Chapitre 1 : Rappels sur la programmation en Python

Dr. Ishak ABDI — ishak.abdi@univ-jijel.dz
Département de Génie civil et hydraulique — Université de Jijel

Concepts de base en informatique et outils numériques 💻

Objectif

> Ce chapitre consolide les bases en **programmation Python** et en **environnement informatique**. Il traite des **systèmes d'exploitation**, des **réseaux**, de l'**installation de Python**, ainsi que des notions clés de **programmation** (variables, conditions, boucles, fonctions, modules). Il introduit enfin la **manipulation des structures de données** et l'usage de **bibliothèques scientifiques** comme *NumPy* et *Pandas*.

2. Bases de la programmation Python

Objectif :

> commencer à utiliser Python avec des exemples simples.
> Utiliser l'éditeur **IDLE** (ou tout autre éditeur Python).

Exemple : Hello World

1. Ouvrez votre éditeur Python.
2. Tapez le code suivant :

```
print("Hello World!")
```

Ce programme affiche le texte `Hello World!` à l'écran.

Obtenir de l'aide en Python

- La commande **help()** permet d'accéder à l'aide intégrée de Python depuis l'interpréteur.
- Explorez toutes les fonctionnalités et modules disponibles.
- Appuyez sur **q** pour quitter l'aide et revenir à l'invite Python.

Variables en Python

- Une **variable** est définie avec l'opérateur d'affectation `=`.
- Python est **dynamiquement typé** :
  - Le type n'a pas besoin d'être déclaré à l'avance.
  - Le type peut changer au cours du programme.
- Les valeurs peuvent provenir :
  - de **constantes**,
  - de **calculs avec d'autres variables**,
  - du **résultat d'une fonction**.

Exemple : Création et utilisation de variables en Python

- Utiliser l'éditeur **IDLE** (ou tout autre éditeur Python).
- Tapez le code suivant :

```
>>> x = 3
>>> x
3
```

Écriture de commandes et scripts Python

- Dans **IDLE**, on peut exécuter **une commande à la fois**.
- **Attention** : à la fermeture d'IDLE, toutes les variables et données sont perdues.
- Pour écrire un programme plus long :
  - Utiliser un **éditeur de texte** pour préparer le code.
  - Exécuter le fichier en tant qu'**entrée pour l'interpréteur**.
- Ce fichier s'appelle un **script Python** et est sauvegardé avec l'extension **.py**.

Exemple : Calculs avec des variables en Python

- Utilisation de variables dans des calculs et affichage avec `print()` :

```
# Calcul simple
x = 3
y = 3 * x
print(y)  # Affiche 9
```

```
# Formule y = ax + b
a = 2
b = 5
x = 3
y = a * x + b
print(y)  # Affiche 11
```

Noms et règles des variables en Python

- Une variable peut avoir :
  - un **nom court** : `x`, `y`
  - un **nom descriptif** : `sum`, `amount`, etc.
- **Pas besoin de déclarer le type** avant usage (contrairement à C/C++).
- **Règles de base pour les noms de variables** :
  - Commencer par une **lettre** ou un **underscore** `_`
  - Ne **pas commencer par un chiffre**
  - Contenir uniquement des **caractères alphanumériques** et `_`
  - **case-sensitive** : `amount`, `Amount` et `AMOUNT` sont trois variables différentes

Exemple de noms de fichiers et bonnes pratiques

- Python et la plupart des systèmes de fichiers permettent des **noms descriptifs**.
  - Exemple : `pr_day_CNRM-CM6-1_historical_r1i1p1f2_gr_19860101-19901231.nc`

Conseils pour nommer vos fichiers :

> - **Type de données** : `pr_day` → précipitations journalières
> - **Modèle utilisé** : `CNRM-CM6-1`
> - **Période des données** : `19860101-19901231`
> - Respecter les règles : lettres, chiffres, underscore, **pas d'espaces**

Règles de nommage des variables en Python

- Les noms de variables commencent généralement par une **minuscule**
  (ex. `variable`, `compteur`, `tableDesMatieres`).
- Les **majuscules** peuvent apparaître **à l'intérieur du nom** pour améliorer la lisibilité
  → Convention du *camelCase* : `tableDesMatieres`
- Il s'agit d'une simple convention, mais elle est largement respectée.

Mots réservés en Python

Vous **ne pouvez pas utiliser** les mots suivants comme noms de variables :
Ils sont utilisés par le langage lui-même:

`and` `as` `assert` `break` `class` `continue`
`def` `del` `elif` `else` `except` `False`
`finally` `for` `from` `global` `if` `import`
`in` `is` `lambda` `None` `nonlocal` `not`
`or` `pass` `raise` `return` `True` `try`
`while` `with` `yield`

Types numériques en Python

- Python comprend **trois types numériques** :
  - **int** : entiers
  - **float** : nombres à virgule flottante
  - **complex** : nombres complexes

Exemple :

- Les variables numériques sont créées automatiquement lors de l'affectation d'une valeur.

```
x = 1       # int
y = 2.8     # float
z = 3 + 2j  # complex
```

Détermination automatique du type en Python

- Vous pouvez **assigner une valeur à une variable** sans vous soucier de son type.
- Python détermine automatiquement le type de chaque variable :

```
print(type(x))
print(type(y))
print(type(z))
```

Les chaînes de caractères en Python

- Les **chaînes de caractères** (strings) sont entourées de **guillemets simples ou doubles** :
  `'Hello'` est équivalent à `"Hello"`
- Elles peuvent être affichées avec la fonction `print()` :

```
print("Hello")
```

Exemple:

```
a = "Hello World!"

print(a)              # Affiche la chaîne entière
print(a[1])           # Affiche le 2e caractère
print(a[2:5])         # Affiche les caractères 3 à 5
print(len(a))         # Longueur de la chaîne
print(a.lower())      # Convertit en minuscules
print(a.upper())      # Convertit en majuscules
print(a.replace("H", "J"))  # Remplace 'H' par 'J'
print(a.split(" "))         # Sépare la chaîne en liste de mots
```

- Comme on le voit dans l'exemple, Python propose de **nombreuses fonctions intégrées** pour manipuler les chaînes de caractères.
    - L'exemple montre seulement quelques-unes d'entre elles.
- Les chaînes en Python sont des **tableaux d'octets**, et on peut utiliser un **indice** pour accéder à un caractère spécifique dans la chaîne, comme illustré dans le code.

Saisie de chaînes en Python

- Python permet de **demander une saisie à l'utilisateur** depuis la ligne de commande.
- On utilise la fonction `input()` pour récupérer la valeur saisie.

Exemple : demander le nom de l'utilisateur

```
x = input("Entrez votre nom : ")
print("Hello, " + x)
```

Manipulation des chaînes en Python

Indexation des chaînes 🧵

```
>>> S = 'Spam'
>>> len(S)                # Longueur
4
>>> S[0]                  # Premier caractère
'S'
>>> S[1]                  # Deuxième caractère
'p'

>>> S[-1]                 # Dernier caractère
'm'
>>> S[-2]                 # Avant-dernier caractère
'a'

>>> S[-1]                 # Indexation négative
'm'
>>> S[len(S)-1]           # Même résultat, version longue
'm'
```

Manipulation des chaînes en Python

✂️ Slicing (extraction de sous-chaînes)

```
>>> S = 'Spam'
>>> S[1:3]                  # De l'index 1 à 2
'pa'

>>> S[1:]                   # Tout après le premier (1:len(S))
'pam'
>>> S                       # S n'a pas changé
'Spam'

>>> S[0:3]                  # Tout sauf le dernier
'Spa'
>>> S[:3]                   # Identique à S[0:3]
'Spa'

>>> S[:-1]                  # Tout sauf le dernier (plus simple)
'Spa'
>>> S[:]                    # Copie complète de S (0:len(S))
'Spam'
```

Manipulation des chaînes en Python

Concaténation et répétition 🔗

```
>>> S = 'Spam'
>>> S + 'xyz'             # Concaténation
'Spamxyz'

>>> S * 8                 # Répétition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Immuabilité des chaînes de caractères 🔒

- En Python, les chaînes sont immutables :
  - une fois créées, elles ne peuvent pas être modifiées directement.
  - Toutes les opérations sur les chaînes produisent un nouvel objet, sans altérer l'original.

```
>>> S
'Spam'

>>> S[0] = 'z'             # Impossible : les chaînes sont immuables
...erreur omise...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]        # On crée une nouvelle chaîne
>>> S
'zpam'
```

Les listes en Python

- Les listes sont des séquences ordonnées.
- Elles peuvent contenir des objets de types variés.
- Leur taille est modifiable : on peut ajouter ou supprimer des éléments.
- Les listes sont mutables, c'est-à-dire modifiables en place (contrairement aux chaînes).

Opérations sur les séquences (Listes)

- Les listes étant des séquences, elles supportent les mêmes opérations que les chaînes de caractères :
- Indexation
- Longueur `len`
- Slicing (extraction de sous-listes)
- Concaténation et répétition
- Résultats toujours sous forme de liste

Exemple :

```
>>> L = [123, 'spam', 1.23]   # A list of three different-type objects
>>> len(L)                    # Number of items in the list
3
>>> L[0]          # Indexing by position
123

>>> L[:-1]        # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6] # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]

>>> L             # We're not changing the original list
[123, 'spam', 1.23]
```

Opérations spécifiques aux listes

- Les listes Python sont similaires aux tableaux dans d'autres langages, mais elles sont plus flexibles :
  - Pas de contrainte de type : une liste peut contenir des éléments de types différents (`int` , `str` , `float` , etc.).
  - Taille dynamique : elles peuvent grandir ou rétrécir selon les opérations.

```
>>> L.append('NI')   # Ajouter un élément à la fin
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)         # Supprimer l'élément à l'index 2
1.23
>>> L                 # On peut aussi utiliser del L[2]
[123, 'spam', 'NI']
```

D’autres méthodes importantes :

- insert(index, valeur) : insérer un élément à une position spécifique
- remove(valeur) : supprimer un élément par sa valeur
- Les listes sont mutables, la plupart des méthodes modifient la liste en place :

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()          # Tri croissant
>>> M
['aa', 'bb', 'cc']

>>> M.reverse()       # Inversion de la liste
>>> M
['cc', 'bb', 'aa']
```

Imbrication (Nesting) des listes

- Une caractéristique pratique des types de données de base de Python est qu'ils supportent l'imbrication arbitraire :
  - Les structures peuvent être combinées à n'importe quel niveau, autant que nécessaire.
  - Exemple : une liste peut contenir un *dictionnaire*, qui contient une autre liste, etc.
  - Une application courante : représenter des matrices ou des tableaux multidimensionnels.

Exemple

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

>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> M[1]       # Obtenir la ligne 2
[4, 5, 6]

>>> M[1][2]    # Obtenir l'élément 3 de la ligne 2
6
```

Compréhensions de listes (List Comprehensions)

- Python propose une opération avancée appelée list comprehension :
  - Une manière concise et puissante de traiter les séquences, y compris les matrices.
  - Permet de créer de nouvelles listes à partir d'anciennes en une seule ligne.

```
>>> col2 = [row[1] for row in M]  # Extraire les éléments de la colonne 2
>>> col2
[2, 5, 8]

>>> M   # La matrice originale reste inchangée
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List Comprehensions avancées

- Les list comprehensions peuvent être plus complexes :

```
>>> [row[1] + 1 for row in M]  # Ajouter 1 à chaque élément de la colonne 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]  # Filtrer les éléments impairs
[2, 8]
```

🗂️ Dictionnaires (Dictionaries)

Les **dictionnaires Python** sont très différents des listes ou chaînes :

- Ce ne sont pas des **séquences**, mais des **mappings** (associations clé-valeur).
- Ils **stockent les objets par clé**, et non par position.
- Les mappings **n'ont pas d'ordre fiable** de gauche à droite.
- Les dictionnaires sont **mutables** : ils peuvent être modifiés sur place et peuvent **grandir ou rétrécir** à volonté.
- Les dictionnaires sont le **seul type de mapping** dans les objets de base de Python.

Opérations sur les dictionnaires (Mapping Operations)

- Les dictionnaires s'écrivent avec des accolades `{}` et contiennent des paires clé: valeur.
- Ils sont utiles pour associer des valeurs à des clés, par exemple pour décrire des propriétés d'un objet.

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

Indexation et modification des dictionnaires

- On peut accéder et modifier les valeurs associées aux clés.
- La syntaxe utilise des crochets `[]`, mais l'élément est une clé, pas une position.

```
>>> D['food']  # Récupérer la valeur associée à la clé 'food'
'Spam'

>>> D['quantity'] += 1  # Ajouter 1 à la valeur de 'quantity'
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

- On peut créer un dictionnaire vide et ajouter des clés une par une :

```
>>> D = {}
>>> D['name'] = 'Bob'  # Création de clés par affectation
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print(D['name'])
Bob
```

Dictionnaires imbriqués (Nesting)🗃️

- Parfois, les informations sont plus complexes et nécessitent des structures imbriquées.
- Exemple : enregistrer le prénom, le nom et plusieurs postes d'une personne.

```
rec = {'name': {'first': 'Bob', 'last': 'Smith'},
'job': ['dev', 'mgr'],
'age': 40.5}
```

Tuples

- Un tuple est similaire à une liste, mais immutable (on ne peut pas le modifier).
- Comme les listes, c'est une séquence qui peut contenir des objets de types variés et être imbriquée.
- Syntaxe : parenthèses au lieu de crochets `()`.
- Supporte les opérations sur séquences : indexation, slicing, concaténation, répétition.

```
>>> T = (1, 2, 3, 4)  # Tuple de 4 éléments
>>> len(T)  # Longueur
4
>>> T + (5, 6)  # Concaténation
(1, 2, 3, 4, 5, 6)
>>> T[0]  # Indexation
1
```

Liste vs Tuple

Caractéristique	Liste (`list`)	Tuple (`tuple`)
Mutabilité	Modifiable (mutable)	Non modifiable (immutable)
Syntaxe	`[1, 2, 3]`	`(1, 2, 3)`
Taille	Peut croître ou rétrécir	Fixe après création
Usage	Données modifiables, collections	Données constantes, clés de dictionnaires
Performance	Un peu plus lente	Plus rapide pour accès et itérations

Fichiers en Python

- Les objets fichiers sont l'interface principale pour accéder aux fichiers sur votre ordinateur.
- Type fondamental : mais pas de syntaxe littérale spécifique pour les créer.
- Création : utiliser la fonction intégrée `open()`, en précisant :

le nom du fichier

- le mode d'ouverture (`r`, `w`, `a`, etc.)

example

```
f = open('data.txt', 'w')  # Crée un nouveau fichier en mode écriture
>>> f.write('Hello\n')           # Écrit une chaîne de caractères
6                               # Retourne le nombre de caractères écrits
>>> f.write('world\n')           # Écrit une autre chaîne
6
>>> f.close()                    # Ferme le fichier pour sauvegarder les données
```

```
>>> f = open('data.txt')        # 'r' est le mode par défaut
>>> text = f.read()             # Lit tout le contenu du fichier dans une chaîne
>>> text
'Hello\nworld\n'
>>> print(text)                 # Affiche le contenu interprété
Hello
world
>>> text.split()                # Transforme le contenu en liste de mots
['Hello', 'world']
```

Autres types de base

- Au-delà des types de base classiques, Python propose d'autres types qui peuvent ou non être considérés comme fondamentaux, selon la définition adoptée.
- Ensembles (`set`) :
  - Ajout récent au langage.
  - Ni séquences, ni mappings.
  - Collections non ordonnées, uniques et immuables.
  - Création :

```
s = set([1, 2, 3, 2])  # À partir d'une liste
s = {1, 2, 3}           # Littéral Python 3.0+
```

Synthèse

Type d'objet	Exemple / création
Nombres	`1234`, `3.1415`, `3+4j`, `Fraction`
Chaînes	`'spam'`, `"guido's"`, `b'a\x01c'`
Listes	`[1, [2, 'three'], 4]`
Dictionnaires	`{'food': 'spam', 'taste': 'yum'}`
Tuples	`(1, 'spam', 4, 'U')`
Fichiers	`myfile = open('eggs', 'r')`
Ensembles	`set('abc')`, `{'a', 'b', 'c'}`
Autres types de base	`Booleans`, `types`, `None`
Unités de programme	`Functions`, `modules`, `classes`
Types liés à l'implémentation	`Compiled code`, `stack tracebacks`