

```
personne = {
    "nom": "Alice",
    "âge": 30,
    "ville": "Paris"
}

# Obtenir les clés
cles = personne.keys()
print(cles)     # Affiche : dict_keys(['nom', 'âge', 'ville'])
```

Obtenir les valeurs avec values() : La méthode values() retourne toutes les valeurs du dictionnaire. Vous pouvez l'utiliser pour parcourir uniquement les valeurs.

Exemple d'utilisation de values() :

```
# Obtenir les valeurs
valeurs = personne.values()
print(valeurs)  # Affiche : dict_values(['Alice', 30, 'Paris'])
```

Obtenir les paires clé-valeur avec items() : La méthode items() renvoie une vue contenant des paires *clé-valeur* sous forme de tuples. Cette méthode est particulièrement utile pour parcourir un dictionnaire en obtenant à la fois les clés et les valeurs.

Exemple d'utilisation de items() :

```
# Obtenir les paires clé-valeur
paires = personne.items()
print(paires)   # Affiche : dict_items([('nom', 'Alice'), ('âge', 30),
↪  ('ville', 'Paris')])
```

Exemple de parcours avec items() : Vous pouvez utiliser items() dans une boucle pour accéder aux clés et valeurs en même temps.

```
for cle, valeur in personne.items():
    print(f"{cle} : {valeur}")
# Affiche :
# nom : Alice
# âge : 30
# ville : Paris
```

Ces méthodes vous permettent de manipuler efficacement le contenu d'un **dictionnaire** et d'accéder facilement à ses différentes parties selon vos besoins.

3.3.5 Compréhension de dictionnaire

La **compréhension de dictionnaire** est une méthode concise pour créer et manipuler des dictionnaires en Python. Inspirée des compréhensions de listes, cette syntaxe permet de générer un dictionnaire en une seule ligne, en appliquant des filtres ou transformations aux clés et valeurs.

Syntaxe de base : La syntaxe d'une compréhension de dictionnaire est similaire à celle des listes, mais utilise des accolades {} et suit le format *clé : valeur for élément in iterable*.

Exemple de compréhension de dictionnaire simple :

```
# Créer un dictionnaire de carrés
carres = {x: x**2 for x in range(1, 6)}
print(carres)   # Affiche : {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Dans cet exemple, j'ai créé un dictionnaire où chaque nombre de 1 à 5 est une **clé** associée à son carré comme **valeur**.

Filtrer les éléments avec une condition : Vous pouvez ajouter une condition pour filtrer les éléments à inclure dans le dictionnaire.

Exemple de compréhension avec condition :

```
# Dictionnaire de carrés pour les nombres pairs seulement
carres_pairs = {x: x**2 for x in range(1, 6) if x \% 2 == 0}
print(carres_pairs)  # Affiche : {2: 4, 4: 16}
```

Ici, seules les clés représentant des nombres pairs sont ajoutées au dictionnaire, car j'ai appliqué une condition *if x % 2 == 0*.

Transformer les valeurs : La compréhension de dictionnaire permet aussi de transformer les **valeurs** tout en gardant les mêmes **clés**.

Exemple de transformation des valeurs :

```
personne = {
    "nom": "Alice",
    "âge": 30,
    "ville": "Paris"
}

# Transformation : ajouter "!" à chaque valeur
personne_maj = {cle: str(valeur) + "!" for cle, valeur in personne.items()}
print(personne_maj)  # Affiche : {'nom': 'Alice!', 'âge': '30!', 'ville':
↪  'Paris!'}
```

Dans cet exemple, chaque **valeur** est transformée en chaîne de caractères avec un point d'exclamation ajouté à la fin.

Dictionnaires imbriqués

Un **dictionnaire imbriqué** est un dictionnaire qui contient d'autres **dictionnaires** en tant que valeurs. Cette structure est idéale pour stocker des données complexes et hiérarchiques, comme des informations sur des objets ayant plusieurs attributs ou sous-attributs.

Création d'un dictionnaire imbriqué : Vous pouvez créer un dictionnaire imbriqué en assignant un dictionnaire comme valeur à une clé.

Exemple de dictionnaire imbriqué :

```
employes = {
    "employe1": {"nom": "Alice", "âge": 30, "poste": "Développeuse"},
```

```
    "employe2": {"nom": "Bob", "âge": 25, "poste": "Designer"},
    "employe3": {"nom": "Charlie", "âge": 35, "poste": "Manager"}
}

print(employes)
# Affiche :
# {
#   'employe1': {'nom': 'Alice', 'âge': 30, 'poste': 'Développeuse'},
#   'employe2': {'nom': 'Bob', 'âge': 25, 'poste': 'Designer'},
#   'employe3': {'nom': 'Charlie', 'âge': 35, 'poste': 'Manager'}
# }
```

Dans cet exemple, chaque employé est une clé dans le dictionnaire employes et sa valeur est un autre dictionnaire contenant ses attributs.

Accéder aux éléments dans un dictionnaire imbriqué : Pour accéder aux éléments d'un dictionnaire imbriqué, utilisez plusieurs paires de crochets

pour descendre dans les niveaux de la structure.

Exemple d'accès aux données :

```
# Accéder au poste de "employe2"
poste_employe2 = employes["employe2"]["poste"]
print(poste_employe2)  # Affiche : Designer
```

Ici, j'accède d'abord à la clé "employe2", puis à la clé "poste" à l'intérieur du sous-dictionnaire correspondant.

Modifier un élément dans un dictionnaire imbriqué : Vous pouvez modifier des valeurs dans un dictionnaire imbriqué de la même manière que dans un dictionnaire simple.

Exemple de modification :

```
# Changer le poste de "employe3"
employes["employe3"]["poste"] = "Directeur"
print(employes["employe3"])  # Affiche : {'nom': 'Charlie', 'âge': 35,
↪  'poste': 'Directeur'}
```

Parcourir un dictionnaire imbriqué : Pour parcourir un dictionnaire imbriqué, utilisez une boucle for et items() pour accéder aux clés et sous-dictionnaires, puis une autre boucle pour parcourir les éléments du sous-dictionnaire.

Exemple de parcours :

```
for employe, details in employes.items():
    print(f"Détails de {employe} :")
```

```
    for cle, valeur in details.items():
        print(f"  {cle} : {valeur}")
# Affiche :
# Détails de employe1 :
#   nom : Alice
#   âge : 30
#   poste : Développeuse
# Détails de employe2 :
#   nom : Bob
#   âge : 25
#   poste : Designer
# Détails de employe3 :
#   nom : Charlie
#   âge : 35
#   poste : Directeur
```

Manipuler des dictionnaires avec des fonctions avancées

Python propose plusieurs fonctions intégrées qui facilitent les manipulations avancées avec les **dictionnaires**, permettant de créer, fusionner et transformer des dictionnaires efficacement.

Utiliser dict() pour créer un dictionnaire : La fonction dict() permet de créer un dictionnaire à partir de paires **clé-valeur** sous forme de tuples, ou d'un autre type de données compatible.

Exemple avec dict() :

```
# Création d'un dictionnaire avec des tuples
infos_personne = dict([("nom", "Alice"), ("âge", 30), ("ville", "Paris")])
print(infos_personne)  # Affiche : {'nom': 'Alice', 'âge': 30, 'ville':
↪  'Paris'}
```

Fusionner des dictionnaires avec update() : La méthode update() permet de fusionner deux dictionnaires en mettant à jour le premier avec les clés et valeurs du second. Si des clés existent déjà, leurs valeurs seront écrasées.

Exemple d'utilisation de update() :

```
personne = {"nom": "Alice", "âge": 30}
autres_infos = {"ville": "Paris", "âge": 31}  # Âge mis à jour

personne.update(autres_infos)
print(personne)  # Affiche : {'nom': 'Alice', 'âge': 31, 'ville': 'Paris'}
```

Ici, la valeur de "âge" a été mise à jour avec celle de autres_infos et la clé "ville" a été ajoutée.

Créer un dictionnaire avec zip() : La fonction zip() est pratique pour créer un dictionnaire à partir de deux listes : l'une pour les clés, l'autre pour les valeurs.

Exemple de création avec zip() :

```
cles = ["nom", "âge", "ville"]
valeurs = ["Alice", 30, "Paris"]

personne = dict(zip(cles, valeurs))
print(personne)  # Affiche : {'nom': 'Alice', 'âge': 30, 'ville': 'Paris'}
```

Fusionner plusieurs dictionnaires avec ** Vous pouvez également utiliser l'opérateur ** pour fusionner plusieurs dictionnaires en un seul.

Exemple de fusion avec ** :

```
dictionnaire1 = {"nom": "Alice", "âge": 30}
dictionnaire2 = {"ville": "Paris", "poste": "Développeuse"}

personne_complete = {**dictionnaire1, **dictionnaire2}
print(personne_complete)
# Affiche : {'nom': 'Alice', 'âge': 30, 'ville': 'Paris', 'poste':
↪  'Développeuse'}
```

Utiliser des expressions générées par dictionnaire Vous pouvez aussi créer un dictionnaire en utilisant des compréhensions et des fonctions ensemble.

Exemple de transformation de valeurs avec une fonction :

```
# Multiplication par 2 des valeurs dans un dictionnaire de nombres
nombres = {"a": 1, "b": 2, "c": 3}
nombres_double = {k: v * 2 for k, v in nombres.items()}
print(nombres_double)  # Affiche : {'a': 2, 'b': 4, 'c': 6}
```

Ces fonctions avancées permettent de manipuler les dictionnaires de manière flexible et efficace, que ce soit pour des opérations simples ou des transformations plus complexes.

TP sur les dictionnaires

Dans ces travaux pratiques, vous allez construire pas à pas un programme en Python pour gérer une bibliothèque de livres. L'objectif est de pratiquer les manipulations de dictionnaires : *ajout, modification, suppression* et *recherche de données*. Vous allez aussi pratiquer l'usage des méthodes clés des dictionnaires pour explorer et manipuler ces données.

Scénario : Vous êtes responsable de la gestion des informations de la bibliothèque d'une petite école. Vous devez créer un programme pour stocker les informations de chaque livre, telles que le **titre**, l'**auteur**, l'**année** de publication, et le **genre**. Vous allez suivre chaque étape pour construire votre programme.

Étapes des TP

1. **Etape 1 : Créer un dictionnaire pour un livre -** Créez un dictionnaire pour un livre avec les informations suivantes : titre, auteur, année de publication et genre. Remplissez-le avec un exemple de votre choix.

   Solution :

```
# Exemple de dictionnaire pour un livre
livre1 = {
    "titre": "Le Petit Prince",
    "auteur": "Antoine de Saint-Exupéry",
    "annee": 1943,
    "genre": "Fiction"
}
```

2. **Étape 2 : Créer une bibliothèque sous forme de dictionnaire**

   — Créez un dictionnaire vide appelé *bibliotheque* pour stocker plusieurs livres.
   — Ajoutez le livre que vous avez créé dans la première étape en lui assignant un identifiant unique (comme "*Livre1*").

   Solution :

```
# Création de la bibliothèque et ajout du premier livre
bibliotheque = {}
bibliotheque["Livre1"] = livre1
```

3. **Étape 3 : Ajouter plusieurs livres à la bibliothèque**

   — Créez au moins deux autres livres sous forme de dictionnaires avec les mêmes informations (titre, auteur, année, genre).
   — Ajoutez-les à *bibliotheque* en utilisant des identifiants uniques (comme "*Livre2*", "*Livre3*", etc.).

   Solution :

```
# Exemple de deux autres livres
livre2 = {
    "titre": "1984",
    "auteur": "George Orwell",
    "annee": 1949,
    "genre": "Dystopie"
}

livre3 = {
    "titre": "Moby Dick",
    "auteur": "Herman Melville",
    "annee": 1851,
    "genre": "Aventure"
}
```

```
# Ajout des livres à la bibliothèque
bibliotheque["Livre2"] = livre2
bibliotheque["Livre3"] = livre3
```

4. **Étape 4 : Afficher les informations de tous les livres -** Parcourez le dictionnaire bibliotheque pour afficher les informations de chaque livre. Utilisez une boucle for pour afficher chaque livre sous la forme :

```
Livre1:
Titre : [titre]
Auteur : [auteur]
Année : [année]
Genre : [genre]
```

Solution :

```
# Parcourir et afficher les informations de tous les livres
for identifiant, livre in bibliotheque.items():
    print(f"{identifiant}:")
    print(f"  Titre : {livre['titre']}")
    print(f"  Auteur : {livre['auteur']}")
    print(f"  Année : {livre['annee']}")
    print(f"  Genre : {livre['genre']}")
```

5. **Étape 5 : Rechercher un livre par titre**

   — Demandez à l'utilisateur de saisir un titre de livre.
   — Parcourez *bibliotheque* pour vérifier si un livre avec ce titre existe.
   — Si le livre est trouvé, affichez toutes ses informations. Sinon, affichez un message indiquant que le livre n'a pas été trouvé.

Solution :

```
# Parcourir et afficher les informations de tous les livres
for identifiant, livre in bibliotheque.items():
    print(f"{identifiant}:")
    print(f"  Titre : {livre['titre']}")
    print(f"  Auteur : {livre['auteur']}")
    print(f"  Année : {livre['annee']}")
    print(f"  Genre : {livre['genre']}")
```

6. **Étape 6 : Modifier les informations d'un livre**

   — Demandez à l'utilisateur de saisir l'identifiant d'un livre (par exemple, "*Livre1*").
   — Demandez-lui ensuite quelle information il souhaite modifier (titre, auteur, année ou genre) et la nouvelle valeur.
   — Mettez à jour le dictionnaire *bibliotheque* avec cette nouvelle information.

Solution :

```python
# Modification des informations d'un livre
identifiant = input("Entrez l'identifiant du livre à modifier (par exemple, 'Livre1') : ")
if identifiant in bibliotheque:
    champ = input("Quel champ souhaitez-vous modifier ? (titre, auteur, annee, genre) : ")
    nouvelle_valeur = input(f"Entrez la nouvelle valeur pour {champ} : ")
    if champ in bibliotheque[identifiant]:
        bibliotheque[identifiant][champ] = nouvelle_valeur
        print(f"Le champ {champ} a été mis à jour.")
    else:
        print("Champ non valide.")
else:
    print("Identifiant non trouvé.")
```

7. **Étape 7 : Supprimer un livre de la bibliothèque**

   — Demandez à l'utilisateur de saisir l'identifiant d'un livre à supprimer (par exemple, "Livre2").
   — Supprimez le livre correspondant du dictionnaire *bibliotheque*.

   Solution :

```python
# Suppression d'un livre
identifiant = input("Entrez l'identifiant du livre à supprimer (par exemple, 'Livre2') : ")
if identifiant in bibliotheque:
    bibliotheque.pop(identifiant)
    print(f"{identifiant} a été supprimé de la bibliothèque.")
else:
    print("Livre non trouvé.")
```

8. **Étape 8 : Afficher les livres d'un genre spécifique**

   — Demandez à l'utilisateur de saisir un genre (par exemple, "Science-fiction").
   — Parcourez *bibliotheque* et affichez les informations de tous les livres appartenant à ce genre.

   Solution :

```python
# Afficher les livres par genre
genre_recherche = input("Entrez le genre des livres à afficher : ")
for identifiant, livre in bibliotheque.items():
    if livre["genre"].lower() == genre_recherche.lower():
        print(f"{identifiant}: {livre['titre']} par {livre['auteur']} ({livre['annee']})")
```

Bonnes pratiques pour travailler avec les dictionnaires en Python

Les **dictionnaires** sont des structures très puissantes en Python, mais pour les utiliser efficacement et éviter certains écueils, il est important de suivre quelques bonnes pratiques. Ces conseils vous aideront à écrire du code Python plus lisible, performant et moins sujet aux erreurs.

Utilisez des clés immuables et descriptives : Les clés dans un dictionnaire doivent être immuables, comme des chaînes de caractères, des entiers, ou des tuples (si les éléments du tuple sont eux-mêmes immuables). Par ailleurs, choisissez des noms de clés significatifs qui rendent votre code lisible et compréhensible. Par exemple, préférez "nom" plutôt que "n" pour stocker un nom.

Exemple :

```
# Bonnes pratiques pour les clés
profil_utilisateur = {
    "nom": "Alice",
    "âge": 30,
    "email": "alice@example.com"
}
```

Préférez get() pour accéder aux valeurs lorsque les clés peuvent être absentes : Plutôt que d'accéder directement à une clé avec

, utilisez la méthode get() lorsque vous n'êtes pas sûr que la clé existe dans le dictionnaire. get() vous permet d'éviter une erreur KeyError et de fournir une valeur par défaut si la clé est absente.

Exemple :

```
# Accès sécurisé avec get()
age = profil_utilisateur.get("âge", "Non spécifié")
```

Utilisez des méthodes de parcours adaptées au contexte : Python propose plusieurs façons de parcourir les dictionnaires. En fonction de votre besoin (seulement les clés, seulement les valeurs, ou les deux), choisissez la méthode la plus adaptée.

- Utilisez keys() si vous ne voulez parcourir que les clés.
- Utilisez values() pour parcourir uniquement les valeurs.
- Utilisez items() pour parcourir les paires clé-valeur.

Cela rend votre code plus lisible et exprime votre intention plus clairement.

Exemple :

```
# Parcours avec items() pour accéder à la fois aux clés et valeurs
for cle, valeur in profil_utilisateur.items():
    print(f"{cle} : {valeur}")
```

Évitez d'imbriquer trop de niveaux de dictionnaires : Bien que les dictionnaires imbriqués soient utiles, une structure trop complexe devient rapidement difficile à lire et à maintenir. Si un dictionnaire contient plusieurs niveaux d'imbrication, envisagez de simplifier la structure ou de regrouper les données en utilisant des classes ou des objets pour organiser les informations de manière plus claire.

Exemple :

Au lieu de ceci :

```
# Dictionnaire avec plusieurs niveaux d'imbrication
societe = {
    "service_IT": {
        "employes": {
            "Alice": {"poste": "Développeuse"},
            "Bob": {"poste": "Administrateur"}
        }
    }
}
```

Envisagez une structure plus plate ou l'utilisation d'objets.

Utilisez des compréhensions de dictionnaire pour des transformations simples : Les compréhensions de dictionnaire sont idéales pour créer de nouveaux dictionnaires en appliquant des transformations aux données. Elles sont souvent plus efficaces et lisibles que les boucles traditionnelles, à condition que l'opération reste simple.

Exemple :

```
# Double les valeurs dans un dictionnaire de notes
notes = {"Alice": 10, "Bob": 8, "Charlie": 9}
notes_double = {nom: note * 2 for nom, note in notes.items()}
```

Videz un dictionnaire avec clear() plutôt que de recréer une nouvelle variable : Si vous devez vider un dictionnaire existant sans créer une nouvelle variable, utilisez clear(). Cela peut être plus performant, surtout si d'autres références pointent vers le même dictionnaire.

Exemple :

```
# Vider un dictionnaire
profil_utilisateur.clear()
```

Évitez les modifications de dictionnaire pendant un parcours : Modifier un dictionnaire tout en le parcourant peut provoquer des erreurs et des comportements imprévisibles. Si vous avez besoin de modifier un dictionnaire pendant que vous le parcourez, envisagez d'abord de créer une copie avec .copy() ou d'accumuler les changements dans un dictionnaire séparé.

Exemple :

```
# Parcours sécurisé avec copy()
for cle in list(profil_utilisateur.keys()):
    if cle == "email":
        profil_utilisateur.pop(cle)
```

Commentez les structures complexes : Lorsque vous utilisez des dictionnaires avec des structures complexes ou imbriquées, commentez votre code pour que vous (ou d'autres développeurs) puissiez comprendre la structure et l'intention derrière chaque clé. Cela facilite la maintenance et la lisibilité.

Exemple :

```
# Structure d'un employé avec des informations imbriquées
employe = {
    "nom": "Alice",
    "poste": "Développeuse",
    # Informations de contact
    "contact": {
        "email": "alice@example.com",
        "telephone": "123456789"
    }
}
```

Testez l'existence d'une clé avant de modifier ses valeurs : Si vous devez mettre à jour la valeur d'une clé, vérifiez d'abord son existence avec in pour éviter des erreurs KeyError. Cela est particulièrement utile dans des applications où les données proviennent de sources externes (fichiers, API, etc.).

Exemple :

```
# Structure d'un employé avec des informations imbriquées
employe = {
    "nom": "Alice",
    "poste": "Développeuse",
    # Informations de contact
    "contact": {
        "email": "alice@example.com",
        "telephone": "123456789"
    }
}
```

En suivant ces bonnes pratiques, vous vous assurez de tirer le meilleur parti des dictionnaires Python tout en produisant du code lisible, maintenable, et optimisé. Les dictionnaires sont des structures extrêmement puissantes, et maîtriser leur utilisation est essentiel pour écrire des programmes Python performants et robustes.

En conclusion sur les dictionnaires

Les **dictionnaires** avec les **listes** est des types essentiels pour organiser et manipuler des données associatives de manière efficace. Grâce à leur structure en paires **clé-valeur**, ils permettent un accès rapide aux informations, ce qui est précieux pour de nombreuses applications, comme le stockage de configurations, la gestion de données complexes, ou encore le traitement d'objets imbriqués.

En maîtrisant les **dictionnaires** et leurs multiples fonctionnalités, vous disposez d'un atout puissant pour structurer efficacement vos programmes et manipuler vos données de manière fluide et optimisée. N'hésitez pas à expérimenter avec ces techniques et à les adapter selon vos besoins pour tirer le meilleur parti des dictionnaires dans vos projets.

3.3.6 Les chaînes

Nous avons déjà abordé les chaînes de caractères en python. Nous vous renvoyons à cette section pour revisiter toutes les opérations possibles avec les chaînes de caractères en python ici (2.2.5).

3.3.7 Les tableaux

Nous aborderons les tableaux quand sera venu le temps de bosser avec la module *numpy* pour l'analyse des données avec python.

3.3.8 Maîtriser le slicing en Python

Le slicing en Python est une technique incontournable pour extraire, modifier ou analyser des séquences comme les listes, tuples et chaînes de caractères. Grâce à une syntaxe simple basée sur des indices *(start :stop :step)*, il permet de manipuler efficacement des données sans boucles complexes.

Que ce soit pour extraire une sous-liste, inverser une chaîne de caractères ou récupérer des éléments spécifiques, le **slicing** est un outil puissant et intuitif. Je vais vous montrer comment l'utiliser efficacement avec des exemples clairs et des astuces pratiques.

Syntaxe du slicing en Python

Le slicing en Python repose sur une notation simple et efficace qui permet d'extraire une portion d'une liste, d'un tuple ou d'une chaîne de caractères sans avoir besoin de boucles complexes.

L'opérateur de **slicing** s'écrit sous la forme :

```
sequence[start:stop:step]
```

- start : indice de départ (inclus)

- stop : indice de fin (exclu)
- step : pas de saut entre les éléments

Si un des paramètres est omis, **Python** utilise ses valeurs par défaut :

```
sequence[:stop]      # Commence à 0 et s'arrête à stop (exclu)
sequence[start:]     # Commence à start et va jusqu'à la fin
sequence[::step]     # Prend tout avec un pas spécifique
sequence[:]          # Copie complète de la séquence
```

Exemples simples :

1. Slicing sur une liste

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[2:5])   # [2, 3, 4]
print(numbers[:4])    # [0, 1, 2, 3]
print(numbers[5:])    # [5, 6, 7, 8, 9]
print(numbers[::2])   # [0, 2, 4, 6, 8] (un élément sur deux)
```

2. Slicing sur une chaîne de caractères

```
text = "Python slicing"

print(text[0:6])   # 'Python'
print(text[:6])    # 'Python'
print(text[7:])    # 'slicing'
print(text[::2])   # 'Pto lcig'
```

Comprendre les indices négatifs

Python permet d'utiliser des indices négatifs pour compter à partir de la fin de la séquence :

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[-3:])   # [7, 8, 9]
print(numbers[:-3])   # [0, 1, 2, 3, 4, 5, 6]
print(numbers[::-1])  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (inversion)
```

Exemple avec une chaîne de caractères :

```
text = "Python slicing"

print(text[-7:])   # 'slicing'
print(text[:-7])   # 'Python '
print(text[::-1])  # 'gnicils nohtyP' (inversion)
```

Le **slicing** est donc un moyen très puissant pour **extraire**, **transformer** et **manipuler** les séquences en **Python**. Voyons maintenant comment l'appliquer spécifiquement aux **listes**.

Slicing sur les listes

Le slicing est particulièrement utile avec les listes, car il permet d'extraire des sous-listes, d'accéder aux éléments de manière flexible et même de modifier leur contenu.

Extraire une sous-liste : La technique la plus courante consiste à récupérer une **portion** de la liste en utilisant la notation *start :stop*.

```
fruits = ["pomme", "banane", "cerise", "datte", "figue", "kiwi"]

print(fruits[1:4])  # ['banane', 'cerise', 'datte']
print(fruits[:3])   # ['pomme', 'banane', 'cerise']
print(fruits[3:])   # ['datte', 'figue', 'kiwi']
```

Dans chaque cas :

— L'élément *start* est inclus.
— L'élément *stop* est exclu.

Utiliser des indices négatifs : Python permet d'utiliser des **indices négatifs** pour compter depuis la fin de la liste.

```
print(fruits[-3:])   # ['datte', 'figue', 'kiwi']
print(fruits[:-3])   # ['pomme', 'banane', 'cerise']
```

Astuce : L'indice *-1* représente le dernier élément, *-2* l'avant-dernier, etc.

Slicing avec un pas (step) : L'argument *step* permet de sauter des éléments dans la liste.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[::2])  # [0, 2, 4, 6, 8]  (un élément sur deux)
print(numbers[1::2]) # [1, 3, 5, 7, 9]  (éléments impairs)
```

On peut aussi utiliser un **pas négatif** pour parcourir la liste à l'envers.

```
print(numbers[::-1])  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
print(numbers[::-2])  # [9, 7, 5, 3, 1]  (un élément sur deux en inversé)
```

Modifier une liste avec le slicing : Contrairement aux chaînes de caractères qui sont **immuables**, on peut modifier une liste en utilisant le **slicing**.

```
letters = ["a", "b", "c", "d", "e"]

letters[1:3] = ["x", "y"]  # Remplace ['b', 'c'] par ['x', 'y']
print(letters)  # ['a', 'x', 'y', 'd', 'e']
```

On peut aussi supprimer plusieurs éléments en leur attribuant une liste vide.

```
letters[2:4] = []
print(letters)  # ['a', 'x', 'e']
```

Cas particulier : inverser une liste - Une astuce très utilisée consiste à inverser une liste avec [: :-1].

```
words = ["Python", "est", "puissant"]
print(words[::-1])  # ['puissant', 'est', 'Python']
```

Erreurs courantes à éviter

1. Attention aux indices hors limites

```
numbers = [1, 2, 3, 4]
print(numbers[10:])  # Retourne [] au lieu d'une erreur
```

2. Ne pas confondre slicing et accès direct

```
print(numbers[2])   # 3  (un seul élément)
print(numbers[2:3]) # [3] (une liste contenant un seul élément)
```

Le slicing sur les listes offre donc une grande flexibilité. Voyons maintenant comment appliquer ces mêmes principes aux chaînes de caractères.

Slicing sur les chaînes de caractères

Les chaînes de caractères en Python sont immuables, mais on peut les manipuler facilement grâce au slicing. Cette technique permet d'extraire des morceaux de texte, de les inverser ou de récupérer certains caractères selon un motif précis.

Extraire une sous-chaîne : Comme pour les listes, on utilise la notation *start :stop :step* pour extraire une portion de texte.

```
texte = "Python slicing"

print(texte[0:6])   # 'Python'
print(texte[:6])    # 'Python' (équivalent à [0:6])
```

```
print(texte[7:])    # 'slicing' (du 7e caractère à la fin)
```

À noter : Le caractère à l'indice stop est exclu de la sous-chaîne.

Utiliser des indices négatifs : Les indices négatifs permettent de découper la chaîne en partant de la fin.

```
print(texte[-7:])   # 'slicing' (les 7 derniers caractères)
print(texte[:-7])   # 'Python ' (tout sauf les 7 derniers)
```

Slicing avec un pas (step) : On peut utiliser step pour extraire un caractère sur deux, par exemple.

```
print(texte[::2])   # 'Pto lcig' (un caractère sur deux)
print(texte[1::2])  # 'yhnsiin' (les lettres aux indices impairs)
```

Inverser une chaîne de caractères : Une astuce très utile est d'inverser une chaîne avec [: :-1].

```
print(texte[::-1])  # 'gnicils nohtyP'
```

C'est une technique souvent utilisée pour vérifier si un mot est un **palindrome** :

```
def est_palindrome(mot):
return mot == mot[::-1]

print(est_palindrome("radar"))  # True
print(est_palindrome("python"))  # False
```

Différence entre split() et slicing : Le slicing permet d'extraire une partie fixe d'une chaîne, tandis que split() découpe le texte selon un séparateur.

```
phrase = "Python est un langage puissant"
mots = phrase.split()  # Sépare par défaut sur les espaces
print(mots)  # ['Python', 'est', 'un', 'langage', 'puissant']
```

Si on veut extraire les trois premiers mots avec slicing :

```
print(" ".join(mots[:3]))  # 'Python est un'
```

Cas d'usage pratique : récupérer un mot spécifique - On peut extraire un mot précis d'une phrase sans split().

```
phrase = "Le slicing en Python est puissant"
print(phrase[3:10])  # 'slicing'
```

Si on ne connaît pas les indices exacts, split() peut être plus flexible.

```
print(phrase.split()[1])  # 'slicing'
```

Erreurs courantes à éviter

1. Confondre slicing et accès direct

   ```
   texte = "Python"
   print(texte[2])   # 't' (un seul caractère)
   print(texte[2:3]) # 't' (sous-chaîne de longueur 1)
   ```

2. Essayer de modifier une chaîne

   ```
   texte = "Python"
   texte[0] = "J"  # Erreur : TypeError (les chaînes sont immuables)
   ```

Si on veut modifier une chaîne, on doit **la reconstruire**.

```
texte_modifie = "J" + texte[1:]
print(texte_modifie)  # 'Jython'
```

Le **slicing** sur les chaînes de caractères est donc un outil puissant pour manipuler du texte. Passons maintenant au slicing sur les tuples.

Slicing sur les tuples

Les **tuples** sont des séquences **immuables**, similaires aux listes, mais non modifiables après leur création. Le slicing fonctionne exactement comme pour les listes, sauf qu'on ne peut pas modifier le contenu du tuple.

Extraire une sous-partie d'un tuple : On utilise la notation *start :stop :step* pour obtenir une portion d'un tuple.

```
jours = ("lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
↪  "dimanche")

print(jours[1:4])  # ('mardi', 'mercredi', 'jeudi')
print(jours[:3])   # ('lundi', 'mardi', 'mercredi')
print(jours[4:])   # ('vendredi', 'samedi', 'dimanche')
```

Utiliser des indices négatifs : Comme pour les listes et les chaînes, on peut **compter à rebours** depuis la fin du tuple.

```
print(jours[-3:])   # ('vendredi', 'samedi', 'dimanche')
print(jours[:-3])   # ('lundi', 'mardi', 'mercredi', 'jeudi')
```

Slicing avec un pas (step) : On peut utiliser l'option step pour récupérer un élément sur deux.

```
print(jours[::2])  # ('lundi', 'mercredi', 'vendredi', 'dimanche')
print(jours[::-1])  # ('dimanche', 'samedi', 'vendredi', 'jeudi', 'mercredi',
↪  'mardi', 'lundi')  (inversion)
```

Différence avec les listes : immuabilité des tuples - Contrairement aux listes, **on ne peut pas modifier un tuple** après sa création.

```
jours[0] = "dimanche"  # TypeError : un tuple ne peut pas être modifié
```

Si on veut modifier un tuple, il faut le transformer en liste, effectuer les modifications, puis recréer un tuple.

```
jours_liste = list(jours)  # Conversion en liste
jours_liste[0] = "dimanche"
jours_modifie = tuple(jours_liste)  # Conversion en tuple
print(jours_modifie)
# ('dimanche', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche')
```

Utilisation du slicing sur les tuples en pratique

1. **Extraire des données d'un tuple :**Si on stocke des coordonnées GPS dans un tuple (latitude, longitude), on peut facilement extraire chaque valeur.

   ```
   coordonnees = (48.8566, 2.3522)  # Paris

   latitude = coordonnees[:1]   # (48.8566,)
   longitude = coordonnees[1:]  # (2.3522,)
   ```

2. **Obtenir les éléments d'un tuple par plage d'index :** Si on a des données triées, on peut obtenir facilement une plage spécifique.

   ```
   scores = (10, 12, 14, 16, 18, 20)
   print(scores[2:5])  # (14, 16, 18)
   ```

3. **Vérifier si un élément est présent avec slicing**

```python
def contient_element(tuple_data, element):
    return element in tuple_data[:]

print(contient_element(jours, "mardi"))  # True
print(contient_element(jours, "férié"))  # False
```

Les tuples sont donc bien compatibles avec le slicing, mais il faut garder en tête qu'ils sont immuables. Passons maintenant aux cas avancés et astuces de performance.

Slicing et copies en mémoire

Copie superficielle (shallow copy) - Le **slicing** d'une liste **crée une nouvelle liste** avec des références aux mêmes objets.

```python
original = [1, 2, [3, 4]]
copie = original[:]

copie[0] = 99
copie[2][0] = 999

print(original)  # [1, 2, [999, 4]]
print(copie)     # [99, 2, [999, 4]]
```

Explication : Les éléments **simples (entiers, chaînes)** sont copiés indépendamment, mais les objets **mutables (listes, dictionnaires, objets)** conservent leurs références.

Copie profonde (deep copy) : Si on veut copier complètement une structure imbriquée sans effet de bord, il faut utiliser copy.deepcopy().

```python
import copy

original = [1, 2, [3, 4]]
copie_profonde = copy.deepcopy(original)

copie_profonde[2][0] = 999

print(original)         # [1, 2, [3, 4]]
print(copie_profonde)   # [1, 2, [999, 4]]
```

Conclusion sur le slicing

Le **slicing en Python** est une technique essentielle pour manipuler efficacement les **listes, chaînes de caractères et tuples**. Grâce à une syntaxe simple (*start :stop :step*), il permet d'extraire, modifier ou analyser des données sans boucles complexes.

Que ce soit pour extraire des éléments spécifiques, inverser une séquence ou créer des sous-ensembles, le slicing offre une solution rapide et optimisée. Avec un bon usage des indices négatifs, des pas et des copies, on peut éviter les erreurs courantes et améliorer la performance de son code.

En combinant **slicing, compréhension de liste et gestion de mémoire**, vous maîtriserez pleinement cette technique et pourrez l'appliquer à des cas concrets en **data science, traitement de texte et algorithmes avancés**.

3.4 Les fonctions

Imaginez que vous devez expliquer à quelqu'un comment préparer un café. Au lieu de répéter chaque fois toute la procédure, vous pourriez simplement dire "prépare un café". C'est exactement ce que font les fonctions en Python : elles vous permettent de donner un nom à une série d'actions et de les réutiliser facilement.

Dans cette, vous apprendrez à créer vos propres "*recettes*" de code que vous pourrez utiliser encore et encore, rendant vos programmes plus organisés et plus faciles à comprendre.

3.4.1 Qu'est-ce qu'une fonction ?

Pensez à une fonction comme à une machine : vous lui donnez quelque chose en entrée, elle fait son travail, et vous récupérez un résultat en sortie. Parfois, elle peut juste faire une action sans rien vous rendre.

En Python, une fonction ressemble à ceci :

```
def nom_de_la_fonction():
    # Ce que fait la fonction
    print("Hello !")
```

Le mot *def* dit à Python : "Je vais créer une nouvelle fonction". Après, vous choisissez un nom (ici nom_de_la_fonction), puis vous mettez des parenthèses ().

Exemple concret - Une fonction qui dit bonjour :

```
def dire_bonjour():
    print("Bonjour ! Comment allez-vous ?")
```

Pour utiliser cette fonction (on dit "l'appeler"), écrivez simplement :

```
dire_bonjour()  # Affiche "Bonjour ! Comment allez-vous ?"
```

C'est aussi simple que ça ! Vous venez de créer votre première fonction.

3.4.2 Donner des informations à vos fonctions

Souvent, vos fonctions ont besoin d'informations pour travailler. C'est comme quand vous demandez à quelqu'un de vous préparer un sandwich : vous devez préciser quel type de pain et quelle garniture vous voulez.

En programmation, ces informations s'appellent des **paramètres** (quand vous définissez la fonction) et des **arguments** (quand vous l'utilisez).

Fonction avec un paramètre

```
def dire_bonjour_a(nom):
    print(f"Bonjour, {nom} !")

# Utilisation
dire_bonjour_a("Alice")  # Affiche "Bonjour, Alice !"
dire_bonjour_a("Bob")    # Affiche "Bonjour, Bob !"
```

Ici, *nom* est le paramètre (la "case vide" dans votre fonction), et "Alice" ou "Bob" sont les arguments (les valeurs que vous donnez).

Paramètres avec valeurs par défaut

Parfois, vous voulez que votre fonction fonctionne même si on ne lui donne pas toutes les informations. C'est comme avoir une recette avec des ingrédients optionnels :

```
def dire_bonjour_a(nom="cher ami"):
    print(f"Bonjour, {nom} !")

dire_bonjour_a("Marie")  # Affiche "Bonjour, Marie !"
dire_bonjour_a()         # Affiche "Bonjour, cher ami !"
```

Si vous ne précisez pas de nom, la fonction utilise "cher ami" par défaut.

3.4.3 Récupérer un résultat de vos fonctions

Jusqu'à présent, nos fonctions ne font qu'afficher du texte. Mais souvent, vous voulez qu'elles vous **rendent** un résultat que vous pouvez utiliser ailleurs. C'est comme utiliser une calculatrice : vous appuyez sur des boutons et elle vous donne un résultat.

Le mot magique pour cela est *return* :

```
def additionner(a, b):
    resultat = a + b
    return resultat

# Utilisation
ma_somme = additionner(5, 3)
print(ma_somme)  # Affiche 8

# Ou directement
print(additionner(10, 7))  # Affiche 17
```

Analogie simple : Imaginez une fonction comme une usine :

- Vous donnez des matières premières (les paramètres)
- L'usine fait son travail
- Elle vous rend un produit fini (la valeur de retour)

Important : Si votre fonction n'a pas de *return*, elle renvoie automatiquement *None* (c'est comme si elle ne rendait rien).

3.4.4 Exemple pratique : créer une calculatrice simple

Maintenant que vous connaissez les bases, créons quelques fonctions utiles :

```
def calculer_aire_rectangle(longueur, largeur):
    """Calcule l'aire d'un rectangle"""
    aire = longueur * largeur
    return aire

def est_nombre_pair(nombre):
    """Vérifie si un nombre est pair"""
    if nombre % 2 == 0:
        return True
    else:
        return False

# Utilisation
print(calculer_aire_rectangle(5, 3))  # Affiche 15
print(est_nombre_pair(4))              # Affiche True
print(est_nombre_pair(7))              # Affiche False
```

Ces fonctions sont utiles car vous pouvez les réutiliser partout dans votre programme sans réécrire le code.

3.4.5 Concepts avancés (optionnels pour débuter)

Les sections suivantes couvrent des concepts plus avancés. Vous pouvez les explorer une fois que vous maîtrisez les bases ci-dessus.

Les fonctions imbriquées

En **Python**, il est possible de définir une **fonction** à l'intérieur d'une autre. Cela peut être utile pour encapsuler une logique spécifique ou créer des **closures**.

Les **fonctions imbriquées** sont particulièrement utiles quand vous avez besoin de créer des **fonctions spécialisées** qui n'existent que dans un contexte précis. Elles permettent d'organiser le code en gardant des fonctions auxiliaires proches de la fonction principale qui les utilise.

Un cas typique est la **création de fonctions personnalisées** selon des paramètres, comme construire différents types de calculatrices ou de validateurs. Elles sont également essentielles

pour implémenter des **décorateurs personnalisés** et créer des **closures** qui mémorisent des valeurs de leur environnement parent.

Exemple de fonction imbriquée :

```
def operation(arithmetique):
    def addition(a, b):
        return a + b
    def soustraction(a, b):
        return a - b
    if arithmetique == "addition":
        return addition
    else:
        return soustraction

fonction_calc = operation("addition")
print(fonction_calc(10, 5))  # Affiche 15
```

Les fonctions courtes (lambda) - Version simplifiée

Parfois, vous avez besoin d'une fonction très simple pour une tâche rapide. Au lieu d'écrire une fonction complète, vous pouvez utiliser ce qu'on appelle une *"lambda"* :

```
# Au lieu d'écrire :
def multiplier(x, y):
    return x * y

# Vous pouvez écrire :
multiplier = lambda x, y: x * y

print(multiplier(4, 5))  # Affiche 20
```

Quand utiliser *"lambda"*? Pour des fonctions très simples d'une seule ligne. C'est comme utiliser une calculatrice de poche au lieu d'installer un logiciel complet.

Les fonctions *"lambda"* sont idéales pour les opérations de transformation rapides sur des collections de données, notamment avec les fonctions *map()*, *filter()* et *sorted()*. Elles permettent d'éviter de créer des fonctions nommées pour des calculs ponctuels et simples.

Elles sont particulièrement appréciées dans le **traitement de données** (tri personnalisé, filtrage, transformations mathématiques) et dans la **programmation fonctionnelle**. Les *"lambda"* excellent aussi comme **fonctions de rappel** (*callbacks*) dans les interfaces graphiques ou les API qui attendent une fonction simple en paramètre.

Conseil pour débuter : Ne vous préoccupez pas trop des *"lambda"* au début. Les fonctions normales avec *def* sont plus claires et suffisantes.

Les arguments variables (*args et **kwargs)

Pour créer des fonctions acceptant un nombre variable d'arguments, on utilise **args* pour les arguments positionnels et ***kwargs* pour les arguments nommés.

Les arguments variables sont essentiels pour créer des fonctions flexibles qui peuvent traiter un nombre indéterminé de paramètres. **args* est parfait pour des fonctions comme somme(), maximum(), ou concatener() où vous ne savez pas à l'avance combien de valeurs vous allez recevoir.

***kwargs* est indispensable pour créer des fonctions de configuration où vous voulez accepter des options nommées variées, ou pour transmettre des paramètres à d'autres fonctions. Ces mécanismes sont couramment utilisés dans les frameworks web, les APIs, et pour créer des wrappers autour de fonctions existantes.

Exemple avec **args* :

```
def somme(*nombres):
    total = 0
    for nombre in nombres:
        total += nombre
    return total

print(somme(1, 2, 3, 4))  # Affiche 10
```

Exemple avec ***kwargs* :

```
def afficher_infos(**infos):
    for cle, valeur in infos.items():
        print(f"{cle} : {valeur}")

afficher_infos(nom="Alice", age=30, ville="Paris")
# Affiche :
# nom : Alice
# age : 30
# ville : Paris
```

Les annotations de type

Depuis **Python 3.5**, il est possible d'ajouter des **annotations de type** aux paramètres et au type de retour des fonctions. Cela améliore la lisibilité du code et facilite la maintenance.

Exemple d'annotations de type :

```
def saluer(nom: str) -> None:
    print(f"Bonjour, {nom} !")
```

Ces annotations sont informatives et n'affectent pas l'exécution du programme.

Les fonctions récursives

Une **fonction récursive** est une fonction qui s'appelle elle-même. C'est utile pour résoudre des problèmes pouvant être décomposés en sous-problèmes similaires.

La **récursivité** est naturelle pour traiter des **structures hiérarchiques** comme les arbres de fichiers, les menus imbriqués, ou les structures de données arborescentes. Elle excelle dans la résolution de **problèmes mathématiques** (factorielles, suites de Fibonacci, calculs combinatoires) et les **algorithmes de tri** (tri rapide, tri fusion).

Les **fonctions récursives** sont également utiles pour parcourir des structures complexes dont la profondeur n'est pas connue à l'avance, comme analyser du JSON imbriqué ou naviguer dans des répertoires avec des sous-dossiers. Cependant, attention aux performances et aux limites de récursion pour de grandes données.

Exemple de fonction récursive pour calculer la factorielle :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)

print(factorielle(5))  # Affiche 120
```

3.4.6 Bonnes pratiques pour écrire des fonctions en Python

Pour écrire des fonctions efficaces et maintenables en Python, il est important de suivre certaines bonnes pratiques. Ces principes améliorent la lisibilité du code, facilitent la collaboration entre développeurs et réduisent les risques d'erreurs.

Nommer les fonctions de manière claire et descriptive

Les noms de fonctions doivent refléter précisément leur rôle. Un nom clair permet de comprendre rapidement ce que fait la fonction sans avoir à lire son code interne.

Exemple :

```
def calculer_moyenne(notes):
    return sum(notes) / len(notes)
```

Garder les fonctions courtes et spécifiques

Une fonction doit accomplir une seule tâche ou fonctionnalité spécifique. Cela facilite la compréhension, le test et la maintenance du code.

Exemple : Au lieu de créer une fonction qui effectue plusieurs opérations, il est préférable de la diviser en fonctions plus petites.

```python
def lire_donnees(fichier):
    # Code pour lire les données
    pass

def analyser_donnees(donnees):
    # Code pour analyser les données
    pass

def afficher_resultats(resultats):
    # Code pour afficher les résultats
    pass
```

Utiliser les docstrings pour documenter les fonctions

Les **docstrings** sont des chaînes de caractères placées immédiatement après la définition d'une fonction pour la documenter. Elles permettent d'expliquer le rôle de la fonction, ses paramètres, ses valeurs de retour et d'autres informations utiles. En Python, les **docstrings** sont accessibles via l'attribut *__doc__* de la fonction ou avec la fonction *help()*.

Exemple de **docstring** :

```python
def additionner(a, b):
    """
    Additionne deux nombres et renvoie le résultat.

    :param a: Le premier nombre à additionner.
    :param b: Le second nombre à additionner.
    :return: La somme de a et b.
    """
    return a + b
```

Dans cet exemple, la **docstring** explique ce que fait la fonction *additionner*, décrit les paramètres *a* et *b* et indique ce que la fonction retourne. Cette documentation est précieuse pour les autres développeurs ou pour soi-même lors de la maintenance du code.

Pour consulter la **docstring**, il est possible d'utiliser *help()* :

```python
help(additionner)
```

Ce qui affichera :

```
Help on function additionner in module __main__:

additionner(a, b)
    Additionne deux nombres et renvoie le résultat.

    :param a: Le premier nombre à additionner.
    :param b: Le second nombre à additionner.
    :return: La somme de a et b.
```

Bonnes pratiques pour les docstrings :

- Commencer par une phrase courte résumant le rôle de la fonction.
- Décrire les **paramètres** et les **valeurs de retour** de manière claire.
- Utiliser un style cohérent, comme le format *reStructuredText* ou *Google Docstrings*.
- Éviter les informations redondantes ou évidentes.
- Mettre à jour la **docstring** si la fonction est modifiée.

Les **docstrings** peuvent également être utilisées par des outils de génération de documentation automatique, ce qui facilite la création de documents complets sur le code.

Éviter les effets de bord

Une fonction doit idéalement être pure, c'est-à-dire qu'elle ne modifie pas l'état du programme en dehors de son *scope* et ne dépend pas de variables externes. Cela rend la fonction plus prévisible et facilite le débogage.

Exemple : Éviter de modifier des variables globales à l'intérieur d'une fonction.

```
# Moins recommandé
compteur = 0

def incrementer_compteur():
    global compteur
    compteur += 1

# Mieux
def incrementer(compteur):
    return compteur + 1
```

Utiliser les annotations de type

Les **annotations de type** améliorent la lisibilité du code et aident à détecter les erreurs potentielles. Elles indiquent les types attendus pour les paramètres et la valeur de retour.

Exemple :

```
def saluer(nom: str) -> None:
    print(f"Bonjour, {nom} !")
```

3.4.7 Gérer les exceptions de manière appropriée

Il est important de prévoir et de gérer les erreurs potentielles au sein des fonctions pour éviter les plantages inattendus.

Exemple :

```python
def diviser(a, b):
    """
    Divise a par b et gère la division par zéro.
    """
    try:
        return a / b
    except ZeroDivisionError:
        print("Erreur : division par zéro.")
        return None
```

Éviter les arguments par défaut mutables

Utiliser des objets mutables comme valeurs par défaut peut entraîner des comportements inattendus.

Exemple problématique :

```python
def ajouter_element(element, liste=[]):
    liste.append(element)
    return liste
```

À chaque appel sans spécifier la liste, l'élément sera ajouté à la même liste, ce qui peut causer des bugs.

Solution :

```python
def ajouter_element(element, liste=None):
    if liste is None:
        liste = []
    liste.append(element)
    return liste
```

Respecter les conventions de style

Suivre les conventions de style, comme celles définies dans PEP 8, rend le code plus cohérent et facile à lire.

Exemples de conventions :

- Utiliser des noms de fonctions en minuscules avec des underscores : calculer_moyenne
- Indenter le code avec quatre espaces
- Limiter la longueur des lignes à 79 caractères

3.4.8 Travaux pratiques sur les fonctions

Vous avez maintenant les bases pour écrire des scripts Python simples et utiliser des structures de données comme les listes, les tuples et les dictionnaires. Pour consolider vos connaissances, il est temps de mettre en pratique ce que vous avez appris.

Ce TP te guide pas à pas pour maîtriser les fonctions Python : de la définition la plus simple jusqu'à la récursion, en passant par les paramètres avancés et les *lambdas*.

Objectifs pédagogiques

- Définir et appeler une fonction
- Manipuler les paramètres (positionnels, nommés, valeurs par défaut)
- Retourner des valeurs et documenter son code
- Utiliser **args* et ***kwargs*
- Créer des *lambdas* et des fonctions imbriquées (**closures**)
- Ajouter des annotations de type et des ***lambdas*** claires
- Comprendre la récursivité
- Valider ses compétences grâce à un challenge automatisé

1. **Étape 1 – Créer *salutations.py*** et y mettre le code suivant :

```
#!/usr/bin/env python3

def saluer():
    """Affiche un message de bienvenue."""
    print("Bonjour ! Bienvenue dans les fonctions.")

if __name__ == "__main__":
    saluer()
```

   Ici if *__name__ == "__main__" :* permet d'exécuter la fonction *saluer()* uniquement si le script est lancé directement, pas s'il est importé.

   Rends le script exécutable :

```
chmod +x salutations.py
```

   Exécute :

```
./salutations.py
```

2. **Étape 2 – Ajouter des paramètres et une valeur par défaut**

   (a) Modifie la fonction saluer() dans salutations.py avec ce contenu :

```python
#!/usr/bin/env python3

def saluer_personne(nom: str, titre: str = "cher visiteur")
 ↪  -> None:
    """Salue la personne avec un titre optionnel."""
    print(f"Bonjour, {titre} {nom} !")

if __name__ == "__main__":
    saluer_personne("Alice")  # Appel sans titre
    saluer_personne("Bob", titre="Dr")  # Appel avec titre
```

(b) Teste depuis le terminal :

```
./salutations.py
```

3. **Étape 3 – Valeur de retour et annotations de type**

(a) Crée le fichier *calculs.py* et ajoute y le code suivant :

```python
#!/usr/bin/env python3

def additionner(a: float, b: float) -> float:
    """Retourne la somme de deux nombres."""
    return a + b

if __name__ == "__main__":
    print(additionner(2, 3))
```

(b) Rends le script exécutable :

```
chmod +x calculs.py
```

(c) Exécute :

```
./calculs.py
```

Quelle sera le résultat ?

4. **Étape 4 – *args et **kwargs**

(a) Dans *calculs.py*, ajopute les fonctions suivantes

```python
def somme(*nombres: float) -> float:
    """Additionne un nombre arbitraire d'arguments."""
    return sum(nombres)

def afficher_infos(**infos) -> None:
    """Affiche les paires clé / valeur fournies."""
```

```python
    for cle, valeur in infos.items():
        print(f"{cle} : {valeur}")


if __name__ == "__main__":
    print(additionner(2, 3))
    print(somme(1, 2, 3, 4))
    afficher_infos(nom="Alice", age=30, ville="Paris")
```

(b) Teste ton code :

```
./calculs.py
```

5. **Étape 5 – Les lambdas**

(a) Crée le fichier *lambda.py* et ajoute-y le code suivant :

```python
#!/usr/bin/env python3
multiplication = lambda x, y: x * y

if __name__ == "__main__":
    print(multiplication(3, 4))          # 12
```

(b) Teste le :

```
python3 lambdas.py
```

6. **Étape 6 – Fonctions imbriquées (closures)**

(a) Crée le fichier *closures.py* et ajoute-y le code suivant :

```python
#!/usr/bin/env python3

def create_multiplier(facteur: float) -> callable:
    """Retourne une fonction qui multiplie par un facteur
    ↪ donné."""
    def multiplier(x: float) -> float:
        return x * facteur
    return multiplier

if __name__ == "__main__":
    double = create_multiplier(2)
    print(double(5))  # 10
```

(b) Teste le :

```
python3 closures.py
```

7. **Étape 7 – Récursivité**

   (a) Rouvre *calculs.py* et ajoute :

```python
def factorielle(n: int) -> int:
    """Calcule n! récursivement."""
    if n == 0:
        return 1
    return n * factorielle(n - 1)

if __name__ == "__main__":
    print(additionner(2, 3))
    print(somme(1, 2, 3, 4))
    afficher_infos(nom="Alice", age=30, ville="Paris")
    print(factorielle(5))  # 120
```

   (b) Teste ton code :

```
./calculs.py
```

Pour mieux structurer le code, le rendre réutilisable et facile maintenable maintenable, vous devez apprendre à créer et manipuler les modules ainsi que les packages.

3.5 Les modules et les packages

Les modules Python permettent de structurer votre code, le rendre réutilisable et maintenable. Un module est simplement un fichier Python contenant des fonctions (3.4), des variables, des classes, etc., que vous pouvez importer et utiliser dans d'autres scripts ou projets Python. Cette section détaillée vous expliquera comment créer vos propres modules, comment les organiser et comment les utiliser dans différents projets.

3.5.1 Qu'est-ce qu'un Module Python ?

Un **module Python** est un fichier contenant du code Python qui peut définir des fonctions, des classes et des variables. Il permet de regrouper du code liés à une fonctionnalité spécifique, facilitant ainsi la réutilisation et la maintenance du code. Par exemple, si vous avez plusieurs fonctions liées aux calculs mathématiques, vous pouvez les regrouper dans un module que vous appelerez *math_utils.py*.

3.5.2 Pourquoi Utiliser des Modules ?

Avant de plonger dans les détails techniques, voyons pourquoi les modules sont utiles :

- **Réutilisation** : Un module peut être utilisé dans plusieurs projets, évitant de réécrire du code.
- **Organisation** : Ils aident à organiser le code en le découpant en petits morceaux logiques.
- **Maintenance** : Une bonne modularisation facilite la mise à jour ou la correction d'un programme sans affecter l'ensemble du projet.

3.5.3 Comment créer un Module Python

Un module Python est simplement un fichier Python avec l'extension *.py*. Par exemple, si vous avez un fichier appelé *math_utils.py*, il devient un module que vous pouvez importer et utiliser dans d'autres scripts Python.

Exemple : Créez un fichier nommé *math_utils.py* et ajoutez-y quelques fonctions simples.

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

def soustraction(a, b):
    return a - b

def multiplication(a, b):
    return a * b

def division(a, b):
    if b == 0:
        raise ValueError("Division par zéro non permise")
    return a / b
```

Ce fichier contient quatre fonctions : *addition*, soustraction, multiplication et division. Maintenant, ce module peut être importé dans un autre fichier Python.

3.5.4 Importation et Utilisation d'un Module

Une fois que votre module est créé, vous pouvez l'utiliser dans un autre script Python en utilisant l'instruction import.

Exemple :

Créez un nouveau fichier nommé *main.py* pour utiliser les fonctions définies dans le module *math_utils.py* :

```
# Code source du fichier main.py

# Importer le module math_utils
import math_utils

# Utiliser les fonctions du module
resultat1 = math_utils.addition(10, 5)
```

```
resultat2 = math_utils.division(10, 2)

print(f"Résultat de l'addition : {resultat1}")
print(f"Résultat de la division : {resultat2}")
```

Sortie attendue :

```
Résultat de l'addition : 15
Résultat de la division : 5.0
```

Dans cet exemple, le fichier *main.py* importe le module *math_utils.py* et utilise ses fonctions.

3.5.5 Organisation des Modules avec les Packages

Un **package** en Python est un moyen d'organiser plusieurs modules dans des sous-répertoires pour créer une structure plus complexe. Un package est simplement un répertoire contenant un ou plusieurs modules et un fichier spécial *__init__.py*. Ce fichier peut être vide, mais il indique à Python que ce répertoire doit être traité comme un package.

Exemple :

Supposons que vous vouliez organiser vos modules pour un projet de calcul mathématique complexe. Vous pouvez structurer votre package ainsi :

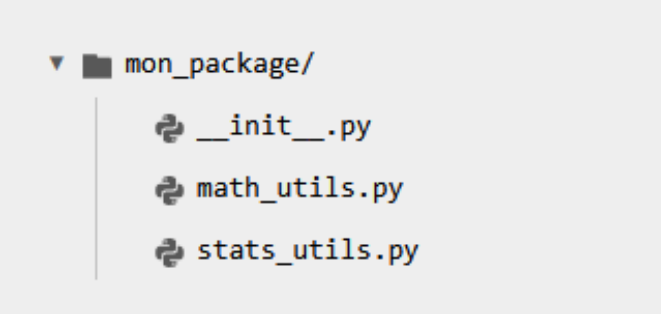


FIGURE 3.1 – Exemple de structuration d'un package

- *__init__.py* : Ce fichier indique que *mon_package* est un package Python.
- *math_utils.py* : Contient des fonctions de base de calcul (comme nous l'avons vu précédemment).
- *stats_utils.py* : Contiendra des fonctions pour des calculs statistiques.

```
# Code source du fichier stats_utils.py

def moyenne(liste):
    return sum(liste) / len(liste)

def mediane(liste):
```

```
    liste_triee = sorted(liste)
    n = len(liste_triee)
    if n % 2 == 1:
        return liste_triee[n // 2]
    else:
        return (liste_triee[n // 2 - 1] + liste_triee[n // 2]) / 2
```

Vous pouvez le laisser vide, ou y ajouter du code pour personnaliser le comportement de votre package :

```
# Code source du fichier __init__.py

print("Le package 'mon_package' a été chargé")
```

Maintenant, pour utiliser ce package, vous pouvez simplement faire :

```
# Code source du fichier main.py

# Importer le package
from mon_package import math_utils, stats_utils

# Utiliser les fonctions du package
resultat_add = math_utils.addition(10, 20)
resultat_moy = stats_utils.moyenne([1, 2, 3, 4, 5])

print(f"Résultat de l'addition : {resultat_add}")
print(f"Moyenne des nombres : {resultat_moy}")
```

Sortie attendue :

```
Le package 'mon_package' a été chargé
Résultat de l'addition : 30
Moyenne des nombres : 3.0
```

3.5.6 Gestion des Importations dans les Packages

Lorsque vous avez un grand nombre de modules dans un package, vous pouvez structurer les importations pour qu'elles soient plus simples à utiliser.

Dans votre fichier *__init__.py*, vous pouvez définir un tableau *__all__* pour spécifier les modules ou sous-modules que vous souhaitez rendre disponibles lorsque le package est importé avec from *mon_package import **.

Exemple :

```
# Code source du fichier __init__.py

__all__ = ["math_utils", "stats_utils"]
```

Cela permet de spécifier quels modules sont accessibles à partir du package. Sans cela, les utilisa-

teurs devront importer chaque module explicitement.

3.5.7 Documentation des Modules

Il est indispensable de documenter correctement vos modules et fonctions pour qu'ils soient faciles à comprendre par d'autres développeurs (ou vous-même dans le futur).

Utilisez des **docstrings** pour décrire chaque module, classe et fonction.

Exemple :

```
# Code source du fichier math_utils.py

def addition(a, b):
    """
    Ajoute deux nombres.

    :param a: Le premier nombre.
    :param b: Le second nombre.
    :return: La somme de a et b.
    """
    return a + b
```

3.5.8 Création de Modules Installables

Si vous souhaitez distribuer votre module pour qu'il soit installé via *pip*, vous devez créer une structure spécifique et un fichier *setup.py*.

Exemple de Structure :

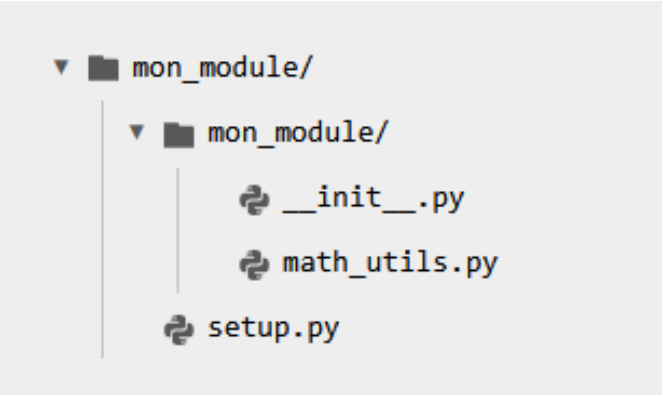


FIGURE 3.2 – Exemple de structuration d'un package

```
# Code source du fichier setup.py

from setuptools import setup, find_packages

setup(
```

```
    name='mon_module',
    version='1.0',
    packages=find_packages(),
    description='Un package de mathématiques simples',
    author='Stéphane Robert',
    author_email='stephane@example.com',
    install_requires=[],
)
```

Une fois que cela est configuré, vous pouvez créer une distribution et l'installer avec *pip*.

Pour compléter votre guide, voici une section détaillée sur la publication des packages sur *PyPI* (Python Package Index), qui vous permettra de partager vos modules avec la communauté Python. Vous pouvez facilement publier vos packages sur *PyPI* afin que d'autres développeurs puissent les installer avec *pip*.

3.5.9 Publication des Packages sur PyPI

Une fois que vous avez créé un package Python, vous pouvez le partager en le publiant sur *PyPI* (Python Package Index). Cela permet à d'autres développeurs de l'installer en utilisant simplement *pip*.

Prérequis

Avant de publier sur *PyPI*, assurez-vous d'avoir :

- Un compte sur pypi.org — créez un compte si vous n'en avez pas déjà un.
- *Python* et *setuptools* installés sur votre machine :

```
pip install setuptools wheel twine
```

Structure du Package

Assurez-vous que votre package a la structure suivante :

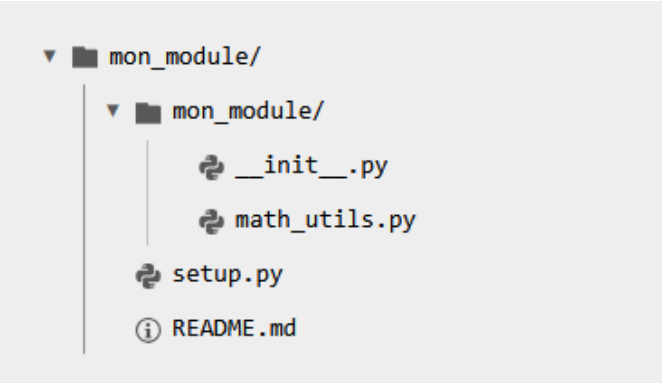


FIGURE 3.3 – Exemple de structuration d'un package

Voici un exemple de fichier *setup.py* que vous pouvez adapter selon votre projet :

```
from setuptools import setup, find_packages

setup(
    name='mon_module',
    version='1.0.0',
    packages=find_packages(),
    description='Un package de mathématiques simples',
    long_description=open('README.md').read(),
    long_description_content_type='text/markdown',
    author='Stéphane Robert',
    author_email='stephane@example.com',
    url='https://github.com/stephane-robert/mon_module',
    classifiers=[
        'Programming Language :: Python :: 3',
        'License :: OSI Approved :: MIT License',
        'Operating System :: OS Independent',
    ],
    python_requires='>=3.6',
    install_requires=[],
)
```

Création de l'Archive

Avant de publier le package, vous devez créer une archive du package. Pour cela, vous pouvez utiliser *setuptools* et *wheel* pour générer les fichiers nécessaires.

Exécutez la commande suivante à la racine de votre projet, où se trouve le fichier *setup.py* :

```
python setup.py sdist bdist_wheel
```

Cela créera un répertoire *dist/* contenant les fichiers de distribution *.tar.gz* et *.whl* (fichiers *wheel*).

Publication avec Twine

Ensuite, vous pouvez utiliser l'outil *twine* pour publier votre package sur PyPI.

1. Vérifiez d'abord que vos fichiers sont corrects en les testant sur TestPyPI, une instance de *PyPI* qui permet de tester le processus de publication :

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

2. Une fois que vous êtes prêt à publier sur le vrai PyPI, utilisez la commande suivante :

```
twine upload dist/*
```

3. Entrez votre nom d'utilisateur et mot de passe PyPI lorsque cela vous est demandé.

3.5.10 Installation du Package

Après la publication, votre package sera disponible sur PyPI et vous (ou tout autre utilisateur) pourrez l'installer en utilisant :

```
pip install mon_module
```

3.5.11 Mise à Jour de Votre Package

Si vous devez publier une nouvelle version de votre package, mettez simplement à jour la version dans *setup.py* (par exemple, de *version='1.0.0'* à *version='1.1.0'*), recréez les fichiers de distribution et répétez la commande *twine upload* :

```
python setup.py sdist bdist_wheel
twine upload dist/*
```

3.5.12 Bonnes Pratiques

Voici quelques bonnes pratiques à suivre lors de la création de vos modules :

- Nommez vos modules clairement : Choisissez des noms explicites pour vos modules et fonctions afin qu'ils soient faciles à comprendre.
- Utilisez des docstrings : Toujours documenter vos fonctions et modules.
- Évitez les importations circulaires : Les modules qui s'importent mutuellement peuvent provoquer des erreurs.
- Modularisez votre code : Ne mettez pas trop de fonctions ou de classes dans un même module, divisez-les logiquement.

- Testez votre code : Utilisez des tests unitaires pour vous assurer que chaque module fonctionne correctement de manière isolée.

Pour la Publication sur Pypi

- Choisissez un nom unique pour votre package sur PyPI pour éviter les conflits avec d'autres packages existants.
- Documentez bien votre projet : Ajoutez un fichier *README.md* complet pour que les utilisateurs comprennent comment utiliser votre package.
- Utilisez des versions cohérentes : Suivez les conventions de versionnement sémantique (*SemVer*) pour indiquer clairement les mises à jour majeures et mineures de votre package.
- Testez toujours sur TestPyPI avant de publier sur le vrai PyPI pour éviter les erreurs.

3.5.13 En conclusion sur les modules et les packages

Créer des **modules** et des **packages** Python vous permet de structurer votre code de manière réutilisable et maintenable. Que ce soit pour vos propres projets ou pour distribuer vos modules à d'autres développeurs, comprendre les bases de la création de modules est essentiel pour tout développeur Python. N'hésitez pas à explorer davantage les packages, la documentation et les bonnes pratiques pour améliorer votre maîtrise des modules Python.

C'est la fin de chapitre certe long et qui vous sera très utlile dans votre carrière de développeur Python. Toutefois, une autre notion, aussi importante qu'indispensable, vous attends au chapitre suivant : La POO en python.

mais avant de vous y frotter, 2 travaux pratiques ainsi que des exercices vous attendes pour consolider les connaissances acquises dans ce chapitre.

3.6 TP sur le chapitre

TP :

- Système de gestion d'étudiants (ajout, suppression, recherche, affichage).
- Jeu du nombre mystère.

3.6.1 TP 1 : Système de gestion d'étudiants

Développer un système de gestion des étudiants permettant :

- L'ajout d'un étudiant ;
- La suppression d'un étudiant ;
- La recherche d'un étudiant
- La modification des informations
- L'affichage de tous les étudiants

Le tout en utilisant les structures de données Python et les fonctions.

Chaque étudiant sera représenté par un dictionnaire :

```
etudiant = {
    "matricule": "ET001",
    "nom": "DUPONT",
    "prenom": "Jean",
    "age": 20,
    "filiere": "Informatique"
}
```

L'ensemble des étudiants sera stocké dans :

```
etudiants = []
```

etudiants est donc une liste de **dictionnaires**.

Voici le travail qui est demandé :

1. Écrire une fonction afficher_menu() qui affiche :

```
===== MENU GESTION DES ETUDIANTS =====
1. Ajouter un étudiant
2. Supprimer un étudiant
3. Rechercher un étudiant
4. Modifier un étudiant
5. Afficher tous les étudiants
```

```
6. Quitter
```

2. Implémenter les fonctions suivantes :

   - Ajouter un étudiant via *def ajouter_etudiant(etudiants)*

     — demander : matricule, nom, prénom, âge, filière ;
     — vérifier que le matricule n'existe pas déjà ;
     — ajouter l'étudiant à la liste.

   - Afficher tous les étudiants via *def afficher_etudiants(etudiants)*

     — afficher chaque étudiant proprement
     — si la liste est vide, afficher un message approprié. Exemple d'afficher :

     ```
     Matricule : ET001
     Nom : DUPONT
     Prénom : Jean
     Âge : 20
     Filière : Informatique
     -------------------------
     ```

   - Rechercher un étudiant via *def rechercher_etudiant(etudiants, matricule)*

     — rechercher un étudiant par son matricule ;
     — afficher ses informations s'il existe ;
     — sinon afficher : "Étudiant introuvable".

   - Supprimer un étudiant via *def supprimer_etudiant(etudiants, matricule)*

     — rechercher l'étudiant ;
     — le supprimer de la liste ;
     — afficher un message de confirmation ;
     — Utiliser remove() ou un parcours par indice.

   - Modifier un étudiant via *def modifier_etudiant(etudiants, matricule*

     — rechercher l'étudiant ;
     — proposer la modification de (nom, prénom, âge, filière) ;
     — conserver le matricule inchangé.

3. Programme principal : Écrire une boucle (boucle while) principale qui

   - affiche le menu
   - demande le choix de l'utilisateur
   - appelle la fonction correspondante

- s'arrête uniquement lorsque l'utilisateur choisit **Quitter**

Proposition de correction : Veuillez créer un fichier *gestion_etudiant.py* et y insérer les différentes fonctions ci-dessous développées.

```
# Liste globale des étudiants
etudiants = []
```

Pour rappel, chaque étudiant est représenté par un dictionnaire :

```
{
    "matricule": "ET001",
    "nom": "DUPONT",
    "prenom": "Jean",
    "age": 20,
    "filiere": "Informatique"
}
```

1. **Affichage du menu :**

```
def afficher_menu():
    print("\n===== MENU GESTION DES ETUDIANTS =====")
    print("1. Ajouter un étudiant")
    print("2. Supprimer un étudiant")
    print("3. Rechercher un étudiant")
    print("4. Modifier un étudiant")
    print("5. Afficher tous les étudiants")
    print("6. Quitter")
```

2. **Ajouter un étudiant :**

```
def ajouter_etudiant(etudiants):
    matricule = input("Matricule : ")

    # Vérifier l'unicité du matricule
    for e in etudiants:
        if e["matricule"] == matricule:
            print("Erreur : ce matricule existe déjà.")
            return

    nom = input("Nom : ")
    prenom = input("Prénom : ")
    age = int(input("Âge : "))
    filiere = input("Filière : ")

    etudiant = {
        "matricule": matricule,
        "nom": nom,
        "prenom": prenom,
        "age": age,
        "filiere": filiere
```

```
    }

    etudiants.append(etudiant)
    print("Étudiant ajouté avec succès.")
```

3. **Afficher tous les étudiants :**

```
def afficher_etudiants(etudiants):
    if len(etudiants) == 0:
        print("Aucun étudiant enregistré.")
        return

    print("\n===== LISTE DES ETUDIANTS =====")
    for e in etudiants:
        print("Matricule :", e["matricule"])
        print("Nom :", e["nom"])
        print("Prénom :", e["prenom"])
        print("Âge :", e["age"])
        print("Filière :", e["filiere"])
        print("----------------------------")
```

4. **Rechercher un étudiant :**

```
def rechercher_etudiant(etudiants, matricule):
    for e in etudiants:
        if e["matricule"] == matricule:
            print("\nÉtudiant trouvé :")
            print("Matricule :", e["matricule"])
            print("Nom :", e["nom"])
            print("Prénom :", e["prenom"])
            print("Âge :", e["age"])
            print("Filière :", e["filiere"])
            return

    print("Étudiant introuvable.")
```

5. **Supprimer un étudiant :**

```
def supprimer_etudiant(etudiants, matricule):
    for e in etudiants:
        if e["matricule"] == matricule:
            etudiants.remove(e)
            print("Étudiant supprimé avec succès.")
            return

    print("Étudiant introuvable.")
```

6. **Modifier un étudiant :**

```
def modifier_etudiant(etudiants, matricule):
    for e in etudiants:
        if e["matricule"] == matricule:
```

```
            print("Modifier les informations (laisser vide pour conserver)")

            nom = input("Nouveau nom : ")
            prenom = input("Nouveau prénom : ")
            age = input("Nouvel âge : ")
            filiere = input("Nouvelle filière : ")

            if nom != "":
                e["nom"] = nom
            if prenom != "":
                e["prenom"] = prenom
            if age != "":
                e["age"] = int(age)
            if filiere != "":
                e["filiere"] = filiere

            print("Informations mises à jour.")
            return

    print("Étudiant introuvable.")
```

7. **Programme principal :**

```
def main():
    etudiants = []

    while True:
        afficher_menu()
        choix = input("Votre choix : ")

        if choix == "1":
            ajouter_etudiant(etudiants)

        elif choix == "2":
            matricule = input("Matricule à supprimer : ")
            supprimer_etudiant(etudiants, matricule)

        elif choix == "3":
            matricule = input("Matricule à rechercher : ")
            rechercher_etudiant(etudiants, matricule)

        elif choix == "4":
            matricule = input("Matricule à modifier : ")
            modifier_etudiant(etudiants, matricule)

        elif choix == "5":
            afficher_etudiants(etudiants)

        elif choix == "6":
            print("Fin du programme.")
            break

        else:
            print("Choix invalide.")
```

8. Enfin ajouter le code suivant dans le même fichier *gestion_etudiant.py*

```
main()
```

Exécutez votre script python pour tester la solution proposée.

Nous allons améliorer ce TP en y insérant la gestion des notes, le calcul des moyennes et l'attribution des mentions.

Pour y arriver, nous proposons la nouvelle structure des données ci-dessous :

```
{
    "matricule": "ET001",
    "nom": "DUPONT",
    "prenom": "Jean",
    "age": 20,
    "filiere": "Informatique",
    "notes": [12, 15, 10]
}
```

NB : *notes* est une liste d'entiers ou de réels.

1. **Ajouter des notes à un étudiant :** L'objectif est de permettre l'ajout de plusieurs notes à un étudiant existant via la fonction *ajouter_notes*

```
def ajouter_notes(etudiants, matricule):
    for e in etudiants:
        if e["matricule"] == matricule:
            n = int(input("Nombre de notes à ajouter : "))

            for i in range(n):
                note = float(input(f"Note {i+1} : "))
                if 0 <= note <= 20:
                    e["notes"].append(note)
                else:
                    print("Note invalide (0 à 20).")

            print("Notes ajoutées avec succès.")
            return

    print("Étudiant introuvable.")
```

2. **Calculer la moyenne à partir de la liste de notes**

```
def calculer_moyenne(notes):
    if len(notes) == 0:
        return 0

    somme = 0
    for n in notes:
```

```
        somme += n

    return somme / len(notes)
```

3. **Déterminer la mention selon les règles suivantes :**

   — Mois de 10 : Echec ;
   — De 10 à 11,99 : Passable ;
   — De 12 à 13,99 : Assez bien ;
   — De 14 à 15,99 : Bien ;
   — A partir de 16 : Très bien.

```
def determiner_mention(moyenne):
    if moyenne < 10:
        return "Échec"
    elif moyenne < 12:
        return "Passable"
    elif moyenne < 14:
        return "Assez bien"
    elif moyenne < 16:
        return "Bien"
    else:
        return "Très bien"
```

4. **Afficher le bulletin d'un étudiant**

```
def afficher_bulletin(etudiants, matricule):
    for e in etudiants:
        if e["matricule"] == matricule:
            moyenne = calculer_moyenne(e["notes"])
            mention = determiner_mention(moyenne)

            print("\n===== BULLETIN DE NOTES =====")
            print("Matricule :", e["matricule"])
            print("Nom :", e["nom"])
            print("Prénom :", e["prenom"])
            print("Filière :", e["filiere"])
            print("Notes :", e["notes"])
            print("Moyenne :", round(moyenne, 2))
            print("Mention :", mention)
            return

    print("Étudiant introuvable.")
```

5. **Mise à jour de la fonction d'ajout d'étudiant :** Lors de l'ajout d'un étudiant, vous devez initialiser la liste des notes. Modifiez le code de l'ajout d'un nouvel étudiant comme ceci dans la fonction *ajouter_etudiant(etudiants)* :

```
etudiant = {
    "matricule": matricule,
```

```
    "nom": nom,
    "prenom": prenom,
    "age": age,
    "filiere": filiere,
    "notes": []
}
```

6. **Extension du menu principal**

```
def afficher_menu():
    print("\n===== MENU GESTION DES ETUDIANTS =====")
    print("1. Ajouter un étudiant")
    print("2. Supprimer un étudiant")
    print("3. Rechercher un étudiant")
    print("4. Modifier un étudiant")
    print("5. Afficher tous les étudiants")
    print("6. Ajouter des notes")
    print("7. Afficher le bulletin d'un étudiant")
    print("8. Quitter")
```

7. **Mise à jour du programme principal :** Ajoutez ceci à la fonction *main()*

```
elif choix == "6":
    matricule = input("Matricule : ")
    ajouter_notes(etudiants, matricule)

elif choix == "7":
    matricule = input("Matricule : ")
    afficher_bulletin(etudiants, matricule)

elif choix == "8":
    print("Fin du programme.")
    break
```

3.6.2 TP 2 : Le nombre mystère

Enoncé

Dans ce projet, tu vas devoir recréer le jeu du nombre mystère. L'ordinateur va choisir un nombre entre 1 et 100 et ton objectif est de trouver ce nombre.

Bien entendu, tu as droit à un nombre limité d'essais (5) pour trouver le nombre. Vous allez donc devoir utiliser un module que l'on a vu dans les parties précédentes, les boucles, la fonction input.

Bref, là encore, beaucoup de notions qui individuellement ne sont pas compliquées, mais qu'il va falloir que tu saches agencer ensemble dans une logique particulière.

Proposition de correction

```
# Ce programme permet de dire si le nombre entré par un user est celui généré
↪ aléatoirement par l'ordi.

# Importation de la librairie random
import random

# Nbre généré entre 0 et 100
resultat = random.randint(0, 100)

# Liste qui enregistre l'historique des essais du user
essai = []
chance = 5

while(chance > 0):
    # On récupère le nombre entré par le user
    nbreUser = int(input("Dévinez le nombre entier qui a été généré entre 0 et
    ↪ 100: "))
    essai.append(nbreUser)

    if(nbreUser == resultat):
        print("Historique de vos tentatives: {0}". format(essai))
        print("Bravo !!! Vous avez gagné en {0} essai(s).". format(len(essai)))
        print("Le nombre à déviner était {0} !!!\n". format(resultat))
        break

    else:
        if(nbreUser < resultat):
            print("Le nombre à déviner est plus grand\n")
        else:
            print("Le nombre à déviner est plus petit\n")
        chance -= 1
        if(chance == 1):
            print("C'est votre dernière chance de trouver le nombre.\n")
        else:
            print("Il vous reste {0} chance(s) de trouver le nombre.\n".
            ↪ format(chance))

if(nbreUser != resultat):
    print("Historique de vos tentatives: {0}". format(essai))
    print("Vous avez échoué après {0} tentattives !!!\n". format(len(essai)))
    print("Le nombre à déviner était {0} !!!\n". format(resultat))


print("Good !!!\n")
```

3.7 EXERCICE

EXO 1 Ecrire le programme python qui permet de créer la liste ci-dessous :

```
[['a1', 'b1', 'c1', 'd1', 'e1', 'f1', 'g1', 'h1'],
 ['a2', 'b2', 'c2', 'd2', 'e2', 'f2', 'g2', 'h2'],
 ['a3', 'b3', 'c3', 'd3', 'e3', 'f3', 'g3', 'h3'],
 ['a4', 'b4', 'c4', 'd4', 'e4', 'f4', 'g4', 'h4'],
 ['a5', 'b5', 'c5', 'd5', 'e5', 'f5', 'g5', 'h5'],
 ['a6', 'b6', 'c6', 'd6', 'e6', 'f6', 'g6', 'h6'],
 ['a7', 'b7', 'c7', 'd7', 'e7', 'f7', 'g7', 'h7'],
 ['a8', 'b8', 'c8', 'd8', 'e8', 'f8', 'g8', 'h8']]
```

— Gestion avancée d'un carnet de contacts. Créer une liste contacts contenant des dictionnaires de la forme :

```
{
    "nom": "Dupont",
    "telephone": "699000111",
    "email": "dupont@gmail.com"
}
```

Écrire les fonctions suivantes : *ajouter_contact(contacts)*, *supprimer_contact(contacts, nom)*, *rechercher_contact(contacts, nom)*, *afficher_contacts(contacts)*. NB : vous devez empêcher l'ajout de deux contacts ayant le même numéro de téléphone.

— **Analyse statistique d'un texte :** On vous demande d'analyser un texte en utilisant les dictionnaires et les sets. TAF :

  — Demander à l'utilisateur de saisir un texte (fonction $frequence_mots(texte)$).

  — *Créer un dictionnaire* frequences *qui contient le nombre d'occurrences de chaque mot (fonction* $\text{mot}_le_plus_frequent(frequences)$).

  — Créer un set des mots distincts (fonction $afficher_statistiques(frequences)$).

— **Mini-module de calcul mathématique : Créer et utiliser un module Python** Créer un fichier *calculs.py* contenant les fonctions :

  — addition(a, b)
  — soustraction(a, b)
  — multiplication(a, b)
  — division(a, b) (gérer division par zéro)

  Créer un fichier *main.py* qui :

  — importe le module calculs
  — utilise les fonctions selon le choix de l'utilisateur

NB : Utiliser au moins deux formes d'import (import calculs et from calculs import ...).

— **Organisation d'un package de gestion scolaire :** Structurer un projet Python sous forme de package. Créer l'arborescence suivante :

```
gestion_scolaire/
├── __init__.py
├── etudiants.py
├── notes.py
└── affichage.py
```

— Dans *etudiants.py* : fonctions de gestion des étudiants (ajout, recherche).
— Dans *notes.py* : fonctions de calcul de moyenne et de mention.
— Dans *affichage.py* : fonctions d'affichage formaté.
— Créer un *main.py* qui utilise le package *gestion_scolaire*.

— **Système de gestion d'inventaire (complet) :** L'objectif est de mettre en œuvre plusieurs notions dans un mini-projet structuré. TAF :

1. Gérer un inventaire sous forme :

```
nventaire = {
            "stylo": 20,
            "cahier": 15,
            "ordinateur": 5
        }
```

2. Créer un module *gestion_inventaire.py* avec les fonctions : *ajouter_produit(), supprimer_produit(), modifier_quantite(), afficher¡nventaire*()
3. Créer un fichier *main.py* qui appelle les fonctions du module.

Chapitre 4

Module 4 : Programmation Orientée Objet et les Décorateurs

Objectifs :

— Maîtriser les concepts POO
— Comprendre les concepts de classes, objets, héritage, encapsulation, polymorphisme.
— Enrichier le comportement de vos fontions avec les décorateurs.

Contenu :

- Création de classes, attributs, méthodes.
- Constructeur (__init__), méthodes spéciales (__str__, __repr__).
- Héritage simple et multiple.
- Notions d'abstraction et interfaces.
- Fonctions améliorées avec les décorateurs.

La programmation orientée objet (POO) en Python peut sembler complexe au début, mais elle devient vite un atout indispensable pour écrire du code clair et efficace.

À mon avis, comprendre la POO est comme apprendre à faire du vélo : une fois que vous avez compris le mécanisme, vous ne l'oubliez plus. Dans ce guide, je vais vous accompagner pas à pas pour explorer les concepts essentiels comme les classes, les objets, l'héritage et bien d'autres. Nous verrons ensemble comment ces notions s'appliquent concrètement en Python, avec des exemples simples et pratiques.

Alors, prêt à plonger dans le monde fascinant de la POO en Python ? Accrochez votre casque, c'est parti !

4.1 Introduction à la POO

La **programmation orientée objet**, ou "**POO**", c'est une façon de programmer en imaginant que tout ce qu'on manipule dans un programme est comme un objet de la vie réelle.

Imagine : Tu veux créer un jeu vidéo où il y a des voitures. Plutôt que de tout écrire pour chaque voiture, tu commences par créer un modèle ou une recette qui décrit ce qu'est une voiture.

Dans ta recette de voiture, tu peux dire :

— Une voiture a des caractéristiques comme une couleur, une marque, ou une vitesse.
— Une voiture peut faire des choses comme rouler, klaxonner, ou freiner.

```
class Voiture:
    def __init__(self, couleur, marque):
        self.couleur = couleur
        self.marque = marque
        self.vitesse = 0

    def rouler(self):
        self.vitesse += 10
        print(f"La {self.marque} roule à {self.vitesse} km/h.")

    def freiner(self):
        self.vitesse = max(0, self.vitesse - 10)
        print(f"La {self.marque} freine et roule à {self.vitesse} km/h.")
```

Ensuite, tu peux fabriquer plein de voitures à partir de ta recette. Ces voitures, on les appelle des **objets**.

```
ma_voiture = Voiture("rouge", "Ferrari")
ta_voiture = Voiture("bleue", "Renault")

ma_voiture.rouler()  # La Ferrari roule à 10 km/h.
ta_voiture.rouler()  # La Renault roule à 10 km/h.
```

Chaque voiture est indépendante, mais elles utilisent la même recette pour exister.

Les avantages :

- Organisation : Tu ranges tout ce qui concerne les voitures au même endroit.
- Réutilisation : Une seule recette peut créer des milliers d'objets.
- Évolution : Si tu veux ajouter un toit ouvrant à toutes les voitures, tu modifies juste la recette.

C'est comme jouer avec des LEGO : tu crées des pièces (classes) et après tu peux construire tout ce que tu veux (objets) ! Imaginez devoir gérer une flotte de véhicules dans un programme. Sans la POO, vous auriez une multitude de variables et de fonctions éparpillées. Avec la programmation orientée objet, vous pouvez créer une classe Véhicule et ensuite instancier autant d'objets Véhicule que nécessaire, chacun avec ses propres caractéristiques.

Python supporte pleinement la programmation orientée objet. Même si vous n'écrivez pas explicitement de classes, tout est objet en Python. Les chaînes de caractères, les listes, les dictionnaires sont tous des objets avec des méthodes que vous utilisez déjà au quotidien.

4.2 Les Concepts de Base de la POO

Cette section est consacrée à vous exposer les éléments clés de la programmation orientée objet en Python.

4.2.1 Les classes et les objets

Une classe est comme un plan ou un moule qui définit les attributs et les méthodes communes à tous les objets de ce type. Un objet est une instance de cette classe, une réalisation concrète.

Par exemple :

```
class Personne:
    pass

individu = Personne()
```

Ici, *Personne* est une classe et *individu* est un **objet** (ou **instance**) de cette classe.

4.2.2 Les attributs et les méthodes

Les attributs sont les données qui décrivent l'état de l'objet, tandis que les méthodes sont les fonctions qui définissent son comportement.

```
class Personne:
    def __init__(self, nom, âge):
        self.nom = nom  # Attribut
        self.âge = âge  # Attribut

    def se_presenter(self):  # Méthode
        print(f"Bonjour, je m'appelle {self.nom} et j'ai {self.âge} ans.")
```

4.2.3 L'instanciation d'un objet

Pour créer un objet à partir d'une classe, on utilise la syntaxe suivante :

```
moi = Personne("Alice", 30)
moi.se_presenter()  # Affiche "Bonjour, je m'appelle Alice et j'ai 30 ans."
```

4.2.4 L'encapsulation

L'encapsulation consiste à protéger les attributs et méthodes internes d'une classe en limitant leur accès depuis l'extérieur. En Python, on peut indiquer qu'un attribut est privé en le précédant de deux underscores __.

```python
class CompteBancaire:
    def __init__(self, solde):
        self.__solde = solde  # Attribut privé

    def deposer(self, montant):
        self.__solde += montant

    def retirer(self, montant):
        if montant <= self.__solde:
            self.__solde -= montant
        else:
            print("Fonds insuffisants")

    def afficher_solde(self):
        print(f"Le solde est de {self.__solde} euros.")
```

Essayer d'accéder directement à __solde depuis l'extérieur ne fonctionnera pas :

```python
compte = CompteBancaire(1000)
print(compte.__solde)  # Provoque une erreur d'attribut
```

4.2.5 Le polymorphisme

Le polymorphisme permet d'utiliser une même interface pour différents objets, ce qui rend le code plus flexible.

```python
class Animal:
    def parler(self):
        pass

class Chien(Animal):
    def parler(self):
        print("Wouf!")

class Chat(Animal):
    def parler(self):
        print("Miaou!")

def faire_parler(animal):
    animal.parler()

faire_parler(Chien())  # Affiche "Wouf!"
faire_parler(Chat())   # Affiche "Miaou!"
```

4.2.6 L'héritage

L'héritage permet à une classe enfant d'hériter des attributs et méthodes d'une classe parent, favorisant la réutilisation du code.

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

    def se_deplacer(self):
        print(f"Le véhicule {self.marque} se déplace.")

class Voiture(Vehicule):
    def __init__(self, marque, modèle):
        super().__init__(marque)
        self.modèle = modèle

    def se_deplacer(self):
        print(f"La voiture {self.marque} {self.modèle} roule sur la route.")

ma_voiture = Voiture("Tesla", "Model S")
ma_voiture.se_deplacer()  # Affiche "La voiture Tesla Model S roule sur la route."
```

Dans cet exemple, *Voiture* hérite de *Vehicule* et surcharge la méthode *se_deplacer()*.

4.3 Les attributs de classe et d'instance

Les attributs de classe sont partagés entre toutes les instances, tandis que les attributs d'instance sont propres à chaque objet.

```python
class Compte:
    banque = "Banque Nationale"  # Attribut de classe

    def __init__(self, titulaire):
        self.titulaire = titulaire  # Attribut d'instance

compte1 = Compte("Alice")
compte2 = Compte("Bob")

print(compte1.banque)  # Affiche "Banque Nationale"
print(compte2.banque)  # Affiche "Banque Nationale"
```

4.4 Créer sa Première Classe en Python

Passons à la pratique ! Quoi de mieux que de créer une **classe** qui nous sera réellement utile en tant que personne en charge d'automatiser des procédures ?

Imaginez que vous devez gérer plusieurs services sur vos serveurs, comme *Nginx*, *Apache* ou *Docker*. À mon avis, avoir une classe qui représente un service et permet de contrôler son état serait un véritable gain de temps.

Voici comment définir une classe en Python :

```
class Service:
    pass # La classe est vide et ne fait rien
```

Bon, ce n'est pas très passionnant pour le moment. Ajoutons quelques attributs et méthodes pour rendre cette classe utile. La méthode __init__ est le constructeur de la classe. Elle initialise les attributs de l'objet lors de sa création.

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

Ajoutons des méthodes pour démarrer, arrêter et redémarrer le service.

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

    def demarrer(self):
        print(f"Démarrage du service {self.nom}")
        # Code pour démarrer le service

    def arreter(self):
        print(f"Arrêt du service {self.nom}")
        # Code pour arrêter le service

    def redemarrer(self):
        print(f"Redémarrage du service {self.nom}")
        # Code pour redémarrer le service
```

Créons une instance de notre classe pour gérer Nginx.

```
nginx = Service("nginx")
nginx.demarrer()    # Affiche "Démarrage du service nginx"
nginx.arreter()     # Affiche "Arrêt du service nginx"
nginx.redemarrer()  # Affiche "Redémarrage du service nginx"
```

Pour rendre notre classe fonctionnelle, utilisons le module *subprocess* pour exécuter des commandes système.

```
import subprocess

class Service:
    def __init__(self, nom):
        self.nom = nom

    def demarrer(self):
        print(f"Démarrage du service {self.nom}")
        subprocess.run(["sudo", "systemctl", "start", self.nom])

    def arreter(self):
        print(f"Arrêt du service {self.nom}")
```

```
        subprocess.run(["sudo", "systemctl", "stop", self.nom])

    def redemarrer(self):
        print(f"Redémarrage du service {self.nom}")
        subprocess.run(["sudo", "systemctl", "restart", self.nom])
```

Ajoutons une méthode pour vérifier si le service est actif.

```
def statut(self):
    result = subprocess.run(["systemctl", "is-active", self.nom],
    ↪  capture_output=True, text=True)
    if result.stdout.strip() == "active":
        print(f"Le service {self.nom} est actif.")
        return True
    else:
        print(f"Le service {self.nom} n'est pas actif.")
        return False
```

Imaginons que nous devons gérer plusieurs **services** simultanément.

```
services = ["nginx", "docker", "mysql"]
for nom_service in services:
    service = Service(nom_service)
    if not service.statut():
        service.demarrer()
```

Ce script vérifie si chaque service est actif et le démarre si nécessaire. À mon avis, la **programmation orientée objet** rend le code plus modulable et réutilisable. En créant une classe générique pour les services, on évite de répéter du code et on facilite la maintenance. De plus, si demain on doit ajouter de nouvelles fonctionnalités, il suffit de les implémenter une fois dans la classe.

Il est important de gérer les erreurs potentielles, par exemple si le service n'existe pas.

```
def demarrer(self):
    print(f"Démarrage du service {self.nom}")
    try:
        subprocess.run(["sudo", "systemctl", "start", self.nom], check=True)
    except subprocess.CalledProcessError:
        print(f"Erreur : Le service {self.nom} n'a pas pu être démarré.")
```

En tant qu'AdminSys, l'automatisation est notre meilleur ami. En utilisant la **POO en Python**, nous pouvons créer des outils puissants pour simplifier nos tâches quotidiennes.

4.5 Les Méthodes Spéciales

Vous avez peut-être déjà entendu parler des **méthodes spéciales en Python**, aussi connues sous le nom de *dunder methods* (pour double underscore). À mon avis, ces méthodes apportent une touche de magie à nos classes en nous permettant de définir le comportement des objets avec les fonctionnalités intégrées de Python.

Les méthodes spéciales sont des fonctions prédéfinies par Python qui ont des noms entourés de doubles underscores, comme *__init__*, *__str__*, *__repr__*, *__add__*, etc. Elles permettent à nos objets d'interagir avec les opérateurs et les fonctions intégrées du langage.

Rappelez-vous de notre classe *Service* que nous avions créée pour gérer les services sur nos serveurs. Nous utilisions des **méthodes spéciales** pour rendre cette classe encore plus puissante et intuitive.

La Méthode __str__

La méthode __str__ définit comment un objet est représenté sous forme de chaîne de caractères, notamment lorsqu'on utilise la fonction print().

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

    def __str__(self):
        return f"Service: {self.nom}"
```

```
nginx = Service("nginx")
print(nginx)  # Affiche "Service: nginx"
```

La Méthode __eq__

Maintenant, chaque fois que nous imprimons un objet Service, nous obtenons une description claire et lisible. Si nous voulons comparer deux services pour savoir s'ils sont identiques, nous pouvons utiliser la méthode __eq__.

```
def __eq__(self, autre):
    return self.nom == autre.nom
```

```
service1 = Service("nginx")
service2 = Service("nginx")
service3 = Service("docker")

print(service1 == service2)  # Affiche True
print(service1 == service3)  # Affiche False
```

Ainsi, l'opérateur == compare directement les noms des services.

La Méthode __iter__

Supposons que nous ayons une classe *GestionnaireServices* qui gère plusieurs services. Nous pouvons la rendre itérable pour parcourir facilement les services.

```python
class GestionnaireServices:
    def __init__(self):
        self.services = []
        self._index = 0

    def ajouter_service(self, service):
        self.services.append(service)

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < len(self.services):
            service = self.services[self._index]
            self._index += 1
            return service
        else:
            raise StopIteration
```

```python
gestionnaire = GestionnaireServices()
gestionnaire.ajouter_service(Service("nginx"))
gestionnaire.ajouter_service(Service("docker"))
gestionnaire.ajouter_service(Service("mysql"))

for service in gestionnaire:
    print(service)
```

Ce code affiche :

```
Service: nginx
Service: docker
Service: mysql
```

La Méthode __add__

Nous pouvons définir comment additionner deux objets *GestionnaireServices* en utilisant la méthode __add__.

```python
def __add__(self, autre):
    nouveau_gestionnaire = GestionnaireServices()
    nouveau_gestionnaire.services = self.services + autre.services
    return nouveau_gestionnaire
```

```python
gestionnaire1 = GestionnaireServices()
gestionnaire1.ajouter_service(Service("nginx"))

gestionnaire2 = GestionnaireServices()
```

```
gestionnaire2.ajouter_service(Service("docker"))

gestionnaire_combiné = gestionnaire1 + gestionnaire2

for service in gestionnaire_combiné:
    print(service)
```

Ce code affiche :

```
Service: nginx
Service: docker
```

Nous avons ainsi combiné deux gestionnaires en un seul, grâce à l'opérateur +.

La Méthode __repr__

La méthode __repr__ fournit une représentation non ambiguë de l'objet, utile pour le débogage.

```
def __repr__(self):
    return f"Service(nom='{self.nom}')"
```

Exemple d'utilisation :

```
print(repr(nginx))  # Affiche "Service(nom='nginx')"
```

La Méthode __del__

Le destructeur est appelé lorsque l'objet est détruit. Parfois cela peut être utile pour libérer des ressources comme des fichiers ou des disques virtuels ouverts.

```
def __del__(self):
    print("Détruit le service", self.nom)
```

La Méthode __call__

Si nous souhaitons pouvoir appeler un objet comme une fonction, nous pouvons utiliser la méthode __call__.

```
def __call__(self, action):
    if action == "démarrer":
        self.demarrer()
    elif action == "arrêter":
        self.arreter()
    elif action == "redémarrer":
        self.redemarrer()
```

```
nginx("démarrer")     # Démarre le service nginx
nginx("redémarrer")   # Redémarre le service nginx
```

Autres méthodes spéciales utiles

- __len__ : Pour définir le comportement de la fonction *len()*.
- __getitem__ : Pour accéder aux éléments via l'indexation.
- __contains__ : Pour vérifier l'appartenance avec l'opérateur *in*.

Exemple avec __len__ et __getitem__

Ajoutons ces méthodes à notre classe *GestionnaireServices* :

```
def __len__(self):
    return len(self.services)

def __getitem__(self, index):
    return self.services[index]
```

Exemple d'utilisation :

```
print(len(gestionnaire))        # Affiche le nombre de services
print(gestionnaire[0])          # Affiche le premier service
print(Service("nginx") in gestionnaire)  # Vérifie si nginx est dans le
↪  gestionnaire
```

Voilà qui boucle cette partie sur la POO en Python. Dans la section suivante, nous verrons un concept plus avancé que sont les **décorateurs** .

4.6 Les décorateurs

Les décorateurs Python, c'est un peu comme ajouter une couche de vernis à vos fonctions : ils permettent de modifier ou d'enrichir leur comportement, sans toucher à leur code. En gros, ils prennent une fonction, l'enrobent de logique supplémentaire et renvoient une version boostée. Que vous vouliez mesurer le temps d'exécution d'une méthode, vérifier des permissions ou simplement ajouter un peu de magie, les décorateurs sont vos alliés. Et le meilleur ? Une fois que vous les maîtrisez, vous ne pourrez plus vous en passer !

4.6.1 Comment fonctionnent les fonctions en Python

Pour bien comprendre les décorateurs, il faut d'abord se plonger dans un concept clé de Python : les fonctions sont des objets de première classe. Ça veut dire quoi ? Tout simplement que les

fonctions en Python sont comme n'importe quel autre objet : elles peuvent être stockées dans des variables, passées en argument à d'autres fonctions, ou même être retournées comme résultats.

Prenons un exemple simple pour illustrer cela :

```
def dire_bonjour():
    print("Bonjour !")

# Assigner la fonction à une variable
salutation = dire_bonjour

# Appeler la fonction via la nouvelle variable
salutation()
```

Ici, *salutation* devient un alias de *dire_bonjour*. Les deux références pointent vers le même objet fonction. On peut aussi passer une fonction comme argument à une autre. C'est ici que la magie des **décorateurs** commence à apparaître.

```
def executer_fonction(func):
    print("Je vais exécuter la fonction passée en argument...")
    func()

# Passer une fonction comme argument
executer_fonction(dire_bonjour)
```

Résultat ? La fonction *dire_bonjour* est exécutée depuis *executer_fonction*. Cela ouvre la voie à des opérations comme ajouter de la logique avant ou après l'exécution d'une fonction.

Une fonction peut aussi en retourner une autre. Ce comportement est crucial pour comprendre comment les décorateurs modifient des fonctions.

```
def faire_salut():
    def message():
        print("Salut !")
    return message

# Appeler faire_salut et récupérer une fonction
fonction_retournee = faire_salut()
fonction_retournee()
```

Ici, *faire_salut* retourne une nouvelle fonction, message, que l'on peut appeler indépendamment.

Maintenant, combinons ces idées : une fonction peut à la fois recevoir une autre fonction et en retourner une. C'est littéralement ce que fait un décorateur.

```
def mon_decorateur(func):
    def wrapper():
        print("Avant la fonction")
        func()
        print("Après la fonction")
    return wrapper
```

```
def dire_au_revoir():
    print("Au revoir !")

# Décorer "à la main"
decorated = mon_decorateur(dire_au_revoir)
decorated()
```

Ici, *mon_decorateur* enveloppe *dire_au_revoir* dans un nouveau comportement. La fonction retournée, *wrapper*, contient les actions avant et après.

Comprendre que les fonctions sont des objets de première classe est essentiel pour utiliser les décorateurs. Cela permet de saisir comment ils interceptent et enrichissent des fonctions existantes. Une fois cette idée en tête, les décorateurs deviennent un outil incroyablement puissant pour optimiser votre code !

4.7 Créer un décorateur simple

Maintenant que l'on sait que les fonctions en Python peuvent être passées et retournées comme des objets, créons un décorateur basique. Un décorateur est, en gros, une fonction qui prend une autre fonction en entrée, lui ajoute du code avant ou après et retourne une nouvelle fonction "enrichie".

4.7.1 Structure de base d'un décorateur

Voici la structure classique d'un **décorateur** :

```
def mon_decorateur(func):
    def wrapper():
        # Code avant l'exécution de la fonction originale
        print("Avant la fonction")
        func()
        # Code après l'exécution de la fonction originale
        print("Après la fonction")
    return wrapper
```

Dans cet exemple, *mon_decorateur* enveloppe une fonction avec un "emballage" (*wrapper*) contenant du code supplémentaire.

4.7.2 Utilisation d'un décorateur

Supposons que nous ayons une fonction *dire_bonjour* que nous voulons enrichir. Voici comment l'appliquer manuellement :

```
def dire_bonjour():
    print("Bonjour tout le monde !")

# Décorer la fonction "à la main"
```

```
decorated = mon_decorateur(dire_bonjour)
decorated()
```

Cela produit :

```
Avant la fonction
Bonjour tout le monde !
Après la fonction
```

Plutôt cool, non ? Mais il y a une façon encore plus simple d'appliquer un décorateur : utiliser la syntaxe @.

4.7.3 La syntaxe @ pour simplifier

Python fournit une syntaxe dédiée pour appliquer un décorateur. Il suffit d'utiliser le symbole @ avant le nom du décorateur :

```
@mon_decorateur
def dire_bonjour():
    print("Bonjour tout le monde !")

# Appeler directement la fonction
dire_bonjour()
```

Cette syntaxe est équivalente à l'exemple précédent, mais bien plus lisible. Décorer des fonctions avec des arguments

Et si notre fonction avait des arguments ? Pas de problème, il suffit de passer ces arguments à *func* dans le *wrapper* :

```
def mon_decorateur(func):
    def wrapper(*args, **kwargs):
        print("Avant la fonction")
        result = func(*args, **kwargs)
        print("Après la fonction")
        return result
    return wrapper

@mon_decorateur
def dire_bonjour_avec_nom(nom):
    print(f"Bonjour, {nom} !")

dire_bonjour_avec_nom("Alice")
```

Sortie :

```
Avant la fonction
Bonjour tout le monde !
Après la fonction
```

Le **args* et ***kwargs* permettent de gérer n'importe quel nombre d'arguments positionnels ou nommés. C'est indispensable pour des fonctions génériques.

4.7.4 Pourquoi utiliser des décorateurs ?

Créer un décorateur permet de réutiliser du code commun, comme ajouter des logs, mesurer le temps d'exécution, ou gérer des permissions. Par exemple, voici un décorateur pour mesurer la durée d'une fonction :

```
import time

def chronometre(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Durée : {end - start:.4f} secondes")
        return result
    return wrapper

@chronometre
def traitement_lourd():
    time.sleep(2)  # Simule un calcul lourd
    print("Traitement terminé !")

traitement_lourd()
```

Sortie :

```
Traitement terminé !
Durée : 2.0021 secondes
```

4.7.5 Empilement de décorateurs

Saviez-vous que vous pouvez appliquer plusieurs décorateurs à une même fonction ? Oui, Python permet ce qu'on appelle l'empilement de décorateurs, et c'est à la fois pratique et puissant. L'idée est simple : chaque décorateur s'applique successivement, dans l'ordre où ils sont définis, pour enrichir votre fonction étape par étape.

Comment empiler des décorateurs

Pour empiler des décorateurs, il suffit de les écrire les uns au-dessus des autres, avec le symbole @. Voici un exemple simple :

```
def decorateur_1(func):
    def wrapper(*args, **kwargs):
        print("Décorateur 1 : Avant la fonction")
```

```
        result = func(*args, **kwargs)
        print("Décorateur 1 : Après la fonction")
        return result
    return wrapper

def decorateur_2(func):
    def wrapper(*args, **kwargs):
        print("Décorateur 2 : Avant la fonction")
        result = func(*args, **kwargs)
        print("Décorateur 2 : Après la fonction")
        return result
    return wrapper

@decorateur_1
@decorateur_2
def ma_fonction():
    print("Exécution de la fonction")

ma_fonction()
```

Sortie :

```
Décorateur 1 : Avant la fonction
Décorateur 2 : Avant la fonction
Exécution de la fonction
Décorateur 2 : Après la fonction
Décorateur 1 : Après la fonction
```

Ordre d'exécution

L'ordre des décorateurs est important. Dans l'exemple ci-dessus, *decorateur_2* est appliqué avant *decorateur_1* car Python traite les décorateurs du bas vers le haut.

Autrement dit :

1. *ma_fonction* est décorée par *decorateur_2*.
2. Puis, le résultat est à nouveau décoré par $decorateur_1$.

Exemple pratique : logs et chronométrage

Imaginons que vous souhaitiez à la fois ajouter des logs et mesurer le temps d'exécution d'une fonction. Voici comment faire :

```
import time

def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Appel de la fonction {func.__name__} avec {args} et {kwargs}")
        return func(*args, **kwargs)
    return wrapper
```

```python
def chronometre(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Temps d'exécution de {func.__name__} : {end - start:.4f}
        ↪  secondes")
        return result
    return wrapper

@log_execution
@chronometre
def traitement_lourd(n):
    time.sleep(n)
    print("Traitement terminé !")

traitement_lourd(2)
```

Sortie :

```
Appel de la fonction wrapper avec (2,) et {}
Temps d'exécution de traitement_lourd : 2.0021 secondes
Traitement terminé !
```

Étendre l'empilement aux méthodes de classes

Les **décorateurs empilés** fonctionnent aussi très bien sur des méthodes de classes. Par exemple, vous pouvez combiner des décorateurs pour vérifier les permissions et ajouter des logs :

```python
def verifier_permissions(func):
    def wrapper(self, *args, **kwargs):
        if not self.est_admin:
            print("Accès refusé.")
            return
        return func(self, *args, **kwargs)
    return wrapper

def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Exécution de {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class Systeme:
    def __init__(self, est_admin):
        self.est_admin = est_admin

    @verifier_permissions
    @log_execution
    def supprimer_donnees(self):
        print("Données supprimées.")
```

```
utilisateur = Systeme(est_admin=False)
utilisateur.supprimer_donnees()

admin = Systeme(est_admin=True)
admin.supprimer_donnees()
```

Sortie :

```
Accès refusé.
Exécution de supprimer_donnees
Données supprimées.
```

4.7.6 Décorateurs dans des classes

Les décorateurs ne sont pas réservés aux fonctions globales. Vous pouvez aussi les appliquer aux méthodes et aux propriétés d'une classe pour ajouter des comportements ou simplifier la gestion de vos données. C'est une fonctionnalité puissante pour structurer un code objet propre et réutilisable.

En Python, il existe des **décorateurs** intégrés spécifiques pour gérer les méthodes dans une classe :

- *@staticmethod* : indique qu'une méthode n'a pas accès à l'instance ou à la classe elle-même. C'est une fonction "indépendante" dans une classe.
- *@classmethod* : transforme une méthode pour qu'elle reçoive la classe comme premier argument au lieu de l'instance (*cls* au lieu de *self*).

Exemple avec *@staticmethod* Une méthode statique est utile pour des fonctions qui ne dépendent pas des attributs de l'instance ou de la classe.

```
class MathUtils:
    @staticmethod
    def addition(a, b):
        return a + b

# Appel direct sans créer d'instance
print(MathUtils.addition(3, 5))  # Sortie : 8
```

Exemple avec *@classmethod* Une méthode de classe permet de travailler avec la classe elle-même.

```
class Compteur:
    total = 0

    @classmethod
    def incrementer(cls):
        cls.total += 1

# Utilisation de la méthode de classe
Compteur.incrementer()
```

```
Compteur.incrementer()
print(Compteur.total)  # Sortie : 2
```

4.7.7 Décorateurs pour les propriétés

Un autre usage fréquent des décorateurs dans les classes est la gestion des propriétés avec *@property*. Cela permet de définir des getters et setters sans appeler explicitement des méthodes.

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

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nouveau_nom):
        if not nouveau_nom:
            raise ValueError("Le nom ne peut pas être vide.")
        self._nom = nouveau_nom

# Utilisation
p = Personne("Alice")
print(p.nom)  # Sortie : Alice
p.nom = "Bob"
print(p.nom)  # Sortie : Bob
```

Ici, *@property* permet d'accéder à l'attribut *_nom* comme si c'était une propriété publique tout en gardant le contrôle sur sa validation.

4.7.8 Les décorateurs de Click

Plus haut dans cette section, nous avons vu comment les décorateurs enrichissent les fonctions. *Click* exploite cette puissance pour simplifier la création de **CLI** (Command Line Interface). En voici la liste :

1. *@click.command* : transforme une fonction en commande CLI.
2. *@click.option* : ajoute des options avec préfixe (–nom).
3. *@click.argument* : gère les arguments positionnels.
4. *@click.group* : regroupe plusieurs commandes sous une même interface.

```
import click

@click.command()
@click.option("--nom", default="utilisateur", help="Le nom à saluer.")
```

```python
def salut(nom):
    click.echo(f"Bonjour, {nom} !")

if __name__ == "__main__":
    salut()
```

4.7.9 Conclusion sur les décorateurs

Les **décorateurs** Python sont une solution élégante pour enrichir vos fonctions et simplifier votre code. Qu'il s'agisse de valider des données, de mesurer des performances ou de structurer des **CLI** avec des outils comme Click, ils offrent une modularité et une lisibilité incomparables.

En maîtrisant les décorateurs, vous pouvez écrire un code plus propre et plus maintenable, tout en réduisant les répétitions inutiles. Les perspectives sont vastes : frameworks, outils d'automatisation ou optimisation de vos propres projets. Lancez-vous et découvrez à quel point les décorateurs peuvent transformer votre manière de coder !

TP :

- Gestion d'une bibliothèque (classes Livre, Auteur, Bibliothèque).
- Mini simulateur de banque (clients, comptes, transactions).

Chapitre 5

Module 5 : Gestion des exceptions et des fichiers

Objectifs :

— Maîtriser la gestion des exceptions en Python.
— Maitriser la gestion des fichiers avec python.

Contenu :

- Gestion des erreurs avec try, except, finally, raise.
- Lecture/écriture dans des fichiers texte, CSV, JSON.

5.1 Maîtriser la gestion des exceptions en Python

La gestion des exceptions est un aspect essentiel en Python qui permet de gérer les erreurs de manière élégante et efficace. Comprendre comment fonctionnent les exceptions aide à écrire des programmes plus robustes et à offrir une meilleure expérience utilisateur.

5.1.1 Pourquoi gérer les exceptions ?

Gérer les **exceptions** permet d'anticiper les erreurs potentielles et d'éviter que le programme ne se termine brutalement. Sans gestion des exceptions, une erreur non traitée provoque l'arrêt immédiat du programme, souvent avec un message d'erreur peu compréhensible pour l'utilisateur final. En capturant et en traitant les exceptions, il est possible de :

- Prévenir les plantages du programme.
- Fournir des messages d'erreur clairs et utiles.
- Libérer proprement les ressources (fichiers, connexions réseau, etc.).
- Maintenir le flux normal du programme en cas d'erreurs mineures.

5.1.2 Comprendre les types d'exceptions

Python dispose de nombreux types d'exceptions intégrées, chacune correspondant à une erreur spécifique. Voici quelques-unes des plus courantes :

- ZeroDivisionError : Tentative de division par zéro.
- TypeError : Opération sur un objet d'un type inapproprié.
- ValueError : Valeur ayant le bon type mais une valeur inappropriée.
- IndexError : Indice de séquence hors de portée.
- KeyError : Clé non trouvée dans un dictionnaire.
- AttributeError : Tentative d'accès à un attribut inexistant d'un objet.
- ImportError : Échec d'importation d'un module.

Exemples :

```
# ZeroDivisionError
print(10 / 0)  # Provoque une ZeroDivisionError

# TypeError
print('5' + 3)  # Provoque une TypeError

# ValueError
int('abc')  # Provoque une ValueError

# IndexError
liste = [1, 2, 3]
print(liste[5])  # Provoque une IndexError

# KeyError
dictionnaire = {'a': 1, 'b': 2}
print(dictionnaire['c'])  # Provoque une KeyError
```

Comprendre ces exceptions permet de les anticiper et de les gérer correctement dans le code.

5.1.3 Utiliser try et except

Le bloc *try...except* permet de capturer les exceptions et de définir une réponse appropriée. Le code susceptible de provoquer une exception est placé dans le bloc *try* et le traitement de l'erreur dans le bloc *except*.

Syntaxe de base :

```
try:
    # Code susceptible de provoquer une exception
except NomDeLException:
    # Code à exécuter en cas d'exception
```

Exemple :

```python
try:
    num = int(input("Entrez un nombre : "))
    resultat = 10 / num
    print(f"Résultat : {resultat}")
except ZeroDivisionError:
    print("Erreur : division par zéro.")
except ValueError:
    print("Erreur : veuillez entrer un nombre valide.")
```

Dans cet exemple :

- Si l'utilisateur entre zéro, une *ZeroDivisionError* est levée et gérée.
- Si l'utilisateur entre une valeur non numérique, une *ValueError* est levée et gérée.
- Dans les deux cas, le programme affiche un message d'erreur clair au lieu de se terminer brutalement.

5.1.4 Capturer plusieurs exceptions

Il est possible de gérer plusieurs exceptions dans un même bloc *except* en les regroupant dans un tuple. Cela permet de traiter de la même manière plusieurs types d'exceptions.

Exemple :

```python
try:
    num = int(input("Entrez un nombre : "))
    resultat = 10 / num
except (ZeroDivisionError, ValueError) as e:
    print(f"Erreur : {e}")
```

Ici, si une *ZeroDivisionError* ou une *ValueError* est levée, le même bloc *except* les gère et le message d'erreur associé est affiché. Utiliser else avec *try...except*.

Le bloc *else* s'exécute si aucune exception n'est levée dans le bloc *try*. Il est utile pour exécuter du code qui doit seulement s'exécuter si tout s'est bien passé dans le *try*.

Exemple :

```python
try:
    num = int(input("Entrez un nombre : "))
except ValueError:
    print("Erreur : veuillez entrer un nombre valide.")
else:
    print(f"Vous avez entré le nombre {num}.")
```

Dans cet exemple, le message de confirmation n'est affiché que si la conversion en entier s'est déroulée sans erreur.

5.1.5 Le bloc finally

Le bloc finally est exécuté qu'une exception ait été levée ou non. Il est utile pour effectuer des opérations de nettoyage ou libérer des ressources, comme fermer un fichier ou une connexion.

Exemple :

```
try:
    num = int(input("Entrez un nombre : "))
    resultat = 10 / num
except ZeroDivisionError:
    print("Erreur : division par zéro.")
except ValueError:
    print("Erreur : veuillez entrer un nombre valide.")
finally:
    print("Fin de l'opération.")
```

Ici, le message *"Fin de l'opération."* est toujours affiché, que le programme ait rencontré une exception ou non.

5.1.6 Lever des exceptions avec raise

Il est possible de lever explicitement une exception en utilisant le mot-clé raise. Cela permet de signaler qu'une condition anormale s'est produite et de transférer le contrôle au bloc except approprié.

Exemple :

```
def verifier_age(age):
    if age < 0:
        raise ValueError("L'âge ne peut pas être négatif.")
    elif age < 18:
        print("Vous êtes mineur.")
    else:
        print("Vous êtes majeur.")

try:
    verifier_age(-5)
except ValueError as e:
    print(f"Erreur : {e}")
```

Dans cet exemple, la fonction *verifier_age* lève une *ValueError* si l'âge est négatif et le bloc *try...except* capture cette exception pour afficher un message d'erreur.

5.1.7 Créer ses propres exceptions

Il est possible de définir des **exceptions personnalisées** en créant des classes qui héritent de la classe *Exception*. Cela permet de créer des erreurs spécifiques à l'application ou au domaine.

Exemple :

```python
class TemperatureError(Exception):
    """Exception levée lorsque la température est invalide."""
    pass

def verifier_temperature(temp):
    if temp < -273.15:
        raise TemperatureError("La température ne peut pas être inférieure au zéro
        ↪ absolu.")

try:
    verifier_temperature(-300)
except TemperatureError as e:
    print(f"Erreur : {e}")
```

Dans cet exemple, une exception personnalisée *TemperatureError* est créée pour gérer des erreurs liées à la température.

5.1.8 Les assertions

Les assertions permettent de vérifier qu'une condition est vraie pendant l'exécution du programme. Si la condition est fausse, une *AssertionError* est levée. Elles sont principalement utilisées pendant le développement pour détecter des erreurs de logique.

Exemple :

```python
def calculer_factorielle(n):
    assert n >= 0, "Le nombre doit être positif ou nul."
    if n == 0:
        return 1
    else:
        return n * calculer_factorielle(n - 1)

print(calculer_factorielle(5))  # Affiche 120
print(calculer_factorielle(-3))  # Provoque une AssertionError
```

Attention : les assertions peuvent être désactivées lors de l'exécution du programme avec l'option *-O*, il ne faut donc pas les utiliser pour la gestion des erreurs en production.

5.1.9 Gestion des exceptions imbriquées

Il est possible d'imbriquer des blocs *try...except* pour gérer des exceptions à différents niveaux.

Exemple :

```python
try:
    try:
        num = int(input("Entrez un nombre : "))
        resultat = 10 / num
    except ZeroDivisionError:
```

```
        print("Erreur : division par zéro.")
    else:
        print(f"Résultat : {resultat}")
except ValueError:
    print("Erreur : veuillez entrer un nombre valide.")
```

5.1.10 Exercice sur les exceptions

Cet exercice vous aidera à comprendre comment utiliser les exceptions en Python pour gérer des erreurs qui pourraient survenir lors de l'exécution d'un programme. L'idée est de prévoir ces erreurs et d'agir en conséquence pour éviter que le programme ne plante.

Vous devez écrire une fonction qui permet de diviser deux nombres et de gérer les erreurs possibles lors de cette division. Le but est de comprendre comment utiliser le bloc *try/except* pour attraper des exceptions courantes, comme la division par zéro ou une entrée non valide.

Détails de la fonction :

- La fonction s'appellera diviser.
- Elle prendra deux arguments : numérateur et dénominateur.
- La fonction doit gérer les erreurs suivantes :
  - **Division par zéro** : Si le dénominateur est égal à 0, la fonction doit afficher un message d'erreur approprié : *"Erreur : Division par zéro non permise."*
  - **Entrée non numérique** : Si l'un des deux arguments n'est pas un nombre (par exemple une chaîne de caractères), la fonction doit afficher un message d'erreur : *"Erreur : Les deux valeurs doivent être des nombres."*

Solution : Voici comment vous pourriez structurer votre fonction :

```
def diviser(numerateur, denominateur):
    try:
        # Effectuer la division
        resultat = numerateur / denominateur
    except ZeroDivisionError:
        # Gérer l'erreur de division par zéro
        print("Erreur : Division par zéro non permise.")
    except TypeError:
        # Gérer l'erreur de type non numérique
        print("Erreur : Les deux valeurs doivent être des nombres.")
    else:
        # Si tout se passe bien, afficher le résultat
        print(f"Le résultat de la division est : {resultat}")
```

Cas de test :

1. **Cas normal (division valide) :**

```
diviser(10, 2)
```

Sortie attendue :

```
Le résultat de la division est : 5.0
```

2. **Cas de division par zéro :**

```
diviser(10, 0)
```

Sortie attendue :

```
Erreur : Division par zéro non permise.
```

3. **Cas avec des valeurs non numériques :**

```
diviser(10, "deux")
```

Sortie attendue :

```
Erreur : Les deux valeurs doivent être des nombres.
```

4. **Cas où le numérateur est une chaîne de caractères :**

```
diviser("dix", 2)
```

Sortie attendue :

```
Erreur : Les deux valeurs doivent être des nombres.
```

5.1.11 Bonnes pratiques en gestion d'exceptions

- **Spécifier les exceptions :** Capturer des exceptions spécifiques permet de mieux comprendre et gérer les erreurs. Éviter d'utiliser un *except* sans préciser le type d'exception.
- **Fournir des messages clairs :** Les messages d'erreur doivent être compréhensibles pour l'utilisateur, indiquant ce qui s'est mal passé et, si possible, comment y remédier.
- **Ne pas masquer les erreurs :** Éviter de capturer toutes les exceptions sans traitement approprié, car cela peut rendre le débogage difficile.
- **Utiliser le bloc *finally* pour le nettoyage** : S'assurer que les ressources sont correctement libérées, même en cas d'erreur.
- **Lever des exceptions au lieu de retourner des codes d'erreur :** Cela rend le code plus clair et facilite la gestion des erreurs.
- **Ne pas abuser des exceptions :** Les exceptions doivent être utilisées pour des conditions exceptionnelles, pas pour le contrôle de flux normal du programme.

```python
try:
    # Code susceptible de provoquer une exception
except Exception:
    pass  # Mauvaise pratique, toutes les exceptions sont masquées
```

Exemple de bonne pratique :

```python
try:
    # Code susceptible de provoquer une exception
except SpecificException as e:
    # Traitement approprié de l'exception
    print(f"Erreur : {e}")
```

5.1.12 Conclusion sur les exceptions

La **gestion des exceptions** est essentielle pour écrire des programmes robustes en Python. En maîtrisant les techniques pour gérer, lever et créer des exceptions, il est possible d'améliorer significativement la fiabilité et la maintenabilité du code. Une gestion appropriée des erreurs permet non seulement d'éviter les plantages, mais aussi d'offrir une meilleure expérience utilisateur.

5.2 Maîtriser la gestion des fichiers avec python

La gestion des fichiers est une compétence essentielle en Python. Elle permet de lire, écrire et manipuler des données stockées dans des fichiers sur le disque dur. Que ce soit pour traiter des fichiers texte, des données structurées ou pour enregistrer les résultats d'un programme, savoir comment gérer les fichiers est indispensable pour tout développeur en Python.

5.2.1 Pourquoi gérer les fichiers en Python

Les fichiers sont un moyen courant de stocker et d'échanger des données. En Python, la gestion des fichiers permet de :

- Lire des données depuis des fichiers pour les traiter ou les analyser.
- Écrire des données dans des fichiers pour les sauvegarder ou les partager.
- Manipuler différents types de fichiers, tels que les fichiers texte, CSV, JSON, etc.
- Automatiser des tâches liées au traitement de données, comme la génération de rapports ou la transformation de données.

Comprendre comment interagir avec les fichiers permet de développer des applications plus complètes et utiles.

5.2.2 Ouvrir et fermer des fichiers

Pour travailler avec un fichier en Python, il faut d'abord l'ouvrir en utilisant la fonction intégrée *open()*. Cette fonction retourne un objet fichier qui peut être utilisé pour lire ou écrire dans le fichier. Une fois les opérations terminées, il est important de fermer le fichier en appelant la méthode *close()* pour libérer les ressources.

Syntaxe de base :

```
fichier = open('chemin/vers/fichier.txt', mode)
## Opérations sur le fichier
fichier.close()
```

Le paramètre mode spécifie le mode d'ouverture du fichier. Ci-dessous la liste des les différents modes d'ouverture de fichiers disponibles :

- **r** : Lecture seule. Le fichier doit exister.
- **w** : Écriture. Crée un nouveau fichier ou écrase le fichier existant.
- **a** : Ajout. Ajoute les données à la fin du fichier existant.
- **r+** : Lecture et écriture. Le fichier doit exister.
- **w+** : Lecture et écriture. Crée un nouveau fichier ou écrase le fichier existant.
- **a+** : Lecture et ajout. Ajoute les données à la fin du fichier existant.
- **rb** : Lecture en mode binaire. Le fichier doit exister.
- **wb** : Écriture en mode binaire. Crée un nouveau fichier ou écrase le fichier existant.
- **ab** : Ajout en mode binaire. Ajoute les données à la fin du fichier existant.
- **rb+** : Lecture et écriture en mode binaire. Le fichier doit exister.
- **wb+** : Lecture et écriture en mode binaire. Crée un nouveau fichier ou écrase le fichier existant.
- **ab+** : Lecture et ajout en mode binaire. Ajoute les données à la fin du fichier existant.

Exemple d'ouverture en lecture :

```
fichier = open('donnees.txt', 'r')
## Lecture du fichier
fichier.close()
```

Exemple d'ouverture en écriture :

```
fichier = open('resultats.txt', 'w')
## Écriture dans le fichier
fichier.close()
```

Exemple d'ouverture en mode binaire :

```
fichier = open('image.png', 'rb')
contenu = fichier.read()
fichier.close()
```

Il est essentiel de fermer le fichier après utilisation pour éviter les fuites de ressources et les problèmes de corruption de données.

5.2.3 Lire des fichiers

Une fois le fichier ouvert en mode lecture, plusieurs méthodes permettent de lire son contenu. Lire tout le contenu

La méthode *read()* lit tout le contenu du fichier et le retourne sous forme de chaîne de caractères.

Exemple :

```
fichier = open('donnees.txt', 'r')
contenu = fichier.read()
print(contenu)
fichier.close()
```

5.2.4 Lire ligne par ligne

La méthode *readline()* lit une seule ligne à la fois. La fonction *readline()* lit une seule ligne à la fois d'un fichier ouvert, ce qui est pratique pour traiter les fichiers ligne par ligne sans charger l'ensemble du fichier en mémoire. Par exemple, si vous avez un fichier texte contenant plusieurs lignes et que vous souhaitez afficher chaque ligne séparément, vous pouvez utiliser *readline()* dans une boucle while pour lire chaque ligne successivement.

Exemple :

```
fichier = open('donnees.txt', 'r')
ligne = fichier.readline()
while ligne:
    print(ligne.strip())
    ligne = fichier.readline()
fichier.close()
```

l'utilisation de la fonction *readline()* présente de nombreux avantages, tels que :

- **Gestion efficace de la mémoire :** En lisant le fichier ligne par ligne, *readline()* évite de charger l'ensemble du fichier en mémoire, ce qui est particulièrement bénéfique lorsque vous travaillez avec des fichiers volumineux. Cela permet de gérer des fichiers de taille variable sans risquer de surcharger la mémoire disponible.
- **Traitement séquentiel des données :** Grâce à *readline()*, vous pouvez traiter chaque ligne individuellement au fur et à mesure qu'elle est lue, ce qui est idéal pour les opérations nécessitant une analyse ligne par ligne. Cela facilite la gestion de données structurées ou de logs où chaque ligne contient des informations distinctes.
- **Flexibilité dans la gestion des fichiers :** La capacité de lire les fichiers ligne par ligne permet de gérer des fichiers de tailles variables avec une plus grande flexibilité. Vous pouvez adapter votre traitement en fonction du contenu de chaque ligne, ce qui offre un contrôle plus précis sur le traitement des données.

- **Simplicité d'utilisation :** *readline()* est facile à utiliser dans une boucle pour itérer à travers chaque ligne du fichier. Cette simplicité facilite l'implémentation de processus de lecture et de traitement de fichiers sans nécessiter de structures complexes.

5.2.5 Lire plusieurs lignes avec *readlines()*

La fonction *readlines()* permet de lire toutes les lignes d'un fichier ouvert et de les retourner sous forme de liste. Chaque élément de cette liste correspond à une ligne du fichier, y compris le caractère de fin de ligne (). Cette fonction est particulièrement utile lorsque vous souhaitez charger l'intégralité du contenu d'un fichier en mémoire et traiter ses lignes individuellement par la suite.

Lors de l'utilisation de *readlines()*, chaque ligne est lue et stockée comme un élément distinct dans une liste Python, ce qui permet un accès rapide et facile à n'importe quelle ligne en utilisant les index de la liste. Toutefois, il est important de noter que *readlines()* peut consommer beaucoup de mémoire si le fichier est très volumineux, car toutes les lignes sont chargées simultanément en mémoire.

```
with open("file.txt", "r") as f:
    lignes = f.readlines()  # Lire toutes les lignes du fichier dans une liste
for ligne in lignes:
    print(ligne)  # Afficher chaque ligne
```

Dans cet exemple, *readlines()* lit toutes les lignes du fichier *example.txt* et les stocke dans la liste lignes. Ensuite, une boucle for est utilisée pour parcourir chaque élément de la liste et afficher son contenu.

L'utilisation de *readlines()* est pratique lorsque vous avez besoin de traiter toutes les lignes d'un fichier en même temps, comme pour des opérations de filtrage, de tri ou d'analyse qui nécessitent d'accéder à l'ensemble du contenu du fichier.

L'utilisation de la fonction *readlines()* présente de nombreux avantages, tels que :

- **Accès rapide aux lignes :** En utilisant *readlines()*, toutes les lignes du fichier sont chargées en mémoire sous forme de liste. Cela permet un accès immédiat et rapide à n'importe quelle ligne en utilisant les index de la liste, ce qui est pratique lorsque vous avez besoin de lire ou de manipuler des lignes spécifiques du fichier plusieurs fois.
- **Simplicité de traitement :** La fonction retourne les lignes du fichier sous forme de liste, ce qui simplifie le traitement des données. Vous pouvez facilement itérer sur les lignes, appliquer des transformations, ou effectuer des analyses sur l'ensemble du contenu du fichier sans avoir à gérer des lectures répétées.
- **Manipulation facile des données :** Une fois les lignes chargées dans une liste, vous pouvez utiliser toutes les fonctionnalités de manipulation des listes en Python, comme les tranches (slicing), les méthodes de recherche, ou les opérations de filtrage, pour traiter les données du fichier.
- **Opérations en mémoire :** Lorsque la taille du fichier est gérable, charger toutes les lignes en mémoire peut simplifier le traitement, car vous pouvez effectuer des opérations comme la recherche, le tri, ou les modifications sur l'ensemble des données en une seule fois, **sans avoir à réouvrir le fichier**.

- **Pratique pour les petits fichiers :** Pour les fichiers de taille modeste, *readlines()* offre une manière simple et efficace de lire toutes les lignes en une seule opération, facilitant ainsi le traitement et l'analyse sans complexité additionnelle.

5.2.6 Itérer sur le fichier

Il est également possible d'itérer directement sur l'objet fichier.

Exemple :

```
fichier = open('donnees.txt', 'r')
for ligne in fichier:
    print(ligne.strip())
fichier.close()
```

Cette méthode est efficace pour les fichiers volumineux car elle lit une ligne à la fois.

5.2.7 Écrire dans des fichiers

Pour écrire dans un fichier, il faut l'ouvrir en mode écriture (*'w'*), ajout (*'a'*) ou lecture-écriture (*'r+'*). Écrire du texte

La méthode *write()* permet d'écrire une chaîne de caractères dans le fichier.

Exemple :

```
fichier = open('resultats.txt', 'w')
fichier.write('Ceci est un test.\n')
fichier.write('Écriture dans un fichier en Python.\n')
fichier.close()
```

5.2.8 Écrire plusieurs lignes

Il est possible d'écrire une liste de lignes avec la méthode *writelines()*.

Exemple :

```
lignes = ['Ligne 1\n', 'Ligne 2\n', 'Ligne 3\n']
fichier = open('resultats.txt', 'w')
fichier.writelines(lignes)
fichier.close()
```

5.2.9 Ajouter du contenu à un fichier existant

En ouvrant le fichier en mode ajout (*'a'*), le nouveau contenu est ajouté à la fin du fichier sans effacer le contenu existant.

Exemple :

```
fichier = open('resultats.txt', 'a')
fichier.write('Nouvelle ligne ajoutée.\n')
fichier.close()
```

5.2.10 Utiliser le bloc 'with'

L'utilisation du mot-clé with permet de gérer automatiquement l'ouverture et la fermeture du fichier, même en cas d'exception. Cela rend le code plus sûr et plus lisible.

Syntaxe :

```
with open('chemin/vers/fichier.txt', mode) as fichier:
    # Opérations sur le fichier
```

Le fichier est automatiquement fermé à la fin du bloc with.

Exemple :

```
with open('donnees.txt', 'r') as fichier:
    for ligne in fichier:
        print(ligne.strip())
```

5.2.11 Travailler avec différents formats de fichiers

Fichiers CSV

Les fichiers CSV (*Comma-Separated Values*) sont couramment utilisés pour stocker des données tabulaires.

Lecture d'un fichier CSV :

```
import csv

with open('donnees.csv', 'r') as fichier_csv:
    lecteur = csv.reader(fichier_csv)
    for ligne in lecteur:
        print(ligne)
```

Écriture dans un fichier CSV :

```python
import csv

donnees = [['Nom', 'Age'], ['Alice', '30'], ['Bob', '25']]

with open('resultats.csv', 'w', newline='') as fichier_csv:
    ecrivain = csv.writer(fichier_csv)
    ecrivain.writerows(donnees)
```

Fichiers JSON

Les fichiers JSON (*JavaScript Object Notation*) sont utilisés pour stocker des données structurées de manière hiérarchique.

Lecture d'un fichier JSON :

```python
import json

with open('donnees.json', 'r') as fichier_json:
    donnees = json.load(fichier_json)
    print(donnees)
```

Écriture dans un fichier JSON :

```python
import json

donnees = {'nom': 'Alice', 'age': 30, 'ville': 'Paris'}

with open('resultats.json', 'w') as fichier_json:
    json.dump(donnees, fichier_json)
```

5.2.12 Les exceptions liées aux fichiers

Lors de la gestion des fichiers, plusieurs exceptions peuvent survenir, comme *FileNotFoundError*, *PermissionError* ou *IOError*. Il est important de gérer ces exceptions pour rendre le programme plus robuste.

Exemple de gestion d'exception :

```python
try:
    with open('inexistant.txt', 'r') as fichier:
        contenu = fichier.read()
except FileNotFoundError:
    print("Erreur : le fichier n'a pas été trouvé.")
except PermissionError:
    print("Erreur : permissions insuffisantes pour ouvrir le fichier.")
except Exception as e:
    print(f"Erreur inattendue : {e}")
```

5.2.13 Navigation dans le système de fichiers avec Python

Naviguer dans le système de fichiers est une compétence cruciale pour gérer les fichiers et les répertoires de manière efficace. Python offre plusieurs modules pour cette tâche, notamment *os* et *os.path*.

Changer de répertoire

Pour changer le répertoire de travail courant :

```
import os
os.chdir('/chemin/vers/repertoire')
```

Obtenir le répertoire courant

Pour afficher le répertoire de travail actuel :

```
current_directory = os.getcwd()
print(current_directory)
```

Lister les fichiers et répertoires

Pour lister le contenu d'un répertoire :

```
contenu = os.listdir('/chemin/vers/repertoire')
print(contenu)
```

Créer et supprimer des répertoires

Pour créer un nouveau répertoire :

```
os.mkdir('nouveau_repertoire')
```

Pour supprimer un répertoire vide :

```
os.rmdir('nouveau_repertoire')
```

Gérer les chemins de fichiers

Le module *os.path* permet de manipuler les chemins de fichiers de manière portable :

```
import os.path
chemin_complet = os.path.join('dossier', 'fichier.txt')
print(chemin_complet)
```

Vous pouvez également vérifier l'existence d'un fichier ou répertoire :

```
existe = os.path.exists('chemin/vers/fichier.txt')
print(existe)  # Affiche True ou False
```

Ces outils vous permettent de naviguer et gérer efficacement le système de fichiers, rendant vos scripts Python plus robustes et polyvalents.

5.2.14 Manipulation avancée des fichiers

La manipulation avancée des fichiers en Python implique des opérations plus complexes comme la copie, le déplacement, la suppression récursive et la gestion des permissions. Pour cela, nous utilisons principalement les modules *shutil* et *os*.

Copier et déplacer des fichiers

Pour copier un fichier :

```
import shutil
shutil.copy('source.txt', 'destination.txt')
```

Pour déplacer un fichier :

```
shutil.move('source.txt', 'nouveau_dossier/destination.txt')
```

Suppression récursive de répertoires

Pour supprimer un répertoire et tout son contenu :

```
shutil.rmtree('repertoire_a_supprimer')
```

Modification des permissions de fichiers

Changer les permissions d'un fichier :

```
import os
os.chmod('fichier.txt', 0o755)
```

Lecture et écriture des métadonnées

Pour obtenir des informations détaillées sur un fichier :

```
import os
info = os.stat('fichier.txt')
print(info)
```

Gestion des fichiers temporaires

Pour créer et utiliser des fichiers temporaires :

```
import tempfile
with tempfile.NamedTemporaryFile(delete=True) as temp_file:
    temp_file.write(b'Quelques donnees temporaires')
    print(temp_file.name)
```

Ces techniques avancées permettent une gestion plus fine et flexible des fichiers, essentielle pour des applications robustes et professionnelles.

5.2.15 Exemples pratiques et cas d'utilisation

Pour illustrer l'utilisation des techniques de manipulation de fichiers en Python, voici quelques exemples pratiques et cas d'utilisation.

Exemple 1 : Traitement de fichiers CSV

Lire un fichier *CSV* et calculer la moyenne d'une colonne :

```
import csv

with open('data.csv', 'r') as file:
    reader = csv.DictReader(file)
    total = 0
    count = 0
    for row in reader:
        total += int(row['valeur'])
        count += 1

    moyenne = total / count
    print(f'Moyenne: {moyenne}')
```

Exemple 2 : Fusion de plusieurs fichiers texte

Combiner le contenu de plusieurs fichiers texte en un seul fichier :

```python
import os

fichiers = ['fichier1.txt', 'fichier2.txt', 'fichier3.txt']
with open('fusion.txt', 'w') as fichier_fusion:
    for nom_fichier in fichiers:
        with open(nom_fichier, 'r') as fichier:
            contenu = fichier.read()
            fichier_fusion.write(contenu + '\n')
```

Exemple 3 : Nettoyage de répertoires

Supprimer les fichiers temporaires d'un répertoire :

```python
import os
import shutil

dossier = '/chemin/vers/dossier'
for nom_fichier in os.listdir(dossier):
    chemin_fichier = os.path.join(dossier, nom_fichier)
    if nom_fichier.endswith('.tmp'):
        os.remove(chemin_fichier)
```

Ces exemples montrent comment les techniques de manipulation de fichiers peuvent être appliquées à des tâches courantes, améliorant ainsi l'efficacité et l'automatisation de vos scripts Python.

5.2.16 Exercice sur les fichiers

L'objectif de ce programme est de lire un fichier texte contenant des informations sur des utilisateurs, traiter ces informations et les afficher de manière formatée. Le programme est conçu pour gérer les erreurs courantes, comme l'absence du fichier ou un mauvais formatage des données dans le fichier.

Ce que doit contenir le programme :

1. Une fonction pour lire le fichier : Le programme lit un fichier texte contenant des informations utilisateur. Le fichier contient plusieurs lignes, chaque ligne ayant le format : nom,age,ville.
2. Une fonction pour traiter les lignes : Le programme doit analyser chaque ligne, en extraire les informations et les traiter.
3. Une fonction pour afficher les informations : Le programme doit afficher les informations extraites (nom, âge, ville) de manière lisible.
4. Gestion des erreurs :

   — Gérer les erreurs lorsque le fichier est absent (afficher un message approprié).
   — Gérer les erreurs lorsque certaines lignes sont mal formatées (afficher un message indiquant la ligne problématique).

5. Une fonction principale : C'est le point d'entrée du programme qui gère l'exécution des différentes étapes.

Fonctionnement du programme :

1. **Étape 1 : Lecture du fichier texte**

   — Le programme commence par lire un fichier texte contenant des informations d'utilisateurs.
   — Le fichier à lire s'appelle utilisateurs.txt et doit être placé dans le même répertoire que le programme.
   — Chaque ligne du fichier doit être formatée comme ceci : nom,âge,ville. Par exemple :

   ```
   Alice,30,Paris
   Bob,25,Lyon
   Claire,28,Marseille
   ```

   Solution : Commençons par écrire un programme qui ouvre puis lit le fichier, affiche chaque ligne dans une boucle et finit par fermer le fichier :

   ```python
   # Lecture du fichier utilisateurs.txt
   fichier = open('utilisateurs.txt', 'r')
   lignes = fichier.readlines()

   # Afficher chaque utilisateur
   for ligne in lignes:
       print(ligne.strip())  # .strip() pour enlever les espaces et retours
       ↪  à la ligne

   fichier.close()
   ```

   Explication :
   — **open()** : Ouvre le fichier en mode lecture ('r').
   — **readlines()** : Lit toutes les lignes du fichier et les stocke dans une liste.
   — **strip()** : Supprime les espaces inutiles et les sauts de ligne à la fin de chaque ligne.

   Sortie attendue :

   ```
   Alice,30,Paris
   Bob,25,Lyon
   Claire,28,Marseille
   ```

2. **Étape 2 : Gestion des erreurs lors de la lecture**

   — Si le fichier n'existe pas, le programme affiche un message d'erreur : Erreur : Le fichier 'utilisateurs.txt' n'a pas été trouvé.
   — Si le fichier est trouvé, il lit toutes les lignes et les renvoie pour être traitées.

   Solution : Nous allons maintenant gérer les erreurs courantes qui peuvent survenir lors de la lecture du fichier, comme un fichier qui n'existe pas. Si le fichier utilisateurs.txt n'est pas trouvé, une erreur FileNotFoundError est levée.

   Code avec gestion d'erreurs (toujours sans fonction) :

```
# Gestion des erreurs lors de la lecture du fichier
try:
    fichier = open('utilisateurs.txt', 'r')
    lignes = fichier.readlines()

    # Afficher chaque utilisateur
    for ligne in lignes:
        print(ligne.strip())

    fichier.close()

except FileNotFoundError:
    print("Erreur : Le fichier 'utilisateurs.txt' n'a pas été trouvé.")
```

** Explication :**

- Bloc try : Le code susceptible de lever une exception est placé dans ce bloc.
- FileNotFoundError : Si le fichier n'existe pas, cette exception est levée et le programme affiche un message d'erreur au lieu de planter.

**Cas de fichier manquant : ** Si le fichier utilisateurs.txt n'existe pas, le programme affiche :

```
Erreur : Le fichier 'utilisateurs.txt' n'a pas été trouvé.
```

3. **Étape 3 : Ajouter une vérification du format des lignes**

- Le programme analyse chaque ligne lue à partir du fichier.
- Il s'attend à ce que chaque ligne soit correctement formatée avec trois éléments séparés par des virgules : nom, âge, ville.
- Si une ligne n'est pas correctement formatée, le programme affiche un message d'erreur spécifique à cette ligne : Erreur de format : 'ligne' ne contient pas les informations attendues.

Solution : Maintenant, nous allons ajouter une gestion des erreurs au cas où une ligne dans le fichier serait mal formatée. Par exemple, si une ligne ne contient pas trois éléments (nom, âge, ville), nous afficherons un message d'erreur spécifique pour cette ligne.

Code avec gestion des erreurs de format (sans fonction) :

```
# Gestion des erreurs lors de la lecture du fichier et du format des
↪ lignes
try:
    fichier = open('utilisateurs.txt', 'r')
    lignes = fichier.readlines()

    # Traitement de chaque ligne utilisateur
    for ligne in lignes:
        try:
            nom, age, ville = ligne.strip().split(',')
            print(f"Nom: {nom}, Âge: {age}, Ville: {ville}")
        except ValueError:
```

```
            print(f"Erreur de format : '{ligne.strip()}' ne contient pas
            ↪ les informations attendues.")

        fichier.close()

    except FileNotFoundError:
        print("Erreur : Le fichier 'utilisateurs.txt' n'a pas été trouvé.")
```

Explication :

— **Bloc try dans un try** : Le premier bloc *try* gère les erreurs liées à l'ouverture du fichier, tandis que le second gère les erreurs liées au format des lignes.

— *ValueError* : Si une ligne ne peut pas être divisée en trois éléments par *split(',')*, une exception *ValueError* est levée.

4. **Étape 4 : Refactorisation avec des fonctions**

— Pour chaque ligne correctement formatée, le programme affiche les informations de l'utilisateur de manière claire : Nom : Alice, Âge : 30, Ville : Paris.

Solution :

Maintenant que notre code fonctionne correctement, nous allons l'organiser en fonctions pour le rendre plus lisible, modulaire et maintenable. Cela nous permet aussi de réutiliser des parties de code et de séparer les responsabilités dans notre programme.

Code refactorisé avec des fonctions :

```
#!/usr/bin/env python3
# Auteur : Stéphane ROBERT
# Date : 2024-09-04
# Description : Ce programme lit un fichier texte contenant des
↪ informations
#               sur des utilisateurs, les traite et les affiche. Il gère
↪ les erreurs
#               liées à l'absence du fichier et aux lignes mal formatées.

# Fonction pour lire le fichier
def lire_fichier(nom_fichier):
    try:
        with open(nom_fichier, 'r') as fichier:
            return fichier.readlines()  # Lire toutes les lignes du
            ↪ fichier
    except FileNotFoundError:
        print(f"Erreur : Le fichier '{nom_fichier}' n'a pas été trouvé.")
        return None

# Fonction pour traiter les lignes du fichier
def traiter_lignes(lignes):
    for ligne in lignes:
        try:
            nom, age, ville = ligne.strip().split(',')
            afficher_utilisateur(nom, age, ville)
        except ValueError:
```

```
                print(f"Erreur de format : '{ligne.strip()}' ne contient pas
                ↪  les informations attendues.")

    # Fonction pour afficher les informations d'un utilisateur
    def afficher_utilisateur(nom, age, ville):
        print(f"Nom: {nom}, Âge: {age}, Ville: {ville}")

    # Fonction principale pour exécuter le programme
    def programme_principal():
        nom_fichier = 'utilisateurs.txt'
        lignes = lire_fichier(nom_fichier)

        if lignes:
            traiter_lignes(lignes)

    # Exécution du programme
    if __name__ == '__main__':
        programme_principal()
```

Explication :

- *lire_fichier()* : Ouvre le fichier et retourne les lignes. Si le fichier est manquant, un message d'erreur est affiché.
- *traiter_lignes()* : Traite chaque ligne en la découpant et en vérifiant si elle respecte le bon format (nom, âge, ville).
- *afficher_utilisateur()* : Affiche les informations d'un utilisateur sous forme formatée.
- *afficher_utilisateur()* : Affiche les informations d'un utilisateur sous forme formatée.
- *programme_principal()* : Fonction principale qui orchestre la lecture et le traitement des données.

5. **Exécution du programme :**

   - Lorsque le programme est lancé, il exécute la fonction principale, qui lit le fichier, traite les lignes et affiche les résultats.

5.2.17 Bonnes pratiques pour la gestion des fichiers

- Utiliser le bloc *with* pour assurer la fermeture automatique des fichiers.
- Gérer les exceptions pour informer l'utilisateur en cas d'erreur.
- Spécifier le bon encodage lors de l'ouverture des fichiers texte (par défaut *'utf-8'*). **Exemple avec encodage :**

```
with open('donnees.txt', 'r', encoding='utf-8') as fichier:
    contenu = fichier.read()
```

- Éviter d'utiliser des chemins de fichiers en dur. Utiliser le module *os* ou *pathlib* pour construire des chemins indépendants du système d'exploitation.

Exemple avec pathlib :

```
from pathlib import Path

chemin = Path('dossier') / 'fichier.txt'
with open(chemin, 'r') as fichier:
    contenu = fichier.read()
```

- **Ne pas oublier de fermer les fichiers** si le bloc *with* n'est pas utilisé.

5.2.18 Conclusion sur les fichiers

La **gestion des fichiers** en Python est une compétence fondamentale pour tout développeur. Elle permet de lire et d'écrire des données, de manipuler différents formats de fichiers et d'automatiser de nombreuses tâches liées au traitement de données. En maîtrisant les méthodes d'ouverture, de lecture, d'écriture et en adoptant les bonnes pratiques, il est possible de créer des programmes robustes et efficaces.

TP :

- Script de traitement d'un fichier CSV d'étudiants.
- Analyseur de logs système.

Chapitre 6

Module 6 : Programmation réseau

Objectifs :

— Savoir développer des applications client-serveur.
— Savoir utiliser les sockets et les protocoles HTTP.

Contenu :

- Les sockets TCP et UDP.
- Serveur et client simples.
- Utilisation des bibliothèques requests et http.server.

TP :

- Développement d'un système de chat interposé
- Téléchargement automatisé de pages web.

Chapitre 7

Module 7 : Python et scripts système (Bash, PowerShell)

Objectifs :

— Automatiser des tâches système
— Interagir avec le shell et les processus

Contenu :

- Utilisation du module os, sys, subprocess.
- Interaction avec le système de fichiers.
- Exécution de commandes Bash/Windows.

TP :

- Script Python pour sauvegarder automatiquement des fichiers.
- Script de surveillance de l'espace disque.

Chapitre 8

Module 8 : Interfaces graphiques

Objectifs :

— Concevoir des applications desktop complètes avec python

Contenu :

- Tkinter (interface native).
- PyQt (interfaces modernes).
- Organisation en événements, widgets, menus.

TP :

- Application de Suivi des dépenses personnelles avec PyQt.
- Application de gestion de contacts avec PyQt.

Chapitre 9

Module 9 : Développement web avec Python

Objectifs :

— Apprendre à developper des applications web avec python.

Contenu :

- Introduction au framework Django (Modèle MVT)
- Création et configuration (fichier "settings") d'un nouveau projet avec Django
- Couche Modèle
- Couche Views
- Couche Template
- Routage, templates, formulaires, gestion des données.

Connaître le développement web est devenu une compétence obligatoire pour un programmeur. Nous allons utiliser la framework Django pour apprendre à développer une application web. Ce sera une application qui vous permettra d'enregistrer toutes les informations relatives aux personnes qui entrent ou sortent du territoire national à destination ou en provenance d'un pays via l'une des frontières terrestres que regorge notre pays. Cette application a été commanditée par les services de la Police des Frontières qui souhaitent acquérir un système informatique simple et sécurisé pour la cause.

Mais avant de nous lancer dans le développement de cette solution, nous allons vous exposer toutes les notions qui se cachent derrière le framework Django.

9.1 Présentation de Django

9.1.1 Qu'est ce que Django ?

Django est un framework python open-source consacré au développement web 2.0 . Les concepteurs de Django lui ont attribué le slogan suivant : " Le framework web pour les perfectionnistes sous

pression ". Il est donc clairement orienté pour les développeurs ayant comme besoin de produire un projet solide rapidement et sans surprise ... c'est à dire à tous les développeurs !

Comme il est toujours compliqué de partir de rien, Django vous propose une base de projet solide. Django est donc une belle boîte à outils qui aide et oriente le développeur dans la construction de ses projets.

Pour la petite histoire Django a vu le jour en 2003 et a été publié sous licence BSD en juillet 2005. Il est gratuit, open source, a une communauté active, une bonne documentation, et plusieurs options pour du support gratuit ou non.

9.1.2 Pourquoi travailler avec le framework Django ?

Beaucoup de développeurs sont beaucoup plus productifs sur python que sur n'importe quel autre language. La technologie derrière ce framework peut être une bonne raison, mais Django est une raison à part entière. Tout est pensé de base. Inutile de réinventer la roue à chaque projet web : les bases sont déjà présentes. Il vous fournira les outils nécessaires à sécuriser votre application, à gérer la structure de vos modèles avec un ORM , etc. Vous n'avez plus qu'à vous consacrer au code métier, à vous concentrer sur l'essence même de votre projet et rien d'autre.

Django vous aide à écrire une application qui est :

- **Complète :** Django suit la philosophie "Piles incluses" et fournit presque tout ce que les développeurs pourraient vouloir faire. Comme tout ce dont vous avez besoin est une partie de ce "produit", tout fonctionne parfaitement ensemble, suivant des principes de conception cohérents, il possède également une documentation complète et à jour.
- **Polyvalent :** Django peut être (et a été) utilisé pour créer presque tous les genres de sites — du gestionnaire de données aux wikis, jusqu'aux réseaux sociaux et aux sites d'actualités. Il peut fonctionner avec n'importe quelle infrastructure côté client, et peut renvoyer des données dans quasiment n'importe quel format (notamment HTML, RSS, JSON, XML, etc). Le site sur lequel vous lisez en ce moment est basé sur Django !

  Tandis qu'il fournit presque toutes les fonctionnalités dont vous pourriez avoir besoin (comme des base de données populaires, des moteurs de modélisation, etc.), il peut tout de même être étendu pour utiliser d'autres composants si besoin.
- **Sécurisé :** Django aide les développeurs à éviter les erreurs de sécurité classique en fournissant une infrastructure conçue pour "faire ce qu'il faut" pour protéger les sites internet automatiquement. Par exemple, Django fournit un moyen sécurisé pour gérer les comptes des utilisateurs ainsi que leurs mots de passe, évitant les erreurs classiques comme mettre des informations sur la session dans des cookies, où elles sont vulnérables (à la place les cookies contiennent seulement une clé, et les données sont stockées dans la base de données), ou directement stocker des mots de passe, au lieu de mot de passe hachés.

  Django active par défaut la protection contre beaucoup de vulnérabilités, comme les injections SQL, le cross-site scripting, le cross-site request forgery et le clickjacking (voir Website security pour plus de détails sur ce genre d'attaques).
- **Scalable :** Django utilise une architecture composite "shared-nothing" (chaque composant de l'architecture est indépendant des autres, et peut ainsi être remplacé ou changé si besoin). En ayant des séparations nettes entres les différentes parties, Django peut se scaler lors d'une hausse de trafic en ajoutant du hardware à tous les niveaux : serveurs cache, serveurs de base de données, serveurs d'application. Certains des sites les plus fréquentés ont réussi à scaler

Django pour répondre à leur demande (par exemple, Instagram et Disqus pour ne nommer qu'eux deux).

- **Maintenable :** Les principes de design du code Django encouragent la création d'un code simple à maintenir et réutilisable. Il fait notamment appel à la philosophie du Ne Vous Répétez Pas (DRY pour Don't Repeat Yourself en anglais), afin d'éviter toute duplication superflue, réduisant la taille de votre code. Django promeut aussi le regroupement de fonctionnalités reliées entre elles en "applications" réutilisables et, à un plus bas niveau, regroupe des lignes de code dépendantes entre elles en modules (suivant les lignes du motif d'architecture Modèle-vue-contrôleur (MVC)).
- **Portable :** Django est écrit en Python, qui fonctionne sous diverses plateformes. Cela veut dire que vous ne serez plus contraint par une plateforme en particulier, et vous pourrez faire fonctionner vos applications sous autant de versions de Linux, Windows et Mac OS X que vous le souhaitez. De plus, Django est très bien supporté par plusieurs fournisseurs d'hébergement web, qui offrent souvent des infrastructures et de la documentation spécifiques pour héberger des sites Django.

9.1.3 IA et Django

Django est également une plateforme adaptée pour le développement d'applications utilisant l'intelligence artificielle. En effet, Python est un langage largement utilisé dans le domaine de l'IA, et Django offre de nombreux outils et bibliothèques pour intégrer facilement des modèles de machine learning dans vos projets. De plus, Django permet de créer des API REST simples et efficaces qui peuvent être utilisées pour interagir avec des modèles d'IA depuis d'autres applications. En utilisant Django pour développer des applications utilisant l'IA, vous bénéficiez de tous les avantages de ce framework tout en profitant des possibilités offertes par l'IA.

9.1.4 API REST et Django

Django offre une intégration native des API REST, ce qui en fait un choix idéal pour les développeurs souhaitant créer des applications qui utilisent cette architecture. En utilisant Django, vous pouvez rapidement et facilement créer une API REST en définissant simplement les modèles de données de votre application et en utilisant les outils fournis par le framework. Django prend également en charge de nombreux standards REST, tels que le format de données JSON et les méthodes HTTP courantes (GET, POST, PUT, DELETE). De plus, Django propose des outils pour authentifier et autoriser les utilisateurs de votre API, ce qui en fait un framework complet pour la création d'API REST sécurisées et fiables.

9.1.5 La structure Django

Django repose sur une architecture *MVT (Modèle-Vue-Template)*, une variante du design pattern *MVC (Modèle-Vue-Contrôleur)*.

Voici les composants clés de sa structure :

1. **Le Modèle (Model) :** Le modèle gère les données et la logique métier. Il définit la structure de la base de données via des classes Python. Django utilise un ORM (Object-Relational

Mapper), ce qui permet de manipuler la base de données sans écrire de SQL. Pour gérer les changements, on utilise le système de Migrations Django.

2. **La Vue (View) :** La vue contient la logique de traitement des requêtes. Elle reçoit une requête HTTP, récupère les données via le Modèle et décide quelle réponse renvoyer. Elle peut renvoyer un fichier HTML, une erreur 404 ou des données JSON (pour une API). Vous pouvez consulter le guide des Vues Django pour plus de détails.
3. **Le Template (Gabarit) :** Le template gère l'interface utilisateur (le rendu visuel). C'est un fichier HTML mélangé avec le Django Template Language (DTL). Il permet d'afficher dynamiquement les données envoyées par la Vue.
4. **L'URL Dispatcher (Le Routage) :** Bien que non inclus dans l'acronyme MVT, le fichier *urls.py* est crucial. Il fait le lien entre une URL saisie par l'utilisateur et la fonction de la Vue correspondante.

Structure type d'un projet Django

Lorsque vous créez un projet, Django génère une arborescence spécifique :

- *manage.py* : L'outil en ligne de commande pour gérer votre projet (lancer le serveur, créer des migrations).
- **Dossier du projet** : Contient *settings.py* (configuration globale) et *urls.py* (routes principales).
- **Applications (Apps)** : Un projet Django est découpé en "*apps*" réutilisables (ex : blog, users, cart). Chaque *app* possède ses propres fichiers *models.py*, *views.py* et *admin.py*.

9.1.6 Comment les requêtes sont traitées dans Django

Lorsqu'un utilisateur interagit avec votre site piloté par Django (en cliquant sur un lien, en envoyant un formulaire ou en accédant à une URL, etc.), une requête est envoyée au serveur. Django traite cette requête au cours du cycle suivant :

1. Le répartiteur d'URL associe la requête à un schéma d'URL défini et la redirige vers la vue correspondante.
2. La vue interagit avec les modèles si des données sont nécessaires, exécute une logique métier, puis transfère les informations à un template.
3. Le template assure le rendu de la page HTML finale, qui est ensuite renvoyée au navigateur de l'utilisateur en tant que réponse.

Ce processus clair permet de comprendre facilement le comportement de votre application à chaque étape et en simplifie le débogage. Pour une présentation plus détaillée, consultez la documentation sur les requêtes et les réponses de Django.

9.1.7 Une documentation officielle très pédagogique

On aime aussi dans Django sa grande popularité , ce qui fait qu'il existe de nombreux tutoriels et la documentation officielle est d'ailleurs très bien faite en plusieurs langues.

Django dispose d'une documentation officielle très complète et accessible, qui est particulièrement adaptée aux débutants. Elle est organisée de manière claire et logique, ce qui en fait une ressource précieuse pour tous ceux qui souhaitent apprendre à utiliser Django. La documentation couvre tous les aspects du framework, des bases de l'installation et de la configuration jusqu'aux fonctionnalités avancées. Elle est également très bien illustrée et inclut de nombreux exemples concrets qui permettent de mieux comprendre les concepts présentés. En outre, la documentation est régulièrement mise à jour pour refléter les dernières évolutions de Django, ce qui en fait une ressource fiable et à jour.

9.1.8 Qui utilise Django ?

En 2025, Django demeure l'un des frameworks web les plus utilisés au monde par une grande variété d'organisations, des géants technologiques aux agences gouvernementales.

Voici les principaux utilisateurs classés par catégorie :

1. **Géants du Web et Médias Sociaux :** Plusieurs des plateformes les plus visitées au monde reposent sur Django pour gérer des milliards d'utilisateurs et de données :

    - Instagram : Utilise Django pour traiter des millions d'interactions quotidiennes (photos, likes, partages).
    - YouTube : Une grande partie de son backend est construite avec ce framework.
    - Udemy : Une grande partie du backend de cette plateforme de formation en ligne est construite avec ce framework.
    - Pinterest : L'architecture du site gère des milliards d'"épingles" grâce à Django.
    - Spotify : Combine Python et Django pour ses services backend et ses algorithmes d'apprentissage automatique.

2. **Services de Stockage et Plateformes Techniques**

    - Dropbox : Utilise le framework pour ses services de stockage cloud sécurisés.
    - Mozilla : Plusieurs services de la fondation, notamment la documentation MDN Web Docs, sont motorisés par Django.
    - Disqus : Ce système de commentaires mondial gère des pics de trafic massifs via Django.

3. **Organisations Scientifiques et Gouvernementales**

    - NASA : Utilise Django pour diverses applications de recherche scientifique et de gestion de données spatiales.
    - The Washington Post : Historiquement lié à la création du framework, le journal l'utilise toujours pour sa publication en ligne rapide.

4. **Secteur Financier (FinTech) :** De nombreuses banques et solutions de paiement adoptent Django pour sa sécurité native : PayPal et Venmo exploitent ses capacités pour sécuriser les transactions financières. HSBC a recours à des solutions basées sur Django pour améliorer l'accessibilité de ses transactions bancaires.

5. **Profil des utilisateurs en 2025 :**

   - **Développeurs** : 74 % des développeurs Django l'utilisent pour le développement full-stack et 60 % pour la création d'API (via Django REST framework).
   - **Adoption de l'IA** : Environ 38 % des développeurs intègrent désormais des outils d'IA dans leurs projets Django.
   - **Secteur de l'emploi** : Le marché reste robuste en 2025, avec une forte demande pour des rôles de développeur Python/Django dans des entreprises comme Octopus Energy ou Lengow.

9.2 Installation du framework Django

Avant de vous plonger dans la construction de votre application Django, vous devez mettre en place votre environnement de développement. Ce processus comprend l'installation de Python, la création d'un environnement virtuel et l'installation de Django. Toutefois il y a des prérequis qui devraient être respectées pour toute personne voulant travailler avec Django.

9.2.1 Conditions préalables

Avant de commencer à travailler avec Django, il est important d'avoir une compréhension de base de certains concepts. Voici ce que vous devez connaître :

- **Connaissance de base de Python :** Une bonne maîtrise des fondamentaux de Python – types de données, structures de contrôle, fonctions, classes, modules.
- **Bases de HTML, CSS et JavaScript :** Connaissances de base en HTML, CSS et JavaScript.
- **Compréhension des bases de données :** Concepts des bases de données – tables, enregistrements, clés primaires, clés étrangères. Des connaissances en SQL sont un plus.
- **Familiarité avec l'interface en ligne de commande (CLI) :** Connaissance de base des commandes Linux et Windows essentielles.

Avec ces prérequis, vous serez bien préparé pour comprendre et suivre les étapes nécessaires à l'installation de Django et à la création de vos applications web.

9.2.2 Installer Python

Pour faire fonctionner Django, vous devez installer Python et pip (Pip Installs Packages qui est un acronyme récursif) sur votre machine. A titre de rappel, rendez-vous à la section 1.2.1 pour revoir comment s'effectue l'installation de python.

Pour installer pip sur linux, exécutez la commande ci-dessous :

```
apt(dnf) install python3-pip
```

Avec PIP, c'est un monde de possibilités qui s'offre à vous et qui vous permettra de télécharger toutes les librairies de Python toutes hébergés sur le site http://pypi.org. Une fois Python et pip installé, vous êtes prêt à mettre en place un environnement virtuel.

9.2.3 Mise en place d'un environnement virtuel

Un environnement virtuel est un espace isolé dans lequel vous pouvez installer des paquets Python et des dépendances pour un projet spécifique. Cette méthode n'affecte pas l'environnement Python du système ni les autres projets.

Ceci est crucial lorsque l'on travaille avec Django, car différents projets peuvent nécessiter différentes versions de paquets. Un environnement virtuel permet de s'assurer que les dépendances de chaque projet sont bien contenues et n'entrent pas en conflit avec d'autres.

Pourquoi utiliser un environnement virtuel :

Isolation : Les dépendances de votre projet sont séparées des autres projets Python. Contrôle de version : Vous pouvez installer différentes versions de paquets dans chaque espace de travail virtuel, ce qui permet d'éviter les conflits entre les projets. Environnement plus propre : Il vous permet d'expérimenter avec des paquets sans affecter l'environnement global de Python sur votre serveur.

Suivez les étapes suivantes pour configurer un environnement virtuel python sur votre machine :

1. Installez le paquetage *venv* (s'il n'est pas déjà installé) :

   ```
   apt install python3-venv -y
   ```

2. Créez et naviguez jusqu'au répertoire dans lequel vous souhaitez stocker votre projet Django et créez un environnement virtuel :

   ```
   mkdir myprojectcd myproject
   ```

   Remplacez myproject par le nom que vous souhaitez donner à votre dossier de projet.

3. Créer un environnement virtuel à l'aide de *venv* :

   ```
   python3 -m venv env_name
   ```

4. Activer l'environnement virtuel :

   ```
   source env_name/bin/activate
   ```

   Une fois activée, l'invite de l'interpréteur de commandes change pour refléter l'environnement (par exemple, *(nom_env)*), indiquant que vous travaillez dans votre environnement virtuel.

   Cette configuration garantit que toutes les dépendances liées à Django sont isolées dans l'environnement virtuel, ce qui permet à votre serveur de rester propre et organisé.

9.2.4 Installer Django

Une fois votre environnement virtuel mis en place, l'étape suivante consiste à installer le framework web lui-même, ce que vous pouvez faire à l'aide du gestionnaire de paquets de Python, pip. L'installation de Django dans votre environnement virtuel permet de l'isoler des autres projets et de le gérer indépendamment.

Pour installer Django, suivez les étapes suivantes :

1. Une fois votre environnement virtuel activé, utilisez *pip* pour installer la dernière version de Django :

   ```
   pip install django
   ```

   Cela permet de télécharger et d'installer toutes les dépendances et le framework web Django dans l'environnement virtuel.
2. Utilisez la commande suivante pour confirmer que Django est installé et pour vérifier la version :

   ```
   django-admin --version
   ```

   Vous obtiendrez ainsi la version de Django actuellement installée dans votre environnement, ce qui confirme que l'installation s'est déroulée avec succès.

9.3 Création d'un nouveau projet Django

Il est temps de créer votre première application Django. Dans cette section, vous établirez la base de code nécessaire pour commencer à développer votre site web.

Créons un nouveau projet Django :

1. **Naviguez jusqu'au répertoire de votre projet** : Si vous ne vous trouvez pas déjà dans le répertoire où vous souhaitez créer le projet, utilisez la commande cd pour vous y rendre.
2. Exécutez la commande *startproject* : Utilisez la commande *startproject* de Django pour créer un nouveau projet. Remplacez *myproject* par le nom de votre projet :

   ```
   django-admin startproject myproject
   ```

   Cela créera un nouveau répertoire nommé *myproject* contenant la structure du projet par défaut.
3. Ensuite, naviguez jusqu'au dossier interne de votre projet :

   ```
   cd myproject/myproject
   ```

4. Ajoutez l'IP de votre serveur à *ALLOWED_HOSTS* : Utilisez l'un des éditeurs de texte par défaut d'Ubuntu (nano ou vim) et ouvrez le fichier *settings.py* :

```
vim settings.py
```

Localisez le paramètre *ALLOWED_HOSTS* et modifiez-le pour ajouter l'IP de votre serveur (Si vous developpez sur votre machine local, ce sera *127.0.0.1*) comme suit :

```
ALLOWED_HOSTS = ['votre\_ip']
```

5. Démarrez le serveur de développement : Pour vérifier que tout est correctement configuré, vous pouvez démarrer le serveur de développement de Django. Retournez dans la catégorie du projet parent et exécutez *runserver* :

```
python manage.py runserver
```

Maintenant, ouvrez votre navigateur et allez sur *http ://votre_ip :8000/*

Si tout a été fait correctement, vous devriez voir la page de bienvenue de Django :

FIGURE 9.1 – Page d'accueil d'un projet Django après installation

Si vous avez suivi ces étapes, vous avez construit et exécuté avec succès un projet Django par défaut sur votre machine (ou serveur). Le reste de ce tutoriel Django va plus loin dans le développement de l'application et la compréhension de Django en général.

9.3.1 Comprendre la structure du projet Django

L'exécution de la commande *startproject* crée une structure de projet Django de base qui comprend plusieurs fichiers et répertoires clés. Voici une décomposition de la structure et de l'objectif de chaque composant :

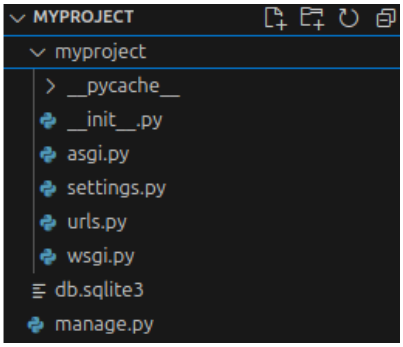


FIGURE 9.2 – Structure d'un projet Django

Chacun de ces fichiers et répertoires joue un rôle spécifique dans votre projet Django :

1. **manage.py** : Il s'agit d'un utilitaire en ligne de commande qui vous aide à interagir avec votre projet Django. Vous l'utilisez pour des tâches telles que l'exécution du serveur de développement, la migration des bases de données et la gestion de votre application.

   Voici à quoi il ressemble :

```
manage.py
1   #!/usr/bin/env python
2   """Django's command-line utility for administrative tasks."""
3   import os
4   import sys
5
6
7   def main():
8       """Run administrative tasks."""
9       os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')
10      try:
11          from django.core.management import execute_from_command_line
12      except ImportError as exc:
13          raise ImportError(
14              "Couldn't import Django. Are you sure it's installed and "
15              "available on your PYTHONPATH environment variable? Did you "
16              "forget to activate a virtual environment?"
17          ) from exc
18      execute_from_command_line(sys.argv)
19
20
21  if __name__ == '__main__':
22      main()
```

FIGURE 9.3 – Extrait du contenu du fichier manage.py

   Voici ses principales fonctions :

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')
```

   Définit la variable d'environnement pour indiquer à Django le fichier de configuration à utiliser.

```
execute_from_command_line(sys.argv)
```

   Exécute diverses commandes Django telles que *runserver*, *migrate* et autres.

2. **myproject/** : Le répertoire interne du projet qui contient les paramètres de base, les configurations et d'autres fichiers par défaut pour un projet Django.
3. **__init__.py** : Ce fichier marque le répertoire comme un paquetage Python, permettant au code d'être importé comme un module dans Python. Il est vide par défaut.
4. **settings.py** : Il s'agit de l'un des fichiers les plus importants de Django. Il contient tous les paramètres de configuration de votre projet, y compris les configurations de base de données, les applications installées, les intergiciels et les paramètres des fichiers statiques.
5. **urls.py** : Ce fichier définit les routes URL pour votre application. Il associe les URL aux vues qui traitent les demandes et renvoient les réponses.

```
myproject > urls.py
1  from django.contrib import admin
2  from django.urls import path
3
4  urlpatterns = [
5      path('admin/', admin.site.urls),
6  ]
```

FIGURE 9.4 – Contenu du fichier urls.py

Au fur et à mesure que vous ajouterez des fonctionnalités à votre application, vous étendrez *urlpatterns* pour inclure les routes de vos propres vues.

6. **asgi.py** : Ce fichier fournit un point d'entrée aux serveurs web compatibles *ASGI* pour servir votre projet. *ASGI* (Asynchronous Server Gateway Interface) est la norme pour les applications asynchrones de Django.

```
myproject > asgi.py
1  import os
2
3  from django.core.asgi import get_asgi_application
4
5  os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')
6
7  application = get_asgi_application()
```

FIGURE 9.5 – Extrait du contenu du fichier asgi.py

7. **wsgi.py** : Ce fichier est le point d'entrée pour les serveurs web compatibles *WSGI* tels que *Gunicorn* ou *uWSGI* pour servir votre projet dans un environnement de production.

```
myproject > wsgi.py
1  import os
2
3  from django.core.wsgi import get_wsgi_application
4
5  os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')
6
7  application = get_wsgi_application()
```

FIGURE 9.6 – Extrait du contenu du fichier wsgi.py

Comprendre le rôle de chaque fichier et répertoire vous permettra de modifier les paramètres, d'ajouter des URL et de gérer la structure de votre projet.

9.4 Comprendre les modèles et les bases de données de Django

Une fois votre projet Django mis en place et opérationnel, l'étape suivante consiste à se plonger dans l'un des composants centraux de toute application web : la gestion des données. Explorons comment Django gère les modèles et les bases de données afin d'alimenter des applications dynamiques et axées sur les données.

9.4.1 Création et migration de modèles

Dans Django, les modèles définissent la structure des tables de votre base de données. Il s'agit de classes Python qui correspondent directement aux tables de la base de données, ce qui vous permet d'interagir avec vos données de manière pratique. Chaque classe de modèle représente une table et chaque attribut de modèle représente un champ de la table.

Django fait abstraction du code SQL, ce qui facilite la création, la mise à jour et la gestion des bases de données sans avoir à écrire des requêtes SQL brutes.

Création d'un modèle

Pour créer un modèle, vous devez définir une classe Python dans le fichier *models.py* de l'une de vos applications. Chaque attribut de la classe correspond à un champ de la base de données et Django gère automatiquement la structure de la base de données sous-jacente.

Voici un exemple de modèle simple dans Django :

```
# Dans le fichier models.py de votre application
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13, unique=True)

    def __str__(self):
        return self.title
```

- title et author : Il s'agit de champs de caractères (CharField) d'une longueur maximale.
- published_date : Il s'agit d'un champ de date (DateField).
- isbn : Un champ de caractères avec une contrainte d'unicité.

La méthode **__str__()** garantit que lorsque vous demandez un livre, le titre du livre sera renvoyé sous la forme d'une chaîne de caractères.

Ajouter le modèle à la base de données

Une fois que vous avez créé votre modèle, Django doit le traduire en une table de base de données. Cela se fait par le biais des migrations, qui sont des fichiers que Django utilise pour garder une trace des modifications apportées à vos modèles.

Créer une migration : Pour créer le fichier de migration pour votre nouveau modèle, exécutez la procédure suivante :

```
python manage.py makemigrations
```

Django analysera les changements dans votre fichier models.py et générera un fichier de migration décrivant ces changements dans des instructions de type SQL.

Vous obtiendrez un résultat similaire à celui-ci :

```
Migrations for 'myapp':
  myapp/migrations/0001_initial.py
    - Create model Book
```

Appliquer la migration : Après avoir généré la migration, appliquez-la à la base de données à l'aide de la commande migrate :

```
python manage.py makemigrations
```

Cela créera les tables nécessaires dans votre base de données. Désormais, le modèle Book est représenté par une table Book correspondante dans votre base de données, et vous pouvez commencer à ajouter, interroger et gérer des données.

9.4.2 Mise en place de la base de données

Django prend en charge plusieurs bases de données, ce qui vous permet de choisir celle qui convient le mieux à votre projet. La base de données par défaut de Django est SQLite, qui est légère et facile à configurer, ce qui la rend idéale pour le développement et les petits projets.

Cependant, pour les environnements de production, des bases de données plus robustes telles que PostgreSQL, MySQL ou MariaDB sont recommandées en raison de leur évolutivité et de leurs performances.

Django fournit un support intégré pour toutes ces bases de données. Commençons par configurer la base de données SQLite par défaut.

Configuration de la base de données SQLite par défaut

Par défaut, Django est configuré pour utiliser SQLite, qui est préinstallé avec Python, de sorte qu'aucune configuration supplémentaire n'est nécessaire. Voici comment vous assurer que votre

projet est prêt à utiliser SQLite :

1. Accédez au fichier settings.py de votre projet, qui se trouve dans le dossier du projet (myproject → myproject → settings.py).
2. Allez dans la section DATABASES de settings.py, où Django aura déjà configuré SQLite comme base de données par défaut. Voici à quoi cela ressemble :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

   - **ENGINE** : Spécifie le back-end de la base de données (dans ce cas, SQLite).
   - **NAME** : Le chemin d'accès au fichier de base de données SQLite (db.sqlite3). Django crée automatiquement ce fichier lorsque vous exécutez la migration initiale.

3. Après avoir confirmé les paramètres de la base de données, vous devez appliquer les migrations initiales pour configurer les tables par défaut (comme les utilisateurs et les sessions). Exécutez la commande suivante :

```
python manage.py migrate
```

   Cette commande crée le fichier db.sqlite3 à la racine du répertoire de votre projet et met en place toutes les tables nécessaires à l'exécution de votre application.
4. Une fois les migrations terminées, vérifiez que le fichier db.sqlite3 a été créé dans le dossier de votre projet. Ce fichier stocke votre base de données localement et s'étoffera au fur et à mesure que vous ajouterez des données.

9.4.3 Intégration des bases de données

Une fois que vous avez configuré la base de données SQLite par défaut pour le développement, vous pouvez avoir besoin d'intégrer un système de gestion de base de données (SGBD) plus robuste au fur et à mesure que votre projet grandit. Django prend en charge plusieurs SGBD, dont PostgreSQL, MySQL, MariaDB et Oracle, chacun ayant ses propres atouts et son propre processus d'intégration.

Passons en revue chacune des options.

SQLite

C'est la base de données par défaut de Django et elle est idéale pour le développement et les petits projets. Elle est légère, basée sur des fichiers et ne nécessite pas d'installation au-delà de ce que Django fournit dans la boîte. Cependant, elle n'est pas adaptée aux environnements de production

car elle manque de fonctionnalités avancées telles que les écritures simultanées, l'extensibilité et la robustesse en cas de trafic important.

Installation : SQLite ne nécessite aucune installation supplémentaire dans Django ; il est prêt à l'emploi dès sa sortie de l'emballage.

Configuration : Django est préconfiguré pour utiliser SQLite avec les paramètres suivants dans settings.py :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Notez que SQLite ne requiert que 2 paramètres – ENGINE et NAME.

PostgreSQL

Une base de données puissante et open-source avec des fonctionnalités avancées telles que la prise en charge de requêtes complexes, de types de données comme JSON et de la recherche en texte intégral. Elle est fortement recommandée pour les applications Django de production en raison de son évolutivité et de son ensemble de fonctionnalités. PostgreSQL est connu pour adhérer strictement aux normes SQL et être fiable pour les applications complexes et à grande échelle.

Installation : PostgreSQL nécessite l'installation du paquetage psycopg2, qui agit comme un adaptateur entre Django et la base de données PostgreSQL.

Configuration : Outre les paramètres ENGINE et NAME par défaut pour SQLite, PostgreSQL requiert également USER, PASSWORD, HOST et PORT.

Tous ces paramètres sont configurés dans le fichier settings.py, dans la section DATABASES :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Différences : Contrairement à SQLite, PostgreSQL est une base de données complète qui prend en charge plusieurs utilisateurs simultanés, des requêtes avancées et des transactions. Vous devrez installer PostgreSQL sur votre serveur, créer une base de données et un utilisateur, et gérer des configurations plus complexes.

MySQL / MariaDB

Les deux sont des bases de données relationnelles rapides et fiables qui sont largement utilisées dans le développement web. MySQL est connu pour sa vitesse, tandis que **MariaDB**, une version dérivée de MySQL, se concentre davantage sur le développement communautaire et est entièrement compatible avec MySQL. Ces bases de données sont excellentes pour les applications web à fort trafic et sont souvent privilégiées pour leurs performances.

Installation : MySQL et MariaDB utilisent tous deux le package mysqlclient pour s'interfacer avec Django.

Configuration : Les deux utilisent les mêmes paramètres que PostgreSQL, configurés dans le fichier settings.py, dans la section DATABASES :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',  # or 'django.db.backends.mariadb'
        ↪  for MariaDB
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Différences : MySQL et MariaDB sont réputés pour leurs performances avec les grands ensembles de données et les applications web à fort trafic. Ils prennent en charge plusieurs moteurs de stockage, des lectures rapides et peuvent traiter rapidement des millions de requêtes.

Oracle / Microsoft SQL Server

Ces bases de données de qualité professionnelle offrent une puissance considérable, en particulier pour les grandes applications web. Cependant, leur configuration peut être plus complexe et elles ne sont généralement utilisées que dans des environnements d'entreprise spécifiques où l'entreprise utilise déjà Oracle ou SQL Server pour d'autres logiciels.

Installation : Les deux requièrent des pilotes supplémentaires pour l'intégration avec Django, tels que cx_Oracle pour Oracle ou pyodbc pour SQL Server. Ces bases de données sont généralement utilisées dans des environnements d'entreprise où d'autres logiciels reposent sur ces systèmes.

Configuration : Chacun de ces systèmes nécessite des processus de configuration plus complexes, notamment l'installation du serveur, la configuration des pilotes et la configuration des informations d'identification de la base de données dans le settings.py, comme pour les options précédentes.

Différences : Oracle et SQL Server offrent des fonctionnalités de niveau entreprise telles que la gestion avancée des transactions, l'optimisation des performances et l'intégration avec des systèmes à grande échelle. Leur configuration est plus complexe et est souvent confiée à des administrateurs de bases de données spécialisés.

Alors, lequel choisir ?

- **Développement et tests :** Utilisez SQLite. Il est simple, ne nécessite aucune installation et est idéal pour les tests locaux.
- **Production à petite échelle :** Pour les petites applications web à faible trafic, vous pouvez choisir **MySQL** ou MariaDB, qui offrent de meilleures performances que SQLite et dont la courbe d'apprentissage est plus facile que celle de PostgreSQL.
- **Production à grande échelle :** PostgreSQL est le plus recommandé pour les projets Django qui nécessitent de l'évolutivité, de la sécurité et des fonctionnalités avancées.
- **Environnements d'entreprise :** Si vous travaillez dans une entreprise qui utilise Oracle ou SQL Server pour d'autres applications, ces bases de données peuvent être le meilleur choix en raison des exigences d'intégration.

Configuration de PostgreSQL avec Django

Dans le cadre du cours, notre utiliserons le SGBD PostgreSQL. Nous verrons donc comment l'intégrer et travailler dans un projet Django.

1. **Installer PostgreSQL :** Pour intégrer PostgreSQL à Django, vous devez d'abord installer PostgreSQL sur votre système :

   ```
   sudo apt update
   ```

   ```
   sudo apt install postgresql postgresql-contrib
   ```

2. **Installer l'adaptateur PostgreSQL pour Django :** Django utilise le package psycopg2 pour interagir avec PostgreSQL. Installez-le à l'aide de pip :

   ```
   pip install psycopg2
   ```

3. **Passez à l'utilisateur PostgreSQL :**

```
sudo -u postgres psql
```

4. **Créer une nouvelle base de données :**

```
CREATE DATABASE mydatabase;
```

5. **Créer un utilisateur avec un mot de passe :**

```
CREATE USER mydatabaseuser WITH PASSWORD 'mypassword';
```

6. Accorder à mydatabaseuser tous les privilèges sur la base de données :

```
GRANT ALL PRIVILEGES ON DATABASE mydatabase TO mydatabaseuser;
```

7. **Quitter le shell PostgreSQL :**

```
\q
```

8. **Configurez Django pour utiliser PostgreSQL :** Ouvrez votre fichier settings.py et localisez la section DATABASES. Modifiez-la pour PostgreSQL comme suit :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',  # Utilisez '127.0.0.1' si 'localhost' ne marche
        ↪  pas.
        'PORT': '5432',       # 5432 est le port par défaut de PostgreSQL.
    }
}
```

Les valeurs PASSWORD et USER doivent correspondre à celles que vous avez définies à **l'étape 5**.

9. **Appliquer les migrations :** Maintenant que votre base de données est configurée, vous devez appliquer les migrations de votre projet pour configurer les tables nécessaires dans PostgreSQL :

```
python manage.py migrate
```

Cette commande créera les tables nécessaires dans votre base de données PostgreSQL.

10. **Tester la connexion :** Enfin, exécutez le serveur de développement de Django à l'aide de la commande runserver, afin de vérifier que l'intégration de PostgreSQL est réussie :

```
python manage.py runserver
```

Ouvrez http://127.0.0.1:8000/ dans votre navigateur. Si la page d'accueil de Django s'affiche sans erreur, votre connexion à la base de données fonctionne correctement.

Avec votre base de données PostgreSQL entièrement intégrée, Django est maintenant prêt à gérer des ensembles de données plus importants et plus complexes, ce qui rend votre projet évolutif et prêt pour la production.

9.5 Construire des vues et des templates

Une fois la base de données et les modèles mis en place pour gérer les données de votre application, l'étape suivante consiste à afficher ces informations aux utilisateurs et à créer des interfaces interactives.

C'est là qu'interviennent les vues et les templates, qui vous permettent de définir la logique et la structure des pages web avec lesquelles les utilisateurs interagiront.

9.5.1 Comprendre les vues

Dans Django, les vues sont un composant essentiel qui définit la logique derrière ce qu'un utilisateur voit lorsqu'il interagit avec votre application.

En termes simples, une vue prend une requête web, la traite et renvoie une réponse, généralement sous la forme d'une page HTML rendue. Les vues font le lien entre votre base de données et l'interface utilisateur grâce à l'utilisation de modèles et de gabarits.

Il existe deux types principaux de vues dans Django : **Les vues basées sur les fonctions (FBV)** et **les vues basées sur les classes (CBV)**. Ces deux types de vues ont le même objectif, mais offrent différentes manières d'organiser le code.

Vues basées sur les fonctions (FBV)

Les vues basées sur des fonctions sont des fonctions Python simples qui acceptent un objet de requête et renvoient une réponse. Elles sont souvent utilisées pour des cas d'utilisation plus simples ou lorsque vous avez besoin de plus de contrôle sur le flux de la logique.

Voici un exemple d'un simple **FBV** qui rend un modèle HTML :

```
from django.shortcuts import render

# Vue basée sur des fonctions
def homepage(request):
    return render(request, 'homepage.html')
```

Dans cet exemple :

- **request :** L'objet de la requête HTTP envoyé par le navigateur de l'utilisateur.

- **render :** Combine le modèle homepage.html avec les données contextuelles fournies et renvoie une réponse HTTP.

Les **FBV** sont faciles à comprendre et à mettre en œuvre, mais lorsque les vues deviennent plus complexes, par exemple lorsqu'elles impliquent des formulaires ou l'authentification, elles peuvent devenir plus difficiles à maintenir.

Vues par classe (CBV)

Les vues basées sur les classes, quant à elles, vous permettent de structurer les vues plus efficacement en utilisant les principes de la programmation orientée objet. Les **CBV** sont plus flexibles et réutilisables puisqu'elles sont basées sur les classes Python.

Django fournit des vues génériques intégrées, comme ListView et DetailView, qui gèrent les cas d'utilisation courants. Ces vues vous permettent de vous concentrer sur leur personnalisation plutôt que sur la réécriture d'une logique répétitive.

Voici un exemple d'un **CBV** qui remplit la même fonction que le **FBV** précédent :

```
from django.views.generic import TemplateView

# Vue basée sur une classe
class HomePageView(TemplateView):
    template_name = 'homepage.html'
```

Dans cet exemple :

- **TemplateView :** Une vue générique intégrée qui rend un modèle.
- **template_name :** Spécifie le fichier modèle à rendre, comme dans l'exemple **FBV**.

Pour utiliser cette **CBV** dans la configuration de l'URL, vous devez inclure la méthode .as_view() lorsque vous l'ajoutez à urls.py :

```
from django.urls import path
from .views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

Quand utiliser les FBV ou les CBV ?

- Utilisez les FBV lorsque vous avez besoin de simplicité et d'un contrôle total sur la logique de votre vue. Ils sont souvent plus intuitifs pour les petites vues ou les logiques personnalisées qui ne correspondent pas aux modèles de vues prédéfinis de Django.
- Utilisez les CBV lorsque vous souhaitez suivre les principes DRY (Don't Repeat Yourself) et réutiliser la logique existante. Les CBV sont idéales lorsque vos vues deviennent de plus

en plus complexes, par exemple lorsque vous gérez des formulaires, des données paginées ou des opérations CRUD standard.

Les FBV et les CBV font tous deux partie intégrante de Django, et le choix entre les deux dépend des exigences de votre projet et de vos préférences personnelles.

9.5.2 Travailler avec des templates

Le système de templates de Django est conçu pour séparer la présentation des données de la logique sous-jacente. Les **templates sont essentiellement des fichiers HTML** qui contiennent un contenu dynamique, vous permettant d'afficher des données passées depuis vos vues. Le langage de template de Django (DTL) fournit des placeholders et de la logique, comme des boucles et des conditionnelles, pour insérer des données dynamiques dans le HTML.

Les templates jouent un rôle crucial dans la définition de l'aspect et de la convivialité de vos pages web, tout en maintenant une séparation nette entre la logique, qui est gérée par les vues, et l'interface utilisateur.

Création d'un template

Pour créer un template dans Django, vous devez créer un fichier HTML dans un répertoire templates. La structure du répertoire peut ressembler à ceci :

```
myproject/
│
├── myapp/
│   ├── templates/
│   │   └── myapp/
│   │       └── homepage.html
│   └── views.py
│
└── manage.py
```

FIGURE 9.7 – Exemple de structuration des templates dans un projet Django

Dans cette structure, Django recherche automatiquement les templates dans le répertoire templates de votre application. Vous pouvez également organiser les templates dans des dossiers spécifiques à l'application (comme *myapp/homepage.html*), en veillant à ce que chaque application ait ses propres templates.

Voici un exemple de template simple (*homepage.html*) :

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
```

```
<body>
    <h1>Welcome to {{ site_name }}</h1>
    <p>{{ message }}</p>
</body>
</html>
```

Dans ce template, *{{ site_name }}* et *{{ message }}* sont des espaces réservés qui seront remplis avec des données dynamiques transmises par la vue.

Rendu d'un template dans une vue

Pour effectuer le rendu d'un template, vous devez le passer à la vue à l'aide de la fonction render() de Django. Voici un exemple de rendu du template homepage.html avec quelques données contextuelles dans une **vue basée sur une fonction (FBV)** :

```
from django.shortcuts import render

def homepage(request):
    context = {
        'site_name': 'My Django Website',
        'message': 'This is the homepage content!'
    }
    return render(request, 'myapp/homepage.html', context)
```

Dans cet exemple :

- render() : Combine le template (homepage.html) avec les données contextuelles.
- context : Un dictionnaire contenant les données (site_name et message) à insérer dans le template.

Lorsqu'un utilisateur visite cette page, les espaces réservés *{{ site_name }}* et *{{ message }}* du template seront remplacés respectivement par "My Django Website" et "This is the homepage content !".

Balises et filtres de templates

Les templates Django prennent également en charge les **balises de template** et les **filtres**, qui vous permettent d'ajouter du contenu dynamique et de la logique dans vos fichiers HTML.

- **Balises de templates :** Effectuer des opérations logiques à l'intérieur des templates, telles que des boucles et des conditions. Par exemple, une condition "if" :

  ```
  {\% if user.is\_authenticated \%}
      <p>Welcome back, {{ user.username }}!</p>
  {\% else \%}
      <p>Hello, Guest!</p>
  {\% endif \%}
  ```

- **Filtres de templates :** Modifier les variables à des fins d'affichage. Par exemple, formater des dates à l'aide d'un filtre :

```
<p>Published on {{ article.published_date|date:"F j, Y" }}</p>
```

Extension des templates

Django prend en charge l'héritage de templates, ce qui permet de créer des templates de base que d'autres templates peuvent étendre. Ceci est particulièrement utile pour maintenir une mise en page cohérente sur plusieurs pages, comme un en-tête et un pied de page communs.

1. **Créer un template de base (base.html)**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{\% block title \%}My Site{\% endblock \%}</title>
</head>
<body>
    <header>
        <h1>Site Header</h1>
    </header>

    <main>
        { \% block content \%}
        <!-- Page-specific content goes here -->
        { \% endblock \%}
    </main>

    <footer>
        <p>Footer Content</p>
    </footer>
</body>
</html>
```

Étendre le template de base dans un template enfant (homepage.html)

```
{\% extends 'base.html' \%}

{\% block title \%}Home Page{\% endblock \%}

{\% block content \%}
    <h2>Welcome to the homepage!</h2>
    <p>This is the homepage content.</p>
```

```
{\% endblock \%}
```

En étendant le template de base, vous pouvez réutiliser l'en-tête et le pied de page sur différentes pages, en conservant une mise en page cohérente et en évitant les répétitions.

Les templates vous permettent également de faire référence à des fichiers statiques (tels que CSS, JavaScript ou des images) à l'aide de la balise de template {% static %}. Par exemple, pour inclure un fichier CSS :

```
<link rel="stylesheet" type="text/css" href="\{\% static 'css/style.css' \%\}">
```

Assurez-vous d'avoir configuré correctement vos répertoires de fichiers statiques dans settings.py :

```
STATIC_URL = '/static/'
```

En utilisant efficacement le système de templates de Django, vous pouvez créer des pages web dynamiques, basées sur des données, et maintenir une séparation nette entre la logique et la présentation.

9.6 Travailler avec des modèles d'URL

Le distributeur d'URL de Django s'appuie sur des expressions régulières ou des convertisseurs de chemin pour faire correspondre les demandes d'URL entrantes avec des vues spécifiques. Ces motifs d'URL sont définis dans urls.py, et Django utilise cette configuration pour déterminer quelle vue doit traiter une requête particulière.

Chaque projet Django possède généralement deux niveaux de configuration d'URL :

1. **URL au niveau du projet :** Défini dans le fichier racine urls.py (qui se trouve dans le répertoire principal du projet), ce fichier contient la structure globale du routage des URL du projet.
2. **URLs au niveau de l'application :** Chaque application de votre projet peut avoir son propre fichier urls.py, ce qui permet une gestion modulaire des URL. Ces URLs au niveau de l'application sont ensuite liées aux URLs au niveau du projet.

9.6.1 Configuration de base de l'URL

Voyons comment un projet simple peut mettre en place des modèles d'URL :

URLs au niveau du projet (myproject → urls.py)

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),  # Site d'admin
    path('', include('myapp.urls')),  # Inclure les URLs au niveau de
    ↪  l'application pour "myapp"
]
```

- **path() :** Fonction de Django permettant de définir des modèles d'URL.
- **admin.site.urls :** Gère toutes les requêtes liées à l'administration.
- **include() :** Utilisée pour inclure les URL au niveau de l'application (à partir de myapp.urls) dans la configuration du projet principal.

URLs au niveau de l'application (myproject → myapp → urls.py)

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.homepage, name='home'),  # Associe l'URL racine à la vue
    ↪  homepage
    path('about/', views.about, name='about'),  # Associe "/about/" à la vue about
]
```

- **L'URL racine (représentée par ' ')** est associée à la vue *homepage*.
- **L'URL /about/** est associée à la vue about.

Routage d'URL avec des vues basées sur des fonctions (FBV)

Voici à quoi pourraient ressembler les vues correspondantes dans views.py pour la configuration d'URL ci-dessus :

```
from django.shortcuts import render

def homepage(request):
    return render(request, 'homepage.html')

def about(request):
    return render(request, 'about.html')
```

Cette configuration lie la page d'accueil "homepage" et l'URL "about" à leurs vues respectives, ce qui garantit que lorsqu'un utilisateur visite *http ://example.com/* ou *http ://example.com/about/*, le template correct est affiché.

Routage d'URL avec des vues basées sur des classes (CBV)

Vous pouvez également faire correspondre des URL à des **vues basées sur des classes (CBV)**, qui sont gérées de manière légèrement différente. Voici un exemple de mappage d'une URL **CBV** :

```
from django.urls import path
from .views import HomePageView, AboutPageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
    path('about/', AboutPageView.as_view(), name='about'),
]
```

Et les vues basées sur les classes correspondantes dans views.py :

```
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'homepage.html'

class AboutPageView(TemplateView):
    template_name = 'about.html'
```

Dans ce cas, la méthode .as_view() est utilisée pour lier les vues basées sur les classes aux modèles d'URL.

Paramètres d'URL et routage dynamique

Django vous permet également de capturer des paramètres d'URL et de les passer aux vues, ce qui est utile pour le mappage dynamique d'URL pour des choses comme des articles de blog ou des pages de produits.

Voici un exemple où l'URL capture un paramètre entier :

```
from django.urls import path
from . import views

urlpatterns = [
    path('article/<int:id>/', views.article_detail, name='article_detail'),
]
```

Dans ce cas, *<int :id>* capture un nombre entier à partir de l'URL et le transmet à la vue *article_detail* en tant qu'argument :

```
def article_detail(request, id):
    # Fetch the article using the provided ID
    article = get_object_or_404(Article, pk=id)
    return render(request, 'article_detail.html', {'article': article})
```

Si un utilisateur visite *http ://example.com/article/5/*, Django transmettra 5 comme argument id à la vue *article_detail*, ce qui vous permettra de récupérer et d'afficher l'article en question.

Modèles d'URL nommés

Il est conseillé de nommer vos modèles d'URL. Cela vous permet de référencer les URL par leur nom dans vos templates et vos vues plutôt que de les coder en dur.

Dans l'exemple urls.py ci-dessus, chaque URL a un argument name :

```
path('about/', views.about, name='about')
```

Dans vos templates, vous pouvez alors utiliser la balise template *{% url %}* pour faire référence à ces URL nommées :

```
<a href="{% url 'about' %}">About Us</a>
```

Ainsi, si l'URL *about* change un jour, il suffit de la mettre à jour dans la configuration de l'URL plutôt que dans chaque template.

En configurant et en utilisant les modèles d'URL de Django, vous pouvez faire correspondre efficacement différentes URL à leurs vues respectives. Cette flexibilité vous permet de gérer à la fois des pages statiques simples et un contenu dynamique plus complexe, adapté à la saisie de l'utilisateur.

9.7 Gestion des fichiers statiques dans Django

Nous avons mis en place des vues, des templates et des modèles d'URL. Maintenant, pour construire une application Django pleinement fonctionnelle, nous devons commencer à manipuler des fichiers statiques, tels que CSS, JavaScript et des images.

Heureusement, Django offre un moyen simple et efficace de gérer les fichiers statiques, en s'assurant qu'ils sont correctement servis et organisés pendant le développement et le déploiement.

9.7.1 Mise en place de fichiers statiques dans Django

Pour gérer les fichiers statiques dans Django, vous devez configurer votre projet pour qu'il sache où les rechercher et les servir.

Définir l'URL statique et le répertoire dans settings.py :

```
# settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    BASE_DIR / "static",  # Optional: Add this if you want to store static files
    ↪  in a 'static' folder within your project
]
```

- **STATIC_URL** : Définit le préfixe de l'URL que Django utilisera pour servir les fichiers statiques.
- **STATICFILES_DIRS** : Spécifie des répertoires supplémentaires dans lesquels Django cherchera des fichiers statiques pendant le développement.

Par défaut, Django recherche également les fichiers statiques dans le répertoire static de chaque application. Par exemple, si vous avez une application nommée myapp, vous pouvez créer une structure de dossiers comme celle-ci :

```
myapp/
│
├── static/
│   └── myapp/
│       └── style.css
└── views.py
```

FIGURE 9.8 – Exemple d'organisation des fichiers statiques

Django reconnaîtra et servira automatiquement ces fichiers pendant le développement.

9.7.2 Utilisation de fichiers statiques dans les templates

Pour utiliser des fichiers statiques dans vos templates, vous devez d'abord charger la balise de template {% static %} de Django. Ensuite, vous pouvez référencer vos fichiers statiques par leurs chemins relatifs :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home Page</title>
    {% load static %}
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
</head>
<body>
    <h1>Welcome to My Django App</h1>
    <img src="{% static 'myapp/images/logo.png' %}" alt="Logo">
</body>
</html>
```

Dans cet exemple :

- {% load static %} : Charge la balise de template des fichiers statiques.
- {% static 'myapp/style.css' %} : Résout le chemin vers le fichier style.css dans le répertoire static/myapp.
- {% static 'myapp/images/logo.png' %} : Résout le chemin d'accès à un fichier image.

9.7.3 Collecte des fichiers statiques pour le déploiement

Pendant le développement, Django sert automatiquement les fichiers statiques lorsque DEBUG = True. Cependant, en production, Django ne sert pas les fichiers statiques directement. Au lieu de cela, vous devez collecter tous les fichiers statiques à partir de différents emplacements et les stocker dans un seul répertoire.

Pour ce faire, Django fournit la commande de gestion *collectstatic* :

1. Assurez-vous que *django.contrib.staticfiles* est inclus dans le paramètre INSTALLED_APPS :

   ```
   INSTALLED_APPS = [
       # Other installed apps
       'django.contrib.staticfiles',
   ]
   ```

2. Configurez le répertoire racine statique dans settings.py :

   ```
   STATIC_ROOT = BASE_DIR / 'staticfiles'
   ```

   STATIC_ROOT : Définit le répertoire dans lequel Django collectera les fichiers statiques pour le déploiement.
3. Exécutez la commande *collectstatic* :

   ```
   STATIC_ROOT = BASE_DIR / 'staticfiles'
   ```

   Cette commande collecte tous les fichiers statiques du répertoire static de chaque application et de tout autre emplacement spécifié dans STATICFILES_DIRS, puis les stocke dans le répertoire STATIC_ROOT.

9.7.4 Servir des fichiers statiques en production

Dans un environnement de production, vous utilisez généralement un serveur web (comme Nginx ou Apache) pour servir les fichiers statiques. Django recommande de configurer votre serveur web pour qu'il serve efficacement le contenu statique, réduisant ainsi la charge sur votre application Django.

Voici comment vous pourriez configurer le service de fichiers statiques dans une installation Nginx :

```
server {
    listen 80;
    server_name yourdomain.com;

    location /static/ {
        alias /path/to/your/project/staticfiles/;  # The path to STATIC_ROOT
    }

    location / {
        proxy_pass http://127.0.0.1:8000;  # Pass requests to Django app
    }
}
```

Dans cette configuration, Nginx sert les fichiers statiques directement depuis le répertoire /static/, tout en transmettant toutes les autres requêtes à l'application Django.

En gérant correctement les fichiers statiques dans Django, vous vous assurez que votre application n'est pas seulement belle avec des styles et des images, mais qu'elle fonctionne aussi efficacement lorsqu'elle est déployée.

9.8 Mise en œuvre de l'authentification et de l'autorisation des utilisateurs

Au fur et à mesure que nous créons des applications web plus dynamiques et plus complexes, la nécessité de gérer les utilisateurs, de sécuriser le contenu et de contrôler l'accès devient critique.

Django fournit un système d'authentification intégré qui facilite la gestion de la connexion, de la déconnexion, de l'enregistrement et des autorisations des utilisateurs. Cela permet à un développeur Django de gérer efficacement l'authentification et l'autorisation des utilisateurs.

9.8.1 Authentification de l'utilisateur

Le système d'authentification intégré de Django simplifie la gestion des utilisateurs avec des fonctionnalités telles que la connexion, la déconnexion, la gestion des mots de passe et l'enregistrement. Par défaut, Django offre une fonctionnalité de modèle d'utilisateur et des vues d'authentification qui gèrent la plupart de ces tâches.

Configuration de la connexion et de la déconnexion des utilisateurs

Pour mettre en œuvre les fonctionnalités de connexion et de déconnexion, vous pouvez utiliser les vues intégrées de Django qui se trouvent dans django.contrib.auth.views. Ces vues gèrent le processus d'authentification et simplifient le code nécessaire à ces opérations.

1. Configurez les URLs de connexion et de déconnexion dans votre fichier urls.py :

```
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

- **LoginView :** Gère la connexion de l'utilisateur.
- **LogoutView :** Déconnecte l'utilisateur et le redirige vers la page spécifiée.

2. **Créez un template de formulaire de connexion :** Vous pouvez créer un template de connexion simple (login.html) dans le répertoire templates de votre application :

```
<form method="post">
    {% csrf_token %}
    <div>
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
    </div>
    <div>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
    </div>
    <button type="submit">Login</button>
</form>
```

Ce template gère l'authentification de l'utilisateur lorsqu'un nom d'utilisateur et un mot de passe valides sont soumis.

3. **Créer la déconnexion :** Vous n'avez pas besoin d'un template séparé pour vous déconnecter, car la LogoutView de Django s'en charge. Vous pouvez simplement ajouter un lien de déconnexion dans votre template :

```
<a href="{% url 'logout' %}">Logout</a>
```

Création d'une vue d'enregistrement des utilisateurs

Django ne fournit pas de vue d'enregistrement intégrée par défaut, mais vous pouvez en créer une en utilisant UserCreationForm de Django :

1. Créer une vue d'enregistrement dans **views.py** :

```
from django.shortcuts import render, redirect
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import login
```

```
def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)  # Automatically log the user in after
            ↪  registration
            return redirect('home')  # Redirect to home page after successful
            ↪  registration
    else:
        form = UserCreationForm()
    return render(request, 'register.html', {'form': form})
```

2. Configurez l'URL pour l'enregistrement dans **urls.py** :

```
urlpatterns = [
    path('register/', views.register, name='register'),
]
```

3. Créez le template de formulaire d'inscription (**register.html**) :

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Register</button>
</form>
```

Cette vue d'enregistrement permet aux utilisateurs de créer un compte à l'aide d'un simple formulaire. Une fois l'inscription réussie, les utilisateurs sont automatiquement connectés.

9.8.2 Autorisation de l'utilisateur

L'autorisation de l'utilisateur permet de contrôler l'accès à certaines vues et actions en fonction des permissions ou des rôles de l'utilisateur. Le système de permissions de Django facilite la gestion des utilisateurs qui peuvent accéder à des parties spécifiques de votre application.

Restreindre l'accès à l'aide d'une connexion obligatoire

Pour restreindre l'accès à certaines vues, vous pouvez utiliser le décorateur @login_required de Django, qui garantit que seuls les utilisateurs authentifiés peuvent accéder à une vue :

1. **Utilisation de @login_required dans les vues :**

```
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request, 'dashboard.html')
```

Cela garantit que si un utilisateur n'est pas connecté, il sera redirigé vers la page de connexion.

2. Application de @login_required dans les URL : Vous pouvez également appliquer @login_required directement dans votre fichier urls.py en utilisant le LoginRequiredMixin de Django pour les vues basées sur des classes :

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import TemplateView

class DashboardView(LoginRequiredMixin, TemplateView):
    template_name = 'dashboard.html'
```

Gestion des autorisations des utilisateurs

Django vous permet d'attribuer différentes permissions aux utilisateurs et aux groupes, qui peuvent être utilisées pour contrôler l'accès à des vues ou à des actions spécifiques. Les permissions peuvent être ajoutées aux modèles d'utilisateurs et vérifiées dans vos vues.

1. **Attribution de permissions :** Lors de la création d'utilisateurs ou de groupes, vous pouvez attribuer des autorisations. Par exemple, un administrateur peut avoir le droit d'ajouter ou de supprimer du contenu, tandis que les utilisateurs ordinaires ne peuvent que consulter le contenu.
2. **Vérification des autorisations dans les vues :** Vous pouvez vérifier si un utilisateur dispose des autorisations requises avant d'autoriser l'accès à certaines vues à l'aide du décorateur @permission_required :

```
from django.contrib.auth.decorators import permission_required

@permission_required('app.add_article')
def add_article(request):
    # View code for adding an article
```

Seuls les utilisateurs disposant de l'autorisation add_article pourront accéder à cette vue.

Restreindre l'accès en fonction des rôles des utilisateurs

La **fonction de groupes de Django** vous permet de créer des rôles (tels que "editor" ou "viewer") et d'assigner des utilisateurs à ces groupes. Vous pouvez ensuite vérifier l'appartenance d'un utilisateur à un groupe pour en contrôler l'accès.

1. Affectation d'utilisateurs à des groupes : Les groupes peuvent être créés dans le tableau d'administration de Django ou par programme :

```
from django.contrib.auth.models import Group, User

group = Group.objects.get(name='editors')
user = User.objects.get(username='john')
user.groups.add(group)  # Affecte l'utilisateur au groupe 'editors'
```

2. **Vérification de l'appartenance à un groupe dans les vues :** Vous pouvez restreindre les vues en fonction de l'appartenance à un groupe :

```
from django.contrib.auth.decorators import user_passes_test

def is_editor(user):
    return user.groups.filter(name='editors').exists()

@user_passes_test(is_editor)
def editor_dashboard(request):
    return render(request, 'editor_dashboard.html')
```

   Dans cet exemple, seuls les utilisateurs appartenant au groupe "editors" peuvent accéder à la vue editor_dashboard.

En tirant parti des systèmes d'authentification et d'autorisation intégrés à Django, vous pouvez gérer en toute sécurité la connexion, l'enregistrement et les autorisations des utilisateurs, en veillant à ce que les contenus sensibles soient protégés et que l'accès soit correctement contrôlé.

9.9 Gestion des formulaires dans Django

Après la mise en œuvre de l'authentification et de l'autorisation de l'utilisateur, la prochaine fonctionnalité cruciale de toute page web est le traitement des entrées de l'utilisateur par le biais de formulaires. Django fournit un système puissant et flexible pour créer, traiter et valider les formulaires.

Le système de gestion des formulaires de Django est construit sur des classes Python qui représentent les formulaires comme des objets. Ces formulaires peuvent être rendus dans des templates, validés et traités en utilisant seulement quelques lignes de code. Les formulaires peuvent être créés manuellement à l'aide de la classe Form de Django ou générés automatiquement à partir de modèles à l'aide de ModelForm.

9.9.1 Création de formulaires à l'aide de la classe Form de Django

Pour créer un formulaire dans Django, vous commencez par définir une classe de formulaire dans votre fichier forms.py. Cette classe définira les champs et les règles de validation.

1. **Définition d'un formulaire :** Voici un exemple de formulaire de contact simple :

```
# forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100, label='Votre Nom')
    email = forms.EmailField(label='Votre Email')
    message = forms.CharField(widget=forms.Textarea, label='Votre Message')
```

   Ce formulaire comprend trois champs : le nom, l'adresse électronique et le message. Chaque champ est représenté par une classe de champ de formulaire, telle que CharField pour le texte court, EmailField pour les courriels et Textarea pour les entrées de texte plus longues.

2. **Afficher le formulaire dans un template :** Pour afficher le formulaire sur une page web, vous pouvez le rendre dans un template. Vous pouvez rendre le formulaire manuellement ou utiliser les aides au rendu de formulaire intégrées à Django :

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Envoyer le Message</button>
</form>
```

   - {% csrf_token %} : Ajoute un jeton CSRF pour se protéger contre les attaques de type "cross-site request forgery".
   - {{ form.as_p }} : Rend les champs du formulaire avec chaque champ enveloppé dans une balise <p> pour faciliter la mise en forme.

3. **Traitement des soumissions de formulaires :** Dans la vue qui gère la soumission du formulaire, vous devrez valider les données du formulaire et les traiter si le formulaire est valide :

```
# views.py
from django.shortcuts import render
from .forms import ContactForm

def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Traiter les données, par exemple envoyer un e-mail ou les
            ↪ enregistrer dans la base de données
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            # Rediriger vers une page de succès ou afficher un message de
            ↪ succès
            return render(request, 'thanks.html')
    else:
        form = ContactForm()
```

```
    return render(request, 'contact.html', {'form': form})
```

Dans cette vue :

- Si le formulaire est soumis via une requête POST, les données du formulaire sont validées.
- form.is_valid() : Vérifie si les données du formulaire sont validées.
- form.cleaned_data : Contient les données validées du formulaire qui peuvent maintenant être traitées (par exemple, envoi d'un courrier électronique).
- Si la requête n'est pas un POST, un formulaire vide est affiché pour que l'utilisateur puisse le remplir.

9.9.2 Validation des formulaires et traitement des erreurs

Les formulaires de Django sont dotés d'une validation intégrée qui garantit que les données soumises sont correctes et sécurisées. Les champs tels que EmailField valident automatiquement que l'entrée est une adresse électronique valide. Vous pouvez également créer une logique de validation personnalisée pour appliquer des règles spécifiques.

Validation intégrée

Django effectue automatiquement certains contrôles de validation pour vous :

- **Champs obligatoires :** Par défaut, tous les champs sont obligatoires, sauf si vous spécifiez required=False.
- **Types de champs :** Les champs tels que EmailField, IntegerField et URLField valident automatiquement le type de données saisies.

Validation personnalisée

Vous pouvez également définir une logique de validation personnalisée en remplaçant la méthode clean() ou en créant des validateurs de champs personnalisés.

1. Ajout d'un validateur personnalisé pour le champ name :

```
# forms.py
from django import forms
from django.core.exceptions import ValidationError

class ContactForm(forms.Form):
```

```
    name = forms.CharField(max_length=100, label='Your Name')
    email = forms.EmailField(label='Your Email')
    message = forms.CharField(widget=forms.Textarea, label='Your Message')

    def clean_name(self):
        name = self.cleaned_data.get('name')
        if "spam" in name.lower():
            raise ValidationError("Invalid name: 'spam' is not allowed.")
        return name
```

Dans cet exemple, si le nom contient le mot *spam*, Django lèvera une erreur de validation.

2. **Gestion des erreurs de formulaire dans les templates :** Vous pouvez personnaliser l'affichage des erreurs de formulaire dans les templates. Django ajoute automatiquement des messages d'erreur à chaque champ, que vous pouvez afficher comme suit :

```
<form method="post">
    {% csrf_token %}
    <div>
        {{ form.name.label_tag }} {{ form.name }}
        {% if form.name.errors %}
            <div class="error">{{ form.name.errors }}</div>
        {% endif %}
    </div>
    <div>
        {{ form.email.label_tag }} {{ form.email }}
        {% if form.email.errors %}
            <div class="error">{{ form.email.errors }}</div>
        {% endif %}
    </div>
    <div>
        {{ form.message.label_tag }} {{ form.message }}
        {% if form.message.errors %}
            <div class="error">{{ form.message.errors }}</div>
        {% endif %}
    </div>
    <button type="submit">Send Message</button>
</form>
```

Les utilisateurs peuvent ainsi voir les messages d'erreur pertinents à côté des champs qu'ils doivent corriger.

9.9.3 Création de formulaires à partir de modèles à l'aide de ModelForm

Dans de nombreux cas, les formulaires sont liés à des modèles. Django facilite la création de formulaires directement à partir de modèles en utilisant ModelForm. Cela permet de gagner du temps en générant automatiquement des champs de formulaire basés sur les champs du modèle.

Création d'un ModelForm

Supposons que vous ayez un modèle Post dans votre fichier models.py :

```
# models.py
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

Vous pouvez créer un formulaire pour ce modèle comme suit :

```
# forms.py
from django import forms
from .models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content']
```

Utilisation du ModelForm dans une vue

```
# views.py
from django.shortcuts import render
from .forms import PostForm

def create_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()  # Enregistre les données du formulaire dans la base de
            ↪  données
            return redirect('post_list')  # Redirige vers une liste de posts
    else:
        form = PostForm()
    return render(request, 'create_post.html', {'form': form})
```

En utilisant ModelForm, Django génère automatiquement des champs de formulaire basés sur les champs du modèle Post et gère l'enregistrement du formulaire dans la base de données.

9.10 Déployer une application Django

Une fois que votre application est entièrement développée et testée sur votre machine locale, l'étape finale consiste à la déployer sur un serveur réel afin que les utilisateurs puissent y accéder. Le déploiement d'un projet Django nécessite un examen attentif des meilleures pratiques pour

garantir la sécurité, l'évolutivité et les performances. Dans cette section, nous aborderons les étapes essentielles du déploiement de votre application sur un **VPS (Serveur Privé Virtuel)** ou sur une machine locale que tout développeur Django devrait connaître.

9.10.1 Meilleures pratiques pour le déploiement d'un projet Django

Avant de plonger dans les étapes techniques, il est important de suivre quelques bonnes pratiques lors de la préparation de votre projet Django pour le déploiement :

1. **Utilisez des variables d'environnement :** Stockez les informations sensibles telles que les clés secrètes, les identifiants de base de données et les clés d'API dans des variables d'environnement au lieu de les coder en dur dans vos paramètres.
2. **Désactiver le mode débogage :** Assurez-vous de définir DEBUG = False dans votre fichier settings.py avant de déployer votre projet en production. Le mode débogage peut exposer des informations sensibles aux attaquants.
3. Définissez les hôtes autorisés : Dans votre fichier settings.py, définissez le paramètre ALLOWED_HOSTS pour inclure le(s) nom(s) de domaine où votre application sera hébergée. Par exemple :

   ```
   ALLOWED_HOSTS = ['votredomaine.com', 'www.votredomaine.com']
   ```

4. Sécurisez l'application : Envisagez d'utiliser HTTPS pour sécuriser les communications entre le serveur et les utilisateurs, et utilisez les paramètres de sécurité intégrés de Django tels que SECURE_SSL_REDIRECT, CSRF_COOKIE_SECURE et SESSION_COOKIE_SECURE.
5. **Traitement des fichiers statiques :** Assurez-vous que les fichiers statiques tels que CSS et JavaScript sont correctement collectés et servis. Django fournit la commande collectstatic pour gérer cela.
6. **Configuration de la base de données :** Assurez-vous que la base de données est correctement configurée et évitez d'utiliser SQLite pour la production. Utilisez une base de données plus robuste comme PostgreSQL ou MySQL.

9.10.2 Déployer Django sur un VPS

Maintenant que vous avez appris les meilleures pratiques pour préparer un projet Django, il est temps de le déployer sur un **serveur privé virtuel (VPS)**. Dans cette section, nous vous guiderons à travers les étapes pour mettre en place votre serveur, configurer les services nécessaires et rendre votre application Django vivante et accessible aux utilisateurs.

Création d'un fichier requirements.txt en Python

Un fichier requirements.txt liste toutes les dépendances Python dont votre projet Django a besoin pour fonctionner. Ce fichier vous permet d'installer tous les paquets nécessaires en une seule commande.

Assurez-vous d'être dans le répertoire principal de votre projet, où vous avez le fichier manage.py, et assurez-vous que votre environnement virtuel est actif.

Ensuite, générez le fichier requirements.txt avec la commande suivante :

```
pip3 freeze > requirements.txt
```

Cette commande créera un fichier requirements.txt dans votre répertoire actuel avec tous les paquets installés et leurs versions que vous avez dans votre environnement virtuel actuel.

Ensuite, ouvrez requirements.txt et assurez-vous que toutes les dépendances nécessaires à l'exécution de votre projet y figurent.

Téléchargement du projet vers un dépôt distant (GitHub)

Avant de déployer votre projet sur un **VPS**, il est conseillé de télécharger votre code sur un dépôt distant, tel que GitHub, afin de contrôler les versions et de faciliter la collaboration. De plus, cela simplifie le processus de déploiement lui-même.

1. **Installez Git CLI et initialisez un dépôt local**. Assurez-vous d'être dans le dossier principal de votre projet, où vous avez le fichier manage.py, et exécutez ces commandes :

   ```
   sudo apt install git-all
   ```

   ```
   git init
   ```

2. **Créez un fichier .gitignore et ajoutez-y du contenu**. Le fichier .gitignore indique à Git les fichiers à ignorer lors de l'envoi de code vers votre dépôt distant. Cela permet d'ignorer les fichiers contenant des informations sensibles ainsi que les fichiers locaux non pertinents tels que les journaux. Voici un exemple de fichier .gitignore pour un projet Django :

   ```
   venv/
   __pycache__/
   db.sqlite3
   *.pyc
   *.log
   media/
   ```

3. Assainissez les informations sensibles dans votre fichier settings.py. Assurez-vous d'assainir les informations sensibles dans votre fichier settings.py, telles que les mots de passe, les noms d'utilisateur, etc. Une fois le projet déployé, vous pouvez ajouter les informations d'identification à votre **VPS**.
4. **Ajoutez et engagez vos fichiers**.

   ```
   git add .
   ```

```
git commit -m "Initial commit"
```

5. S'inscrire sur GitHub (si vous n'avez pas encore de compte) et créer un dépôt distant. Allez sur GitHub et inscrivez-vous ou connectez-vous à votre compte. Créez un nouveau dépôt, définissez-le comme privé et copiez son lien distant.
6. **Configurez l'origine et la branche Git distante**.

```
git remote add origin https://github.com/url-to-repository-you-created
```

```
git branch -M main
```

7. **Vérifiez l'état de votre dépôt local et transférez des fichiers**. Avant de transférer des fichiers, vous devez exécuter la commande status et vous assurer qu'aucun fichier que vous ne souhaitez pas transférer n'a été ajouté :

```
git status
```

Confirmez que seuls les fichiers que vous souhaitez télécharger sont ajoutés et transférez-les dans le référentiel distant :

```
git push -u origin main
```

8. Accédez à votre dépôt distant sur GitHub et confirmez les fichiers téléchargés.

Préparez votre VPS et récupérez le code du projet

Ensuite, vous devez préparer votre VPS pour le déploiement et y télécharger votre code :

1. SSH dans votre VPS

```
ssh root@votre_ip_vps
```

2. Installer Nginx et Gunicorn

```
sudo apt install nginx
```

```
sudo pip3 install gunicorn
```

Nous utiliserons Nginx comme serveur web et Gunicorn comme serveur d'application.

3. **Récupérer le code de GitHub**

```
git clone https://github.com/your-repository/project-name.git
```

```
cd project-name
```

4. **Démarrer un environnement virtuel et installer les dépendances**

```
python3 -m venv env_namesource env_name/bin/activate
```

```
pip3 install -r requirements.txt
```

5. **Remplir les données assainies dans votre fichier settings.py**
6. **Exécuter les migrations et collecter les fichiers statiques**

```
python manage.py migrate
```

```
python manage.py collectstatic
```

7. **Configurer Gunicorn comme serveur d'application. Créez un fichier de service :**

```
sudo nano /etc/systemd/system/gunicorn.service
```

Ajoutez la configuration suivante :

```
Description=gunicorn daemon
After=network.target

[Service]
User=your-username
Group=www-data
WorkingDirectory=/your/project/directory
ExecStart=/your/project/directory/env_name/bin/gunicorn --access-logfile -
↪  --workers 3 --bind unix:/your/project/directory.sock
↪  myproject.wsgi:application

[Install]
WantedBy=multi-user.target
```

Veillez à saisir votre nom d'utilisateur factuel et le chemin du répertoire du projet au lieu de your-username et /your/project/directory.

8. **Démarrer et activer Gunicorn**

```
sudo systemctl start gunicorn
```

```
sudo systemctl enable gunicorn
```

9. **Configurer Nginx comme serveur web. Créez un fichier de configuration :**

```
sudo nano /etc/nginx/sites-available/myproject
```

Ajoutez la configuration suivante :

```
server {
    listen 80;
    server_name yourdomain.com www.yourdomain.com;

    location / {
        proxy_pass http://unix:/your/project/directory/project-name.sock;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /static/ {
        alias /your/project/directory/static/;
    }

    location /media/ {
        alias /your/project/directory/media/;
    }
}
```

Veillez à saisir le chemin de votre répertoire factuel au lieu de /your/project/directory.

10. **Activer le site et redémarrer Nginx**

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

```
sudo systemctl restart nginx
```

11. Sécuriser le serveur avec SSL. Nous allons utiliser Certbot pour appliquer un certificat SSL et rediriger le trafic de HTTP vers HTTPS :

```
sudo apt install certbot python3-certbot-nginxsudo certbot --nginx -d
↪  yourdomain.com -d www.yourdomain.com
```

En suivant ces étapes, vous avez déployé avec succès votre application Django sur un VPS en utilisant les meilleures pratiques, en assurant à la fois la sécurité et la performance.

Avec Nginx et Gunicorn qui s'occupent du web et de l'application, votre application est maintenant prête à gérer du trafic en direct. Visitez votre domaine – si vous avez tout configuré correctement, vous devriez maintenant voir votre application web en direct.

9.11 Conclusion

Dans ce chapitre, nous avons couvert les étapes essentielles pour démarrer avec Django, de la mise en place de votre environnement de développement au déploiement de votre projet sur un serveur virtuel. Vous avez appris à créer un projet, à gérer des modèles et à stocker des données, à construire des vues et des templates, à travailler avec des URL, à gérer des fichiers statiques, à mettre en œuvre l'authentification des utilisateurs et à déployer votre application sur un serveur live.

Avec ces compétences de base, vous êtes maintenant équipé pour commencer à construire vos propres projets Django. Qu'il s'agisse d'un simple blog ou d'une application web plus complexe, Django vous fournit les outils nécessaires pour donner vie à vos idées. Plongez, expérimentez et voyez ce que vous pouvez créer avec ce puissant framework web.

9.12 TP : Développer une application de collecte de données avec Django

cette mini application a été développée pour collecter les informations sur les usagers entrés ou sortis du territoire via les aéroports internationaux de Nsimalen et de Douala.

Elle est dotée de 3 modules :

- Le module de gestion des voyageurs (Enregistrer un nouveau voyageur, voir la liste des voyageurs, modifier les informations sur un voyageurs, supprimer un voyageur) ;
- Le module de gestion des utilisateurs de l'application (Création, liste des users, mise à jour du password d'un user, mise à jour du profile d'un user, suppression du compte d'un user) ;
- Le module statistique de l'application.

Vous avez la possibilité d'importer localement cette application hébergée sur la plateforme GitHub à l'adresse github.com/tchos/voyageurs.git.

CI-dessous quelques captures d'écran de la solution que nous allons développer à from scratch.

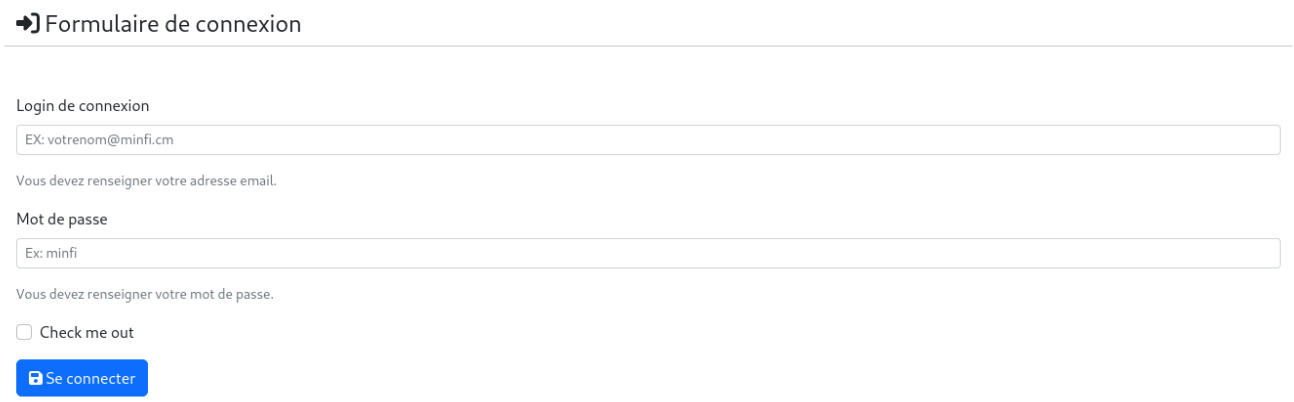


FIGURE 9.9 – Page de connexion à l'application Voyageurs

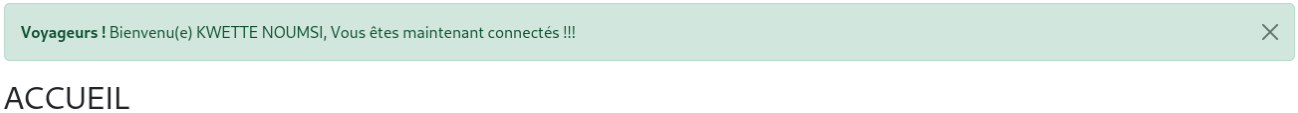


FIGURE 9.10 – Connexion effectuée avec succès

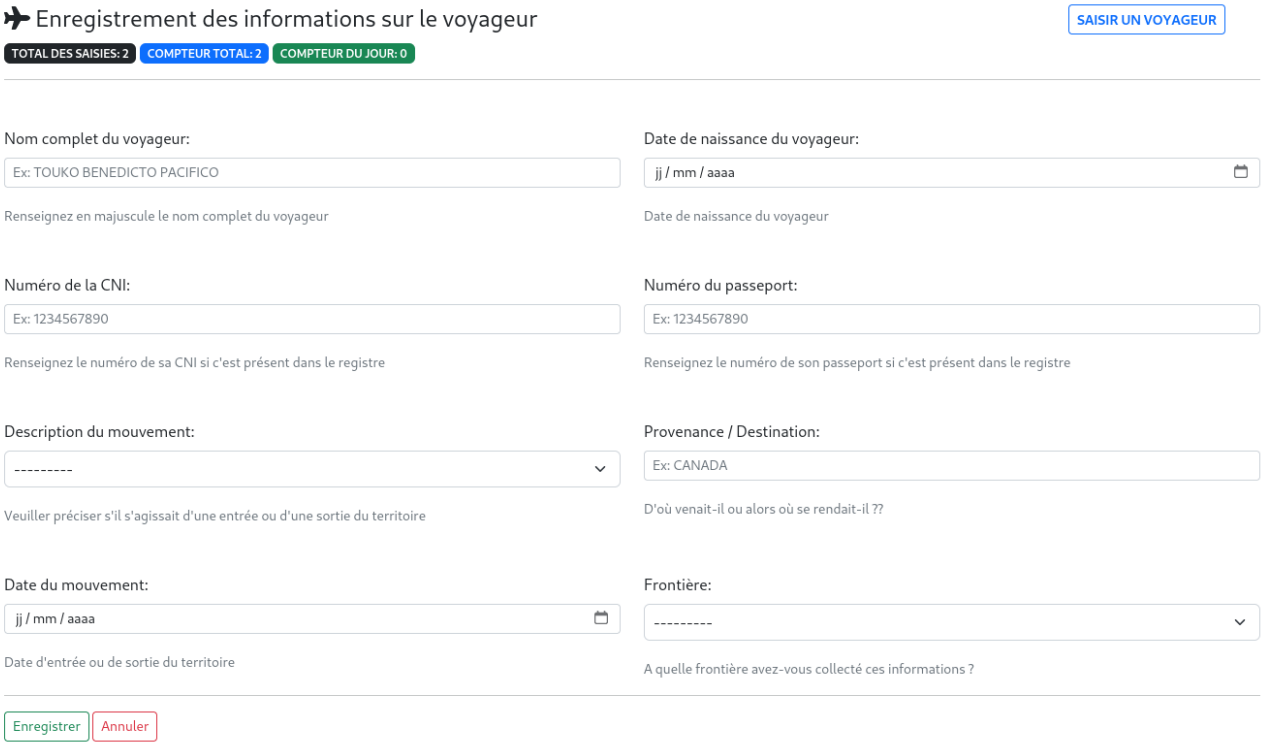


FIGURE 9.11 – Page d'enregistrement des informations sur un voyageur

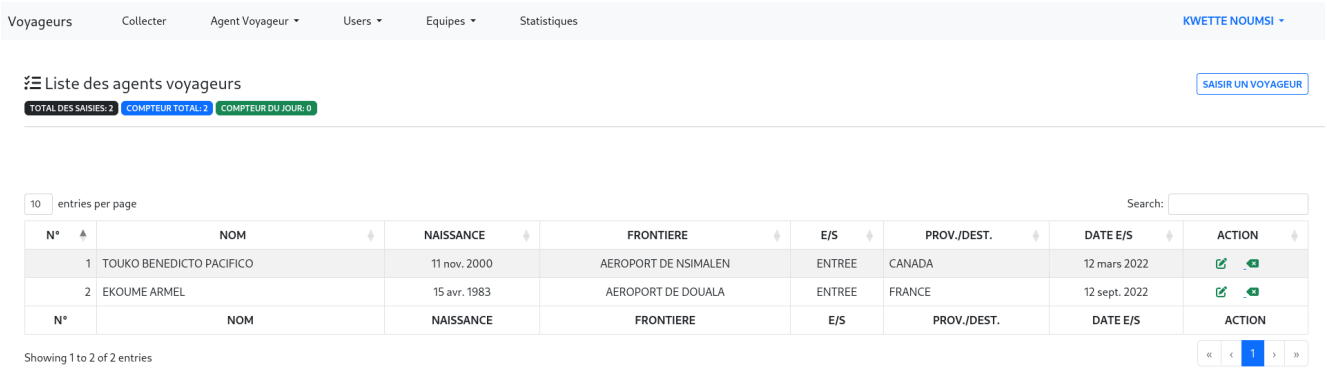


FIGURE 9.12 – Page exposant la liste des voyageurs

Voyageurs Collecter Agent Voyageur Users Equipes Statistiques KWETTE NOUMSI

Inscription d'un utilisateur

AJOUTER UNE USER

TOTAL DES SAISIES: 2 COMPTEUR TOTAL: 0 COMPTEUR DU JOUR: 0

Nom complet de l'utilisateur : Ex: ELOHIM MELCHISEDEK

Equipe de mission : ---------

Adresse email : Ex: kwenol@minfi.cm

Telephone : Ex: 620202020

Mot de passe : minfi

Confirm password minfi

Enregistrer Annuler

FIGURE 9.13 – Création d'un compte utilisateur pour l'application

Voyageurs Collecter Agent Voyageur Users Equipes Statistiques KWETTE NOUMSI

Liste des utilisateurs

AJOUTER UNE USER

TOTAL DES SAISIES: 2 COMPTEUR TOTAL: 2 COMPTEUR DU JOUR: 0

10 entries per page Search:

N°	NOM	EMAIL	DATE CREATION	LAST LOGIN	ACTION
1	KWETTE NOUMSI	kwenol@yahoo.fr	10 juin 2024	8 jan. 2026	
N°	NOM	EMAIL	DATE CREATION	LAST LOGIN	ACTION

Showing 1 to 1 of 1 entry « ‹ 1 › »

FIGURE 9.14 – Liste des users de l'application

Voyageurs Collecter Agent Voyageur Users Equipes Statistiques KWETTE NOUMSI

Liste des équipes

AJOUTER UNE EQUIPE

TOTAL DES SAISIES: 2 COMPTEUR TOTAL: 2 COMPTEUR DU JOUR: 0

10 entries per page Search:

N°	LIBELLE	CHEF D'EQUIPE	ACTION
1	AEROPORT DE DOUALA	SDG	
2	AEROPORT DE GAROUA	SDP	
3	CENTRALISATION	KWETTE NOUMSI	
4	FRONTIERE DE KYE OSSI	CES	
5	FRONTIERE NSIMALEN	CCI	
N°	LIBELLE	CHEF D'EQUIPE	ACTION

Showing 1 to 5 of 5 entries « ‹ 1 › »

FIGURE 9.15 – Liste des équipes de collecte de données

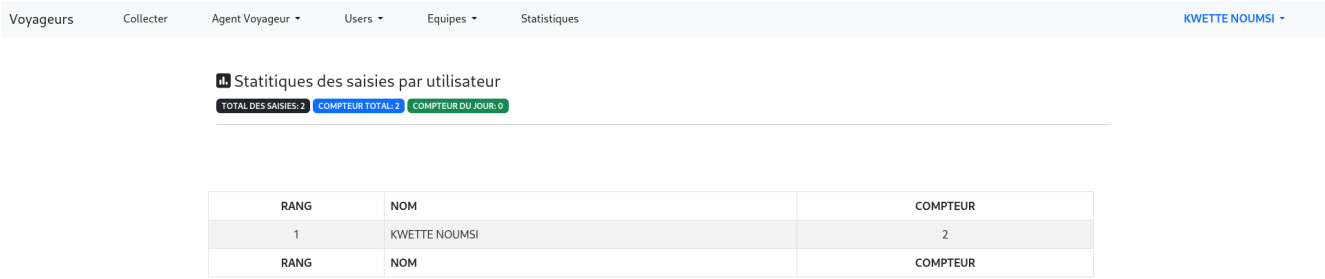


FIGURE 9.16 – Statistiques des saisies par utilisateur

Place au travail, nous allons étape après étape développer cette application en utilisant le framework Django pour développer le backend de l'application, le framework css Bootstrap et le gestionnaire des templates Jinja pour la conception des interfaces web de l'application et le SGBD PostgreSQL pour la gestion de la base de données de l'application.

9.13 Prérequis

Assurez-vous d'avoir déjà installé les outils suivants :

- Python
- Django
- PostgreSQL
- PyCharm

9.14 Etape 1 : Création du projet et de la base de données

Avant de créer notre projet *voyageur*, veuillez activer l'environnement virtuel installé dans le cadre du cours. A titre de rappel voici comment l'activer :

```
source env_name/bin/activate
```

Ensuite il faut se connecter à PostgreSQL :

```
psql -U postgres
```

```
CREATE DATABASE voyageurs;
```

On crée l'utilisateur *lolo* avec pour password *lolo* qui sera renseigné dans le fichier de configuration *settings.py* car il n'est pas recommandé d'utiliser le compte d'administration de PostgreSQL qu'est *postgres*.

Création du user lolo :

```
CREATE USER lolo WITH PASSWORD 'lolo';
```

Faire du user lolo le propriétaire de la bd voyageur dans PostgreSQL

```
ALTER DATABASE voyageurs OWNER TO lolo;
```

Attribuer les privilèges d'adminsitration de la bd voyageurs au user lolo

```
GRANT ALL PRIVILEGES ON DATABASE voyageurs TO lolo;
```

On se connecte à la BD *voyageurs* :

```
 voyageurs;
```

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO lolo;
```

```
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO lolo;
```

```
GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO lolo;
```

Maintenant que tout est OK pour la BD, on reviens dans le terminal pour se déplacer dans le répertoire qui va accueillir notre projet tout en s'assurant d'avoir activé l'environnement virtuel et l'on crée le projet *voyageurs*.

Pour exemple, tous mes projets Django sont développés à partir d'un répertoire nommé *django*

```
tchos@localhost:~/Documents/projets/python/django$
```

FIGURE 9.17 – Exemple d'organisation locale pour stokage des projets django

On active l'environnement virtuel :

```
tchos@localhost:~/Documents/projets/python/django$ source .env314/bin/activate
(.env314) tchos@localhost:~/Documents/projets/python/django$
```

FIGURE 9.18 – Activation de l'environnement virtuel

Comme vous pouvez le constatez, sur ma machine l'environnement virtuel a été mis en place sous le nom de *.env314*. Vous devez le remplacer le nom de l'environnement mis en place sur votre machine.

Maintenant que l'environnement virtuel est activé et que la BD a été créée, nous pouvons créer notre projet :

```
tchos@localhost:~/Documents/projets/python/django$ source .env314/bin/activate
(.env314) tchos@localhost:~/Documents/projets/python/django$
(.env314) tchos@localhost:~/Documents/projets/python/django$ django-admin startproject voyageurs
```

FIGURE 9.19 – Création du projet voyageurs

La première étape dans le développement d'un projet Django est achevée. La suite consistera à configurer notre projet via le fichier settings.py dudit projet.

9.15 Etape 2 : Configurations du projet

Après avoir créé un projet Django, quelques configurations de base devront être effectuées avant d'entamer son développement. Il s'agit notamment de :

- Configurer la connexion à la base données selon le SGBD choisi ;
- Préciser dans quel dossier seront stockés les pages web du projet (dossier *templates* ;
- Préciser dans quel dossier seront stockés les fichiers statiques (css, js, images) du projet (dossier *static* ;
- Préciser dans quel dossier seront stockés les fichiers média (pdfs, audios, vidéos, etc.) du projet (dossier *media* ;
- Configurer les messages flash qui seront utilisés pour notifier les utilisateurs finaux relativement aux actions qu'ils effectue au sein de l'application.
- Configuration de la connexion à l'adresse email de l'administrateur de l'application pour notification.

cependant, pour des raisons de sécurité, vous ne devez pas exposer tous ces paramètres au sein du fichier settings.py mais plutôt dans un fichier caché qui ne sera jamais accessible au public. Sur ma machine je stoke tous les paramètres de sécurité dans un fichier créé à la racine du dossier projet et nommé (Par convention recommandée par la communauté des développeurs Django) *.env*.

Les configurations de base à effectuer pour le bon fonctionnement de notre projet commenceront donc par la création du fichier *.env* à la racine de votre dossier projet. Utilisez le terminal et déplacer vous dans le dossier *voyageurs* généré après la commande *django-admin startproject voyageurs* qui a servi à la création de votre projet à l'étape 1.

```
(.env314) tchos@localhost:~/Documents/projets/python/django$
(.env314) tchos@localhost:~/Documents/projets/python/django$ cd voyageurs
(.env314) tchos@localhost:~/Documents/projets/python/django/voyageurs$
```

Une fois dans le dossier de votre projet, veuillez créer le fichier *.env* :

```
touch .env
```

Ouvrez votre dossier dans l'EDI PyCharm, ensuite ouvrez le fichier et mettez-y le contenu suivant adapté à vos paramètres :

```
# Prendre la valeur de la variable SECRET_KEY dans le fichier settings.py
SECRET_KEY=django-insecure-mf*5=g12k#w$=^ipim-duij*v4cb8e4b%(vzi45m6_#&*1%l#a
DEBUG=True

# Paramtres de la BD
DB_NAME=voyageurs
DB_USER=lolo
DB_PASSWORD=lolo
DB_HOST=localhost
PORT=5432

# paramètres de config de l'email
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_HOST_USER=tchos2012@gmail.com
EMAIL_HOST_PASSWORD=mfjmqdpnavjftsmu
EMAIL_USE_TLS=True
EMAIL_USE_SSL=False
```

Ensuite et toujours à la racine de votre dossier projet, exécutez la commande suivante :

```
pip install python-decouple
```

Le contenu de mon fichier settings.py présenté ci-dessous, récapitule toutes les configurations évoquées à l'introduction de cette section. Veuillez la personnaliser à votre projet.

```
"""
Django settings for voyageurs project.

Generated by 'django-admin startproject' using Django 6.0.1.

For more information on this file, see
https://docs.djangoproject.com/en/6.0/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/6.0/ref/settings/
"""

from pathlib import Path
import os
from decouple import config

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/6.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = config('SECRET_KEY')

# SECURITY WARNING: don't run with debug turned on in production!
```

```
DEBUG = config('DEBUG', cast=bool)

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

ROOT_URLCONF = "voyageurs.urls"

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR/'templates'],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]

WSGI_APPLICATION = "voyageurs.wsgi.application"


# Database
# https://docs.djangoproject.com/en/4.2/ref/settings/#databases

"""
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
```

```
    }
}
"""

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),
        'USER': config('DB_USER'),
        'PASSWORD': config('DB_PASSWORD'),
        'HOST': config('DB_HOST'),
        'PORT': config('PORT')
    }
}



# Password validation
# https://docs.djangoproject.com/en/4.2/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME": "django.contrib.auth.password_validation
                .UserAttributeSimilarityValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME":
        ↪  "django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]


# Internationalization
# https://docs.djangoproject.com/en/4.2/topics/i18n/

LANGUAGE_CODE = "fr-FR"

TIME_ZONE = "UTC"

USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.2/howto/static-files/

STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
```

```
# Configuration du chemin de dossier pour les medias
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'

# Default primary key field type
# https://docs.djangoproject.com/en/4.2/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"

# Pour les messages flash
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.ERROR: "",
}

# EMAIL configuration
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = config('EMAIL_HOST')
EMAIL_PORT = config('EMAIL_PORT', cast=int)
EMAIL_HOST_USER = config('EMAIL_HOST_USER')
EMAIL_HOST_PASSWORD = config('EMAIL_HOST_PASSWORD')
EMAIL_USE_TLS = config('EMAIL_USE_TLS', cast=bool)
EMAIL_USE_SSL = config('EMAIL_USE_SSL', cast=bool)
DEFAULT_FROM_EMAIL = 'foodOnline Marketplace <tchos2012@gmail.com>'

# Pour demander à django d'utiliser notre modèle personnalisé du User
AUTH_USER_MODEL = "accounts.CustomUser"
AUTH_USER_EMAIL_UNIQUE = True

# Pour les messages flash
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.DEBUG: 'alert-secondary',
    messages.INFO: 'alert-info',
    messages.SUCCESS: 'alert-success',
    messages.WARNING: 'alert-warning',
    messages.ERROR: 'alert-danger',
}
```

Pour plus d'informations concernant la configuration du fichier settings.py, veuillez consulter de la documentation de Django sur le sujet à l'adresse https://docs.djangoproject.com/en/6.0/topics/settings/.

Si tout s'est bien déroulé, démarrez votre projet pour voir la page d'accueil s'afficher dans le navigateur en exécutant la commande suivante :

```
python manage.py runserver
```

Maintenant, ouvrez votre navigateur et allez sur http://127.0.0.1:8000/ pour voir votre page d'accueil s'afficher.

9.16 Etape 3 : Développement des modules du projet

Nous allons véritablement entamé le développement de notre. Il est doté de 2 principaux modules que sont le module des gestion des comptes d'utilisateurs et le module de gestion des voyageurs. Dans le jargon de Django les modules sont assimilés au concept d'applications. Pour faire simple voyez les applications Django comme les différents modules (ensemble de fonctionnalités liées à une ou plusieurs entités communes) de votre projet Django.

IL est fortement recommandé de commencer par la mise en place du module des gestion des utilisateurs avant de développer les autres modules sachant que chaque module aura ses fichiers models.py, views.py, urls.py, forms.py (éventuellement) et ses templates.

9.16.1 Module de gestion des compte utilisateurs (accounts)

Pour créer l'application *accounts* qui sera en charge de gérer les utilisateurs dans notre projet Django, veuillez exécuter la commande suivante :

```
python manage.py startapp accounts
```

Après avoir créé une application, vous devez toujours aller l'ajouter dans le fichier settings.py parmi la liste des applications du projets (variable *INSTALLED_APPS*

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'accounts', # Ajout de la nouvelle application que vous avez créée.
]
```

Par ailleurs vous verrez le dossier *accounts* apparaître à la racine de votre dossier projet.

Mise en place du modèle des users

A titre de rappel, Django est un framework qui implémente le design pattern **MVC (Modèle-Vue-Contrôleur)** sous la forme **MVT (Modèle-View-Template, voir 9.1.5)** dont le M représente la couche Modèle qui va générer les différentes tables de la BD du projet.

Voici le contenu du fichier models.py de l'application accounts :

```
from django.utils import timezone
from django.db import models
from django.contrib.auth.base_user import AbstractBaseUser, BaseUserManager
from django.db.models.fields.related import ForeignKey, OneToOneField
from django.db.models.signals import post_save
from voyageurs import settings
```

```python
# Create your models here.
class UserManager(BaseUserManager):
    def create_user(self, email, nom, telephone, equipe, password=None):
        if not email:
            raise ValueError("L'email de l'utilisateur est obligatoire")
        if not nom:
            raise ValueError("L'utilisateur doit avoir un nom")
        if not telephone:
            raise ValueError("On doit pouvoir contacter un utilisateur")

        user = self.model(
            email=self.normalize_email(email),
            nom=nom,
            telephone=telephone,
            equipe=equipe,
        )
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, nom, telephone, password=None):
        if not email:
            raise ValueError("L'email de l'utilisateur est obligatoire")
        if not nom:
            raise ValueError("L'utilisateur doit avoir un nom")
        if not telephone:
            raise ValueError("On doit pouvoir contacter un utilisateur")

        user = self.model(
            email=self.normalize_email(email),
            nom=nom,
            telephone=telephone,
        )

        user.set_password(password)
        user.is_admin = True
        user.is_staff = True
        user.is_active = True
        user.is_superadmin = True
        user.save(using=self._db)
        return user

class Team(models.Model):
    """Model pour les équipes de mission"""
    libelle = models.CharField(max_length=64, blank=False, error_messages="Le nom
    ↪ de l'équipe est obligatoire", unique=True)
    chef = models.CharField(max_length=64, blank=False, error_messages="Une équipe
    ↪ doit toujours avoir un chef d'équipe")

    # Tel que ca va apparaitre dans le panneau d'administration de Django
    class Meta:
        verbose_name = "Equipes de mission"
        ordering = ["libelle"]
```

```
    """ Dans le panneau d'admin des users, ce sont les nom
        et les email des users qui seront affichés. """
    def __str__(self):
        return f"{self.libelle}"


class CustomUser(AbstractBaseUser):
    """Model pour les utilisateurs du système"""
    email = models.EmailField(unique=True, max_length=255, blank=False)
    nom = models.CharField(max_length=64, blank=False,
            error_messages="Le nom est obligatoire", verbose_name='Nom complet')
    equipe = models.ForeignKey(Team, on_delete=models.SET_NULL, null=True)
    telephone = models.CharField(max_length=32, blank=False,
        error_messages="Telephone obligatoire", null=False)

    date_joined = models.DateTimeField(auto_now_add=True,
            verbose_name='date joined', null=True)
    last_login = models.DateTimeField(auto_now_add=True,
            verbose_name='last login')
    created_date = models.DateTimeField(auto_now_add=True)
    modified_date = models.DateTimeField(auto_now=True)

    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False)
    is_superadmin = models.BooleanField(default=False)

    # indique le nom du champ devant être utilisé comme identifiant de connexion.
    USERNAME_FIELD = "email"
    """ liste des champs à spécifier obligatoirement lors de l'utilisation
        de la commande  python manage.py createsuperuser """
    REQUIRED_FIELDS = ["nom","telephone"]

    objects = UserManager()

    def has_perm(self, perm, obj=None):
        return True

    def has_module_perms(self, app_label):
        return True

    # Tel que ca va apparaitre dans le panneau d'administration de Django
    class Meta:
        verbose_name = "Utilisateurs"
        ordering = ["-date_joined"]

    """ Dans le panneau d'admin des users, ce sont les nom
        et les email des users qui seront affichés. """
    def __str__(self):
        return f"{self.nom} - {self.email}"

class UserProfile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
            on_delete=models.CASCADE, blank=True, null=True)
```

```
def post_save_receiver(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)
post_save.connect(post_save_receiver, sender=settings.AUTH_USER_MODEL)
```

Pour en savoir plus sur les modèles de Django, veuillez vous rendre à l'adresse https://docs.djangoproject.com/fr/6.0/topics/db/models/.

Après avoir mis en place le modèle du module *accounts*, il faut appliquer les migrations qui auront pour effet de générer ou de mettre à jour les tables de la BD. Les migrations s'effectuent en exécutant successivement les 2 commandes ci-après :

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Connectez-vous à votre BD sur PostgresSQL pourconstater que de nouvelles tables se sont ajoutées.

Chapitre 10

Module 10 : Python pour la Data Science

Objectifs :

— Maîtriser l'écosystème Data Science avec Python
— Effectuer des analyses et des prédictions sur les données avec python.

Contenu :

- Calcul scientifique avec NumPy.
- Manipulation de données avec Pandas.
- Nettoyage et transformation des données.
- Visualisation avec Matplotlib/Seaborn.
- Analyse exploratoire des données.
- Introduction à Scikit-learn (Machine Learning).

Outils de travail :

- Jupyter Notebook.
- Jupyter Lab.
- Data Spell.
- Google Colab.
- Kaggle.

10.1 Calcul scientifique avec NumPy

NumPy est une puissante bibliothèque d'algèbres linéaires pour Python. Ce qui le rend si important, c'est que presque toutes les bibliothèques de l'écosystème PyData (pandas, scipy, scikit-learn, etc.) s'appuient sur NumPy comme l'un de leurs principaux éléments de construction. De plus, nous l'utiliserons pour générer des données pour nos exemples d'analyse plus tard !

NumPy est également incroyablement rapide, car il possède des relations avec les bibliothèques du langage C. Pour plus d'informations sur les raisons pour lesquelles vous souhaitez utiliser des tableaux plutôt que des listes, consultez ce post StackOverflow.

Nous n'apprendrons que les bases de NumPy. Pour commencer, nous devons l'installer !

Note : Instructions d'installation

NumPy est déjà inclus dans votre environnement ! Nous recommandons VIVEMENT d'utiliser notre environnement comme indiqué dans la vidéo de mise en place de l'environnement. Vous pouvez continuer si vous utilisez l'environnement du cours !

Pour ceux qui n'utilisent pas l'environnement de code recommandé :

Il est fortement recommandé d'installer Python à l'aide de la distribution Anaconda pour vous assurer que toutes les dépendances sous-jacentes (telles que les bibliothèques d'algèbre linéaire) sont synchronisées avec l'utilisation d'une installation de conda. Si vous avez Anaconda, installez NumPy en accédant à votre terminal ou à l'invite de commandes (ou Anaconda Prompt) ou directement sur une cellule Jupyter Notebook (sans oublier de de redémarrer le noyau juste après) et tapez :

```
conda install numpy
```

Ou encore

```
pip install numpy
```

Si vous ne possédez pas Anaconda et que vous ne pouvez pas l'installer, veuillez vous référer à la documentation officielle de NumPy sur les différentes instructions d'installation.

10.1.1 Importation de NumPy

Une fois que vous avez installé NumPy, vous pouvez l'importer en tant que bibliothèque :

```
import numpy as np
```

NumPy dispose de nombreuses fonctions et capacités intégrées. Nous ne les couvrirons pas tous, mais nous nous concentrerons plutôt sur certains des aspects les plus importants de NumPy : les vecteurs, les tableaux, les matrices et la génération de nombres. Commençons par discuter des tableaux.

10.1.2 Tableaux NumPy

Les tableaux NumPy sont la principale façon d'utiliser NumPy tout au long du cours. Les tableaux NumPy se déclinent essentiellement en deux éléments : les vecteurs et les matrices. Les vecteurs sont des tableaux strictement unidimensionnels (1D) et les matrices sont 2D (mais il faut noter qu'une matrice peut toujours ne comporter qu'une seule ligne ou qu'une seule colonne).

10.1.3 Pourquoi utiliser un tableau NumPy ?

Pourquoi ne pas simplement utiliser une liste ?

Il existe de nombreuses raisons d'utiliser un tableau NumPy au lieu d'un objet de liste Python « standard ». Nos principales raisons sont les suivantes :

- Efficacité de la mémoire du « array NumPy » vs. « list »
- Élargissement facile aux objets N-dimensionnels
- Vitesse de calcul du tableau PumPy
- Opérations et fonctions de diffusion (broadcasting) avec NumPy
- Toutes les bibliothèques de Data Science et de Machine Learning que nous utilisons sont construites avec NumPy

Exemple simple d'un tableau NumPy

```
my_list = [1,2,3]
my_array = np.array([1,2,3])
```

```
type(my_list)
```

```
type(my_array)
```

```
my_array
```

Commençons notre introduction en explorant comment créer des tableaux NumPy.

10.1.4 Création de tableaux NumPy à partir d'objets

À partir d'une liste Python

Nous pouvons créer un tableau en convertissant directement une liste ou une liste de listes :

```
my_list = [1,2,3]
my_list
```

```
np.array(my_list)
```

```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
my_matrix
```

```
np.array(my_matrix)
```

Méthodes intégrées pour créer des tableaux

Il existe de nombreuses façons intégrées de générer des tableaux.

arange : Retourner des valeurs uniformément espacées dans un intervalle donné [doc].

```
np.arange(0,10)
```

```
np.arange(0,11,2)
```

zeros et ones : Génère des tableaux de zéros ou de uns. [doc]

```
np.zeros(3)
```

```
np.zeros((5,5))
```

```
np.ones(3)
```

```
np.ones((3,3))
```

linspace : Retourne des nombres uniformément espacés sur un intervalle spécifié. [doc]

```
np.linspace(0,10,3)
```

Notez que .linspace() inclut la valeur stop. Pour obtenir un tableau de fractions courantes, augmentez le nombre d'éléments :

```
np.linspace(0,5,21)
```

eye : Crée une matrice identité [doc]

```
np.eye(4)
```

Random

NumPy propose également de nombreux moyens de créer des tableaux de nombres aléatoires.

rand : Crée un tableau de la forme donnée et le remplit d'échantillons aléatoires provenant d'une distribution uniforme sur les éléments suivants [0, 1). [doc]

```
np.random.rand(2)
```

```
np.random.rand(5,5)
```

randn : Retourne un échantillon (ou des échantillons) de la loi « normale standard » [= 1]. Contrairement à rand qui est uniforme, les valeurs proches de zéro sont plus susceptibles d'apparaître. [doc]

```
np.random.randn(2)
```

```
np.random.randn(5,5)
```

randint : Retourne des nombres entiers aléatoires de low (inclusif) à high (exclusif). [doc]

```
np.random.randint(1,100)
```

```
np.random.randint(1,100,10)
```

seed : Peut être utilisé pour définir l'état aléatoire, de sorte que les mêmes résultats "aléatoires" puissent être reproduits. [doc]

```
np.random.seed(42)
np.random.rand(4)
```

```
np.random.seed(42)
np.random.rand(4)
```

10.1.5 Attributs et méthodes Array

Examinons quelques attributs et méthodes utiles pour un tableau.

```
arr = np.arange(25)
ranarr = np.random.randint(0,50,10)
```

```
arr
```

```
ranarr
```

Reshape

Retourne un tableau contenant les mêmes données avec une nouvelle forme (ou dimension). [doc]

```
arr.reshape(5,5)
```

max, min, argmax, argmin

Ce sont des méthodes utiles pour trouver les valeurs maximales ou minimales. Ou pour trouver leurs emplacements d'index en utilisant argmin ou argmax

```
ranarr
```

```
ranarr.max()
```

```
ranarr.argmax()
```

```
ranarr.min()
```

```
ranarr.argmin()
```

Shape

Shape est un attribut des tableaux (et non une méthode) : [doc]

```
# Vecteur
arr.shape
```

```
# Remarquez les deux sets de parenthèses
arr.reshape(1,25)
```

```
arr.reshape(1,25).shape
```

```
arr.reshape(25,1)
```

```
arr.reshape(25,1).shape
```

dtype

Vous pouvez également saisir le type de données de l'objet dans le tableau : [doc]

```
arr.dtype
```

```
arr2 = np.array([1.2, 3.4, 5.6])
arr2.dtype
```

10.2 Indexation et Selection NumPy

Dans cette section, nous allons voir comment sélectionner des éléments ou des groupes d'éléments dans un tableau

```
import numpy as np
```

```
# Création d'un tableau d'échantillons
arr = np.arange(0,11)
```

```
# Afficher
arr
```

10.2.1 Crochets d'indexation et de sélection

La façon la plus simple de choisir un ou plusieurs éléments d'un tableau ressemble beaucoup aux listes python :

```
# Obtenir une valeur à un index (indice)
arr[8]
```

```
# Obtenir des valeurs dans une plage d'index (ou fourchette)
arr[1:5]
```

```
# Obtenir des valeurs dans une fourchette d'index
arr[0:5]
```

10.2.2 Broadcasting

Les tableaux NumPy diffèrent des listes Python "standards" en raison de leur capacité de diffusion (ou broadcasting en anglais). Avec les listes, vous pouvez uniquement réaffecter des parties d'une liste avec de nouvelles parties de même taille et de même forme. Autrement dit, si vous voulez remplacer les 5 premiers éléments d'une liste par une nouvelle valeur, vous devez passer une nouvelle liste de 5 éléments. Avec les tableaux NumPy, vous pouvez diffuser une valeur unique à travers un ensemble plus large de valeurs :

```
# Réglage d'une valeur avec une plage d'index (Diffusion)
arr[0:5]=100

# Afficher
arr
```

```
# Réinitialiser le tableau, nous verrons pourquoi j'ai dû réinitialiser ci-dessous
arr = np.arange(0,11)

# Afficher
arr
```

```
# Notes importantes sur les slices
slice_of_arr = arr[0:6]

# Afficher la tranche (slice)
slice_of_arr
```

```
# Modifier la tranche (slice)
slice_of_arr[:]=99
```

```
# Afficher de nouveau la tranche (slice)
slice_of_arr
```

Notez maintenant que les changements se produisent également dans notre tableau d'origine !

```
arr
```

Les données ne sont pas copiées, c'est une vue du tableau d'origine ! Cela évite les problèmes de mémoire !

```
# Pour obtenir une copie, il faut être explicite
arr_copy = arr.copy()

arr_copy
```

10.2.3

Le format général est le suivant arr_2d[row][col] ou arr_2d[row,col]. Je recommande d'utiliser la notation avec virgule pour plus de clarté.

```
arr_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))

# Afficher
arr_2d
```

```
# Indexation par ligne
arr_2d[1]
```

```
# Le format est arr_2d[row][col] ou arr_2d[row,col]

# Obtenir la valeur d'un élément individuel
arr_2d[1][0]
```

```
# Obtenir la valeur d'un élément individuel
arr_2d[1,0]
```

```
# Découpage (slicing) d'une matrice 2D

# Forme ou shape (2,2) à partir du coin supérieur droit
arr_2d[:2,1:]
```

```
# Forme (ou shape) de la rangée du bas
arr_2d[2]
```

```
# Forme (ou shape) de la rangée du bas
arr_2d[2,:]
```

10.2.4 Plus d'aide pour l'indexation

L'indexation d'une matrice 2D peut être un peu déroutante au début, surtout lorsque vous commencez à ajouter la taille des pas. Essayez de chercher Indexing NumPy sur Google Image pour trouver des images utiles, comme celle-ci :

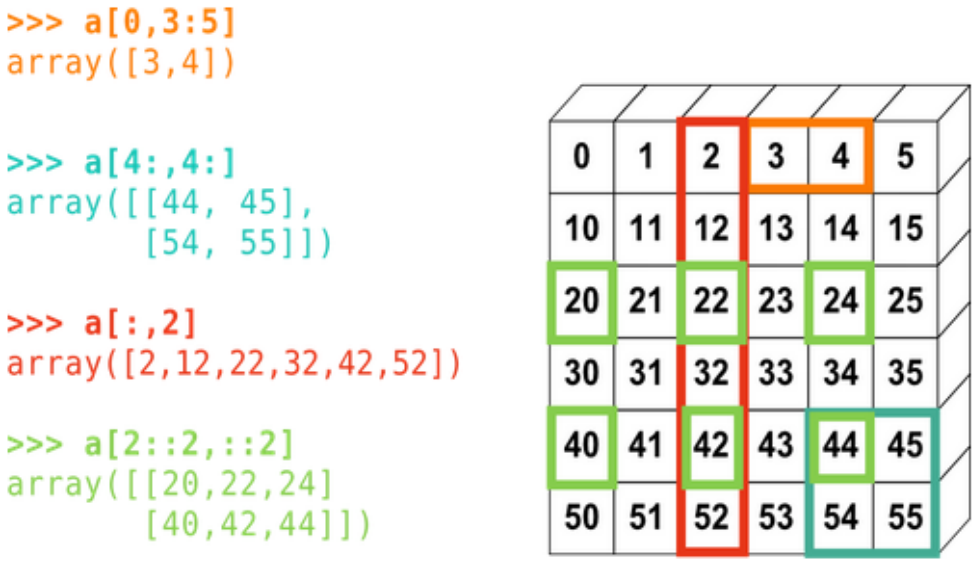


FIGURE 10.1 – Comprendre l'indexation

10.2.5 Sélection conditionnelle

Il s'agit d'un concept très fondamental qui sera directement transposé à Pandas par la suite, assurez-vous de bien comprendre cette partie !

Voyons brièvement comment utiliser les parenthèses pour une sélection basée sur des opérateurs de comparaison.

```
arr = np.arange(1,11)
arr
```

```
arr > 4
```

```
bool_arr = arr>4
```

```
bool_arr
```

```
arr[bool_arr]
```

```
arr[arr>2]
```

```
arr[arr>2]
```

```
x = 2
arr[arr>x]
```

10.3 Opérations NumPy

10.3.1 Arithmétique

Vous pouvez facilement effectuer de l'arithmétique tableau avec tableau ou arithmétique scalaire avec tableau.

Regardons quelques exemples :

```
import numpy as np
arr = np.arange(0,10)
arr
```

```
arr + arr
```

```
arr * arr
```

```
arr - arr
```

```
# Cela déclenchera un avertissement sur la division par zéro, mais pas une erreur
↪ !
# Il remplit simplement la place avec nan
arr/arr
```

```
# Également un avertissement (mais pas une erreur) - relatif à l'infini
1/arr
```

```
arr**3
```

10.3.2 Fonctions universelles de tableaux¶

NumPy est livré avec de nombreuses fonctions universelles de tableaux, ou ufuncs, qui sont essentiellement juste des opérations mathématiques qui peuvent être appliquées à travers le tableau.

Présentons-en quelques unes qui sont très courantes :

```
# Prendre les racines carrées
np.sqrt(arr)
```

```
# Calculer l'exponentielle (e^)
np.exp(arr)
```

```
# Fonctions trigonométriques comme le sinus
np.sin(arr)
```

```
# Calculer le logarithme naturel
np.log(arr)
```

10.4 Statistiques sommaires sur les tableaux

NumPy propose également des statistiques récapitulatives courantes telles que somme, moyenne et max. Vous les appellez comme des méthodes sur un tableau.

```
arr = np.arange(0,10)
arr
```

```
arr.sum()
```

```
arr.mean()
```

```
arr.max()
```

D'autres statistiques récapitulatives incluent :

```
arr.min() # retourne 0                  minimum
arr.var() # retourne 8.25               variance
arr.std() # retourne 2.8722813232690143 écart-type (standard deviation)
```

10.5 Logique d'axe

Lorsque vous travaillez avec des tableaux bidimensionnels (matrices), nous devons prendre en compte les lignes et les colonnes. Cela devient très important lorsque nous arriverons à la section sur Pandas. En termes de tableau, l'axe 0 (zéro) est l'axe vertical (lignes ou rows) et l'axe 1 est l'axe horizontal (colonnes). Ces valeurs (0,1) correspondent à l'ordre dans lequel les valeurs arr.shape sont renvoyées.

Voyons comment cela affecte nos calculs statistiques récapitulatifs ci-dessus.

```
arr_2d = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
arr_2d
```

```
arr_2d.sum(axis=0)
```

En passant axis=0, nous retournons un tableau de sommes le long de l'axe vertical, essentiellement [(1+5+9), (2+6+10), (3+7+11), (4+8+12)].

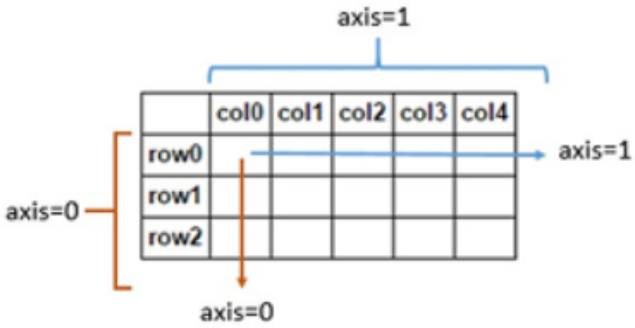


FIGURE 10.2 – Logique des axes sur un array numpy

```
arr_2d.shape
```

Cela nous indique que arr_2d a 3 lignes et 4 colonnes.

Dans arr_2d.sum(axis=0) ci-dessus, le premier élément de chaque ligne a été additionné, puis le deuxième élément, et ainsi de suite.

Alors que devrait retourner arr_2d.sum(axis=1) ?

```
# Réfléchissez au résultat avant d'exécuter la cellule ;)
arr_2d.sum(axis=1)
```

10.6 Analyse de données avec Numpy

Pour effectuer ce travail, veuillez télécharger le dataset World Alcohol.

```
import numpy as np

world_alcohol = np.genfromtxt("world_alcohol.csv", delimiter=",", dtype="U75",
↪  skip_header=1)
world_alcohol
```

10.6.1 Effectuer des comparaisons

```
vector = np.array([5, 10, 11, 10, 23, 7])
```

```
vector == 10
```

Mission

- Extraite la 3e colonne de *world_alcohol* et comparer la pays "Canada". Assigner le résultat à la variable *countries_canada*.
- Extraire la première colonne de *world_alcohol* et comparer la à la chaine de caractères "1984". Assigner le résultat à la variable *years_1984*.

```
countries_canada = (world_alcohol[:,2] == "Canada")
print(countries_canada)

# Affiche toutes les lignes contenant "Canada"
print(world_alcohol[countries_canada])
```

```
years_1984 = (world_alcohol[:,0] == "1984")
print(years_1984)

# Affiche toutes les lignes contenant "1984"
print(world_alcohol[years_1984])
```

Sélection des éléments

- Sélectionner seulement les lignes où le pays est "Algeria".
- Sélectionner seulement les lignes où l'année est "1984".

```
print(world_alcohol)
```

```
# Sélection des lignes où le pays est "Algeria"
countrie_is_algeria = (world_alcohol[:,2] == "Algeria")
print(world_alcohol[countrie_is_algeria])

# On peut aussi faire ==> print(world_alcohol[world_alcohol[:,2] == "Algeria"])
```

Sélection avec les conditions multiples

- Sélectionner les lignes où le pays est "Algeria" avec pour année "1986".

```
#is_algeria_and_1986 = (world_alcohol[world_alcohol[:,2] == "Algeria"]) &
↪  (world_alcohol[world_alcohol[:,0] == "1986"])
is_algeria_and_1986 = (world_alcohol[:,2] == "Algeria") & (world_alcohol[:,0] ==
↪  "1986")

print(world_alcohol[is_algeria_and_1986])
# On pouvait aussi faire ==> print(world_alcohol[(world_alcohol[:,2] == "Algeria")
↪  & (world_alcohol[:,0] == "1986")])
```

```
# Autres méthodes de sélection:
rows_with_algeria_and_1986 = world_alcohol[is_algeria_and_1986,]
print(rows_with_algeria_and_1986)
```

Modification de certaines valeur dans un tableau numpy

- Changer l'année "1986" en "2018" et changer la boisson "Wine" en "Beer".

```
# On duplique world_alcohol: 2 méthodes
world_alcohol_2 = world_alcohol[:] # Méthode 1
world_alcohol_2 = world_alcohol.copy() # Méthode 2
world_alcohol_2
```

```
# Méthode 1
year_1986 = world_alcohol_2[:,0] == "1986"
boisson_wine = world_alcohol_2[:,3] == "Wine"

world_alcohol_2[year_1986,0] = "2018"
world_alcohol_2[boisson_wine,3] = "Beer"

print("Résultat avec la méthode 1:\n", world_alcohol_2)

# Méthode 2
```

```
world_alcohol_2 = world_alcohol.copy()
world_alcohol_2[:,0][world_alcohol_2[:,0] == "1986"] = "2018"
world_alcohol_2[:,3][world_alcohol_2[:,3] == "Wine"] = "Beer"

print("\n Résultat avec la méthode 2:\n", world_alcohol_2)
```

Remplacer les chaines de caractères vides par '0'

```
world_alcohol_2[:,4][world_alcohol_2[:,4] == ""] = "0"
```

Convertir les string en float : cas de la 5e colonne

```
# Pour que la conversion marche, il faut remplacer le vide par 0.
alcohol_consumption = world_alcohol_2[:,4]
alcohol_consumption = alcohol_consumption.astype(float)
print(alcohol_consumption)
```

Opérations mathématiques avec Numpy. Ex avec les fonctions sum(), mean(), max()

```
matrix = np.array([[10,5,15],[20,9,11],[14,13,3]])
print(matrix)
```

```
# Opération sur les colonnes: axis=0
print("Sommes des colonnes :", matrix.sum(axis=0))
print("Moyenne des colonnes :", matrix.mean(axis=0))
print("Valeure maximale des colonnes :", matrix.max(axis=0))
```

```
# Opération sur les lignes: axis=1
print("Sommes des colonnes :", matrix.sum(axis=1))
print("Moyenne des colonnes :", matrix.mean(axis=1))
print("Valeure maximale des colonnes :", matrix.max(axis=1))
```

Somme et Moyenne de consommation en alcool

```
print("Somme des consommations :", alcohol_consumption.sum(axis=0))
print("Moyenne des consommations :", alcohol_consumption.mean(axis=0))
```

Consommation totale d'alcool par habitant en 1986 au Canada

```
# On commence par filtrer les données sur le canada en 1986.
consumption_canada_1986 = world_alcohol[(world_alcohol[:,0] == "1986") &
↪  (world_alcohol[:,2] == "Canada")]
print(consumption_canada_1986)
```

```
# On récupère la colonne des consommations et on la convertit en float
consumption_canada_1986[:,4][consumption_canada_1986[:,4] == ""] = "0"
```

```
consumption_canada_1986_alcohol = consumption_canada_1986[:,4].astype(float)
print(consumption_canada_1986_alcohol)
```

```
total_canada_drinking = consumption_canada_1986_alcohol.sum()
print(total_canada_drinking)
```

Consommation pour chaque pays en 1989

```
# Dictionnaire qui va contenir les pays et leur consommation
totals = {}

years_1989 = world_alcohol[world_alcohol[:,0] == "1989"]

# Liste des pays
countries = world_alcohol[:,2]

print(years_1989)
```

```
# On parcours la liste des pays et pour chaque pays, on va calculer sa
↪ consommation en 1989.
for pays in countries:
    country_consumtion = years_1989[years_1989[:,2] == pays]

    # On remplace le vide par 0.
    country_consumtion[:,4][country_consumtion[:,4] == ""] = "0"

    # Conversion en float
    country_consumtion_alcohol = country_consumtion[:,4].astype(float)

    # On ajoute au dico le pays et sa conso totale
    totals[pays] = country_consumtion_alcohol.sum()
print(totals)
```

Le pays qui a le plus consommé d'alcool en 1989

```
highest_value = 0
highest_country = None

for country, conso in totals.items():
    if conso > highest_value:
        highest_value = conso
        highest_country = country
print("Le pays qui a le plus consommé d'alcool est {0} avec
↪ {1}".format(highest_country,highest_value))
```

10.7 TP sur Numpy

Maintenant que nous avons découvert NumPy, nous allons tester vos connaissances. Nous allons commencer par quelques tâches simples, puis nous vous poserons des questions plus complexes.

1. Importer NumPy

```
import numpy as np
```

2. Créer un tableau de 10 zéros

```
np.zeros(10)
```

3. Créer un tableau de 10 uns

```
np.ones(10)
```

4. Créer un tableau de 10 cinq

```
np.ones(10) * 5
```

5. Créer un tableau des nombres entiers de 10 à 50

```
np.arange(10,51)
```

6. Créer un tableau de tous les entiers pairs de 10 à 50

```
np.arange(10,51,2)
```

7. Créer une matrice 3x3 avec des valeurs allant de 0 à 8

```
np.arange(9).reshape(3,3)
```

8. Créer une matrice d'identité 3x3

```
np.eye(3)
```

9. Utiliser NumPy pour générer un nombre aléatoire entre 0 et 1 (NOTE : La valeur de votre résultat devrait être différente de celle indiquée ci-dessous.)

```
np.random.rand(1)
```

10. Utiliser NumPy pour générer un tableau de 25 nombres aléatoires tirés d'une distribution normale standard (NOTE : Les valeurs de votre résultat devraient être différentes de celles indiquées ci-dessous.).

```
np.random.randn(25)
```

11. Créer la matrice suivante :

```
array([[0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1 ],
       [0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 ],
       [0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 ],
       [0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4 ],
       [0.41, 0.42, 0.43, 0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5 ],
       [0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59, 0.6 ],
       [0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7 ],
       [0.71, 0.72, 0.73, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8 ],
       [0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.9 ],
       [0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.  ]])
```

FIGURE 10.3 – Matrice à coder

```
np.arange(1,101).reshape(10,10) / 100
```

12. Créer un tableau de 20 points linéairement espacés entre 0 et 1

```
np.linspace(0,1,20)
```

Indexation et sélection NumPy

On vous donne maintenant une matrice de départ (n'oubliez pas d'exécuter la cellule ci-dessous !), et on vous demande de reproduire les sorties de la matrice résultante

```
mat = np.arange(1,26).reshape(5,5)
mat
```

13. Écrire un code qui reproduit la sortie indiquée ci-dessous :

```
array([[12, 13, 14, 15],
       [17, 18, 19, 20],
       [22, 23, 24, 25]])
```

FIGURE 10.4 – Résultat à produire

```
mat[2:,1:]
```

14. Écrire le code qui reproduit la sortie suivante : 20

```
mat[3,4]
```

15. Écrire le code qui reproduit la sortie montrée ci-dessous :

```
array([[ 2],
       [ 7],
       [12]])
```

FIGURE 10.5 – Sortie à reproduire

16. Écrire le code qui reproduit la sortie suivante : *array([21, 22, 23, 24, 25])*

```
mat[4,:]
```

17. Écrire le code qui reproduit la sortie montrée ci-dessous

```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

FIGURE 10.6 – Sortie à reproduire

```
mat[3:5,:]
```

Opérations NumPy

18. Obtenir la somme de toutes les valeurs de mat

```
mat.sum()
```

19. Obtenir l'écart type des valeurs de mat

```
mat.std()
```

20. Obtenir la somme de toutes les colonnes de mat

```
mat.sum(axis=0)
```

Question Bonus

Nous allons beaucoup travailler avec des données aléatoires avec NumPy, mais existe-t-il un moyen de s'assurer que nous obtenons toujours les mêmes nombres aléatoires ? Testez la solution ci-dessous

```
np.random.seed(101)
```

10.8 Analyse des données avec Pandas

TP :

- Analyse d'un fichier de ventes.

- Prédiction de notes d'étudiants avec régression linéaire.

Chapitre 11

Module 11 : Python pour la cybersécurité

Objectifs :

— S'initier au développement des outils de sécurité avec python

Contenu :

- Automatisation d'audits de sécurité (scan de ports, analyse de logs).
- Hashing et cryptographie avec hashlib, cryptography.
- Manipulation de paquets avec scapy.

TP :

- Scanner de ports simple.
- Générateur de mots de passe sécurisés.