ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR

Membre de HONORIS UNITED UNIVERSITIES

Introduction à Python - Les Bases

Dr. Hamza Abouabid
H.abouabid@emsi.ma

2024-2025

Objectifs du Cours

- **Objectifs Généraux** :
  - Comprendre les concepts de base de la programmation en Python.
  - Apprendre à écrire et exécuter des scripts Python simples.
  - Se familiariser avec les structures de données et les types de données en Python.
  - Développer des compétences en résolution de problèmes à l'aide de Python.
- **Objectifs Spécifiques** :
  - Avantages de Python.
  - Différences Python 2 vs 3.
  - Utilisation de l'interpréteur Python.
  - Avantages de l'interpréteur IPython.
- **Compétences à Acquérir** :
  - Syntaxe de base et structures de contrôle.
  - Utilisation des listes, tuples, dictionnaires.
  - Manipulation de fichiers et données (JSON, CSV, Pandas).
  - Organisation du code en modules et packages.

- Objectifs du cours et attentes.
- Brève présentation de l'importance de Python.
- Différences entre Python 2 et Python 3.
- Installation de Python.
- Votre premier programme Python.

Qu'est-ce que Python ?

- **Introduction à Python** :
  - Python est un *langage de programmation interprété*, ce qui signifie que les instructions sont exécutées directement sans compilation préalable. Cette caractéristique le rend particulièrement adapté pour le prototypage rapide et les itérations de développement.
  - En tant que *langage de haut niveau*, Python permet une abstraction significative par rapport au langage machine, rendant le code plus lisible, compréhensible et maintenable.

Qu'est-ce que Python ?

- **Caractéristiques Principales de Python** :
  - *Facilité d'apprentissage* : Sa syntaxe claire et sa communauté active en font un choix excellent pour les débutants en programmation.
  - *Interprétable* : Python est exécuté dans un environnement d'exécution, ce qui facilite le test et le débogage de programmes complexes.
  - *Polyvalence* : Utilisé dans une vaste gamme d'applications, de l'automatisation de scripts simples au développement de systèmes complexes, Python est extrêmement adaptable.
  - *Applications diverses* : Python est omniprésent dans le développement web (via des frameworks comme Django et Flask), l'analyse de données (avec des outils tels que Pandas et NumPy), l'intelligence artificielle (grâce à des bibliothèques comme TensorFlow et PyTorch), et dans le calcul scientifique.

Qu'est-ce que Python ?

- **Python dans la Science des Données** :
  - Python est largement utilisé dans la science des données pour des tâches telles que le nettoyage de données, l'analyse statistique, la visualisation de données, et le machine learning.
  - Des bibliothèques comme Pandas pour la manipulation de données, Matplotlib et Seaborn pour la visualisation, et Scikit-learn pour le machine learning, rendent Python particulièrement puissant pour l'analyse de données.
  - Sa capacité à intégrer avec d'autres langages et outils, comme SQL pour les bases de données, et son support pour le calcul parallèle et distribué, le rendent idéal pour travailler avec de grandes ensembles de données.

Différences entre Python 2 et Python 3

print : Instruction vs Fonction

- **Python 2 :** `print` est une instruction, il ne nécessite pas de parenthèses.
  - Exemple : `print "Bonjour"`
- **Python 3 :** `print` est une fonction, les parenthèses sont obligatoires.
  - Exemple : `print("Bonjour")`

Différences entre Python 2 et Python 3

Chaînes de caractères : Unicode vs ASCII

- **Python 2 :** Par défaut, les chaînes de caractères (`str`) sont des séquences d'octets ASCII.
- **Python 3 :** Les chaînes de caractères (`str`) sont Unicode par défaut, facilitant la gestion des caractères internationaux.
- Pour obtenir des chaînes Unicode en Python 2, il fallait utiliser le type `unicode`.

Différences entre Python 2 et Python 3

Division des Entiers

- **Python 2 :** La division entre deux entiers (`5 / 2`) retourne un entier (`2`).
- **Python 3 :** La division entre deux entiers (`5 / 2`) retourne un nombre flottant (`2.5`).
- Pour obtenir une division entière en Python 3, utilisez l'opérateur `//` :
  - Exemple : `5 // 2` donne `2`.

Différences entre Python 2 et Python 3

Fonction `range()`

- **Python 2 :** `range()` retourne une liste. Utilisez `xrange()` pour obtenir un générateur.
- **Python 3 :** `range()` retourne un objet de type générateur, ce qui est plus efficace en termes de mémoire.

Différences entre Python 2 et Python 3

Gestion des Exceptions

- **Python 2 :** La syntaxe pour lever une exception est : `except ValueError, e`
- **Python 3 :** La syntaxe a changé pour : `except ValueError as e`

Différences entre Python 2 et Python 3

`input()` vs `raw_input()`

- **Python 2 :**
  - `input()` évalue l'entrée comme du code Python.
  - `raw_input()` est utilisé pour lire une chaîne de caractères en entrée.
- **Python 3 :**
  - `input()` est utilisé pour lire des chaînes de caractères (comportement de `raw_input()`).

Différences entre Python 2 et Python 3

Métaclasses

- **Python 2 :** Les métaclasses sont définies avec la syntaxe : `__metaclass__ = MetaClassName`
- **Python 3 :** Utilisez la syntaxe dans la définition de la classe : `class MyClass(metaclass=MetaClassName)`

Différences entre Python 2 et Python 3

Bibliothèques Standard Modernisées

- Certaines bibliothèques ont été renommées ou modifiées.
- **Exemple :**
  - `ConfigParser` (Python 2) est devenu `configparser` en Python 3.
  - `cPickle` a été fusionné avec `pickle`.

Différences entre Python 2 et Python 3

Support et Maintenance

- **Python 2 :** La dernière version était Python 2.7, et le support officiel a pris fin en janvier 2020.
- **Python 3 :** Python 3 est activement maintenu et mis à jour, avec de nouvelles fonctionnalités et améliorations.

Différences entre Python 2 et Python 3

Résumé des Différences

- Python 2 est obsolète et ne reçoit plus de support.
- Python 3 est le présent et l'avenir du langage, avec de meilleures performances, une meilleure gestion de la mémoire, et un support accru des technologies modernes.

Installation de Python

- **Guide d'Installation étape par étape** :
  - *Téléchargement* : Accédez au site officiel de Python, python.org, et téléchargez la dernière version stable pour votre système d'exploitation.
  - *Installation* : Lancez l'installateur téléchargé. Sur Windows, assurez-vous de cocher l'option "Add Python to PATH" avant de commencer l'installation.
  - *Configuration Post-Installation* : Il peut être nécessaire de configurer certaines variables d'environnement, surtout sous Linux et macOS, pour faciliter l'accès aux commandes Python et Pip (gestionnaire de paquets Python) depuis le terminal.
- **Vérification de l'Installation de Python** :
  - Après l'installation, ouvrez un terminal ou une invite de commande et tapez ‘python –version‘ (ou ‘python3 –version‘ sur certains systèmes Linux/macOS). Si Python est correctement installé, cette commande affichera la version installée.
  - Il est également recommandé de vérifier l'installation de Pip, le gestionnaire de paquets de Python, en exécutant ‘pip –version‘.

Installation de Python

- **Installation de Python sur Différentes Plateformes** :
  - *Windows* : L'installateur inclut l'option d'ajouter Python au PATH, facilitant l'accès depuis l'invite de commande.
  - *macOS* : Python peut être installé via l'installateur téléchargé ou en utilisant des gestionnaires de paquets comme Homebrew.
  - *Linux* : Python est souvent pré-installé sur de nombreuses distributions Linux. Des versions spécifiques peuvent être installées via le gestionnaire de paquets de la distribution, comme apt pour Ubuntu ou yum pour Fedora.

Votre premier programme Python

```
print("Bonjour le monde!")
```

- **Structure du Programme** :
  - Ce programme est un exemple classique de "Hello World" dans le monde de la programmation.
  - En une seule ligne de code, il démontre la capacité de Python à exécuter une tâche simple : afficher un message à l'utilisateur.
- **Explication de la Syntaxe** :
  - *La fonction print* : ‘print()‘ est une fonction intégrée en Python, utilisée pour afficher le texte ou la valeur de variable que l'utilisateur souhaite voir.
  - *Guillemets pour les chaînes de caractères* : Les guillemets (" ") sont utilisés pour délimiter des chaînes de caractères en Python.
  - *Parenthèses* : Les parenthèses après ‘print‘ indiquent qu'il s'agit d'une fonction.
  - *Importance de la simplicité* : Ce programme illustre également la nature concise et lisible de Python, qui rend le langage accessible aux débutants tout en étant puissant pour les développeurs avancés.

Exercice 1

Tâche

écrire un programme pour imprimer votre nom et la date du jour.

Correction de l'Exercice 1

```python
import datetime

# Afficher le nom
print("Nom: Hamza Abouabid")

# Afficher la date du jour
print("Date: ", datetime.date.today())
```

Partie 2

Les Bases de Python :

- Variables et Types de Données
- Opérateurs en Python
- Structures de Contrôle : If-else
- Structures de Contrôle : Boucles

Variables et Types de Données

- **Définition des variables**
  - Une variable est un conteneur qui stocke des données.
  - Elle permet de référencer et manipuler ces données dans le programme.
- **Types de données courants**
  - `int` : Entiers (ex. : 5, -3, 42)
  - `float` : Nombres à virgule flottante (ex. : 3.14, -0.001, 2.0)
  - `string` : Chaînes de caractères (ex. : "Bonjour", 'A', "123")
  - `boolean` : Valeurs booléennes (True ou False)

- **Exemple : Déclaration et utilisation de différentes variables en Python**
  - Déclaration de variables
    - `age = 25` # Variable entière
    - `temperature = 23.5` # Variable flottante
    - `nom = "Alice"` # Variable chaîne de caractères
    - `est_etudiant = True` # Variable booléenne
  - Utilisation des variables
    - Calculer l'année de naissance

```
annee_actuelle = 2024
annee_naissance = annee_actuelle - age
print("annee de naissance :",
    annee_naissance)
```

    - Afficher un message personnalisé

```
print("Bonjour, je m'appelle", nom, "et j'
    ai", age, "ans.")
```

Exemple de Code Complet

Script Python Exemple

```
# Declaration des variables
age = 25
temperature = 23.5
nom = "Alice"
est_etudiant = True

# Calcul de l'annee de naissance
annee_actuelle = 2024
annee_naissance = annee_actuelle - age
print("Annee de naissance :", annee_naissance)

# Affichage d'un message personnalise
print("Bonjour, je m'appelle", nom, "et j'ai", age, "
    ans.")
```

Exercice 2

Tâche

Créer des variables de différents types et les imprimer.

Consignes

1. Déclarez une variable entière nommée `nombre`.
2. Déclarez une variable flottante nommée `prix`.
3. Déclarez une variable chaîne de caractères nommée `produit`.
4. Déclarez une variable booléenne nommée `en_stock`.
5. Assignez des valeurs appropriées à chaque variable.
6. Imprimez chacune des variables avec un message explicatif.

Exemple de Résultat Attendu

```
nombre = 10
prix = 19.99
produit = "Livre"
en_stock = True

print("Nombre d'articles :", nombre)
print("Prix unitaire :", prix, "DHs")
print("Produit :", produit)
print("En stock :", en_stock)
```

Fonctions `type` et `id` en Python

- **Fonction `type()`**
  - Retourne le type d'une variable.
  - Utile pour vérifier le type de données stocké dans une variable.
- **Fonction `id()`**
  - Retourne l'identifiant unique de l'objet en mémoire.
  - Permet de vérifier si deux variables pointent vers le même objet.

Exemples avec type() et id()

Utilisation de type()

```
print(type(age))             # <class 'int'>
print(type(temperature))     # <class 'float'>
print(type(nom))             # <class 'str'>
print(type(est_etudiant))    # <class 'bool'>
```

Utilisation de id()

```
print(id(age))               # Exemple : 140705302930848
print(id(temperature))       # Exemple : 140705302930944
print(id(nom))               # Exemple : 140705302930992
print(id(est_etudiant))      # Exemple : 140705302931040
```

Exemple de Code avec type() et id()

Script Python Exemple

```
# Utilisation de type()
print("Type de age :", type(age))
print("Type de temperature :", type(temperature))
print("Type de nom :", type(nom))
print("Type de est_etudiant :", type(est_etudiant))

# Utilisation de id()
print("ID de age :", id(age))
print("ID de temperature :", id(temperature))
print("ID de nom :", id(nom))
print("ID de est_etudiant :", id(est_etudiant))
```

Résultat Attendu

```
Type de age : <class 'int'>
Type de temperature : <class 'float'>
Type de nom : <class 'str'>
Type de est_etudiant : <class 'bool'>
ID de age : 140705302930848
ID de temperature : 140705302930944
ID de nom : 140705302930992
ID de est_etudiant : 140705302931040
```

Nommage des Variables en Python

- **Règles de Base**
  - Les noms de variables doivent commencer par une lettre ou un underscore (_).
  - Ils ne peuvent pas commencer par un chiffre.
  - Les noms peuvent contenir des lettres, des chiffres et des underscores.
  - Les noms sont sensibles à la casse (`age` et `Age` sont différents).
- **Bonnes Pratiques**
  - Utiliser des noms significatifs et descriptifs.
  - Suivre la convention `snake_case` pour la lisibilité.
  - éviter les abréviations obscures.
  - Ne pas utiliser de mots réservés par Python (comme `class`, `def`, etc.).
- **Mauvais Exemples**
  - `a`, `b1`, `temp`
  - `NombreArticles`, `PrixUnit`
- **Bons Exemples**
  - `nombre_articles`, `prix_unitaire`
  - `annee_naissance`, `est_etudiant`

Exemples de Nommage des Variables

Mauvais Nommage

```
a = 25
b1 = "Alice"
temp = 23.5
NombreArticles = 10
PrixUnit = 19.99
```

Bon Nommage

```
age = 25
nom_etudiant = "Alice"
temperature = 23.5
nombre_articles = 10
prix_unitaire = 19.99
```

Conseils Supplémentaires pour le Nommage

- **Utiliser des Noms en Anglais**
  - Favorisez l'anglais pour une meilleure compatibilité internationale.
  - Exemple : `is_student` au lieu de `est_etudiant`.
- **éviter les Noms Trop Longs ou Trop Courts**
  - Trouvez un équilibre entre descriptivité et concision.
  - Exemple : `date_de_naissance` peut être trop long, préférez `dob` (date of birth) si le contexte le permet.
- **Utiliser des Préfixes ou Suffixes Pertinents**
  - Exemple : `is_active` pour une variable booléenne indiquant l'état actif.
- **éviter les Noms Répétitifs ou Ambigus**
  - Choisissez des noms qui reflètent clairement le contenu ou le rôle de la variable.

Opérateurs en Python

Introduction aux Opérateurs en Python

- Les opérateurs sont des symboles ou des mots réservés utilisés pour effectuer des opérations sur des valeurs ou des variables.
- Ils permettent de manipuler les données et de construire des expressions.
- Python propose plusieurs catégories d'opérateurs :
  - Opérateurs arithmétiques
  - Opérateurs de comparaison
  - Opérateurs logiques
  - Opérateurs d'assignation
  - Opérateurs spéciaux

Opérateurs Arithmétiques

Opérateur	**Description**
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste de la division)
**	Exponentiation
//	Division entière

Table – Opérateurs arithmétiques en Python

Exemples

```
a = 10
b = 3

print(a + b)    # 13
print(a - b)    # 7
print(a * b)    # 30
print(a / b)    # 3.333...
print(a % b)    # 1
print(a ** b)   # 1000
print(a // b)   # 3
```

Opérateurs de Comparaison

Opérateur	Description
==	Égal à
!=	Différent de
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à

Table – Opérateurs de comparaison en Python

Exemples

```
a = 5
b = 10

print(a == b)  # False
print(a != b)  # True
print(a > b)   # False
print(a < b)   # True
print(a >= 5)  # True
print(b <= 10) # True
```

Opérateurs Logiques

Opérateur	Description
`and`	ET logique
`or`	OU logique
`not`	NON logique

Table – Opérateurs logiques en Python

Exemples

```
a = True
b = False

print(a and b)  # False
print(a or b)   # True
print(not a)    # False
print(not b)    # True
```

Opérateurs d'Assignation

Opérateur	Description
=	Assignation
+=	Addition et assignation
-=	Soustraction et assignation
*=	Multiplication et assignation
/=	Division et assignation
%=	Modulo et assignation
**=	Exponentiation et assignation
//=	Division entière et assignation

Table – Opérateurs d'assignation en Python

Exemples

```
a = 5
a += 3  # a = a + 3 => 8
a -= 2  # a = a - 2 => 6
a *= 4  # a = a * 4 => 24
a /= 3  # a = a / 3 => 8.0
a %= 5  # a = a % 5 => 3.0
a **= 2 # a = a ** 2 => 9.0
a //= 2 # a = a // 2 => 4.0
```

Opérateurs Spéciaux

Opérateur	Description
`is`	Vérifie si deux variables référencent le même objet
`is not`	Vérifie si deux variables ne référencent pas le même objet
`in`	Vérifie si une valeur est présente dans une séquence
`not in`	Vérifie si une valeur n'est pas présente dans une séquence

Table – Opérateurs spéciaux en Python

Exemples

```
a = 25
b = 25
c = "Alice"
d = "Alice"

print(a is b)          # True (mêmes objets entiers immuables)
print(c is d)          # True (mêmes objets string immuables)
print(30 in [10, 20, 30])   # True
print("Bob" not in "Alice") # True
```

Exercice 3

Tâche

écrire un programme pour calculer la surface d'un rectangle.

Structures de Contrôle : If-else

- Explication des instructions conditionnelles.
- Syntaxe et exemples.
- Exemple : Programme utilisant if-else pour prendre des décisions en fonction de conditions.

Qu'est-ce qu'une condition else if ?

- En Python, les instructions conditionnelles permettent de faire des choix dans le code en fonction de conditions spécifiques. Ces conditions utilisent les mots-clés **if**, **elif** (else if), et **else**.
- Syntaxe générale :

```
if condition1:
    # code si condition1 est vraie
elif condition2:
    # code si condition2 est vraie
elif condition3:
    # code si condition3 est vraie
else:
    # code si aucune condition n est vraie
```

Explication :

- **if** : Exécute un bloc d'instructions si la condition est vraie.
- **elif** : Permet d'ajouter d'autres conditions si la première est fausse.
- **else** : Exécute un bloc d'instructions si aucune des conditions précédentes n'est vraie.

Exemple : Attribution des mentions

```
note = float(input("entrer la note : "))
if note >= 16:
    print("Tres bien")
elif note >= 14:
    print("Bien")
elif note >= 12:
    print("Assez bien")
elif note >= 10:
    print("Passable")
else:
    print("Insuffisant")
```

Analyse de l'exemple

- Les conditions sont testées dans l'ordre
- Dès qu'une condition est vraie, le code correspondant est exécuté
- Les autres conditions ne sont pas testées
- Le else final sert de "filet de sécurité"

Exercice 1 : Calculateur d'IMC

Écrivez un programme qui :

- Prend en entrée le poids (en kg) et la taille (en m)
- Calcule l'IMC (poids / taille^2)
- Affiche une catégorie selon l'échelle suivante :
  - IMC $<$ 18.5 : "Maigreur"
  - 18.5 $\leq$ IMC $<$ 25 : "Normal"
  - 25 $\leq$ IMC $<$ 30 : "Surpoids"
  - IMC $<$ 30 : "Obésité"

Solution de l'exercice 1

```python
def calculer_categorie_imc(poids, taille):
    imc = poids / (taille ** 2)

    if imc < 18.5:
        categorie = "Maigreur"
    elif imc < 25:
        categorie = "Normal"
    elif imc < 30:
        categorie = "Surpoids"
    else:
        categorie = "Obesite"

    return f"IMC : {imc:.1f} - Categorie : {categorie}
        "


# Test
print(calculer_categorie_imc(70, 1.75))
```

Points clés à retenir

- Les conditions sont évaluées dans l'ordre
- Une seule branche sera exécutée
- L'ordre des conditions est important
- Le else est optionnel mais recommandé
- Pensez à tous les cas possibles

Introduction aux Boucles

Qu'est-ce qu'une boucle ?

- Une **boucle** est une structure de contrôle qui permet de répéter l'exécution d'un bloc de code plusieurs fois.

Pourquoi utiliser des boucles ?

- Automatiser des tâches répétitives sans avoir à écrire manuellement plusieurs lignes de code identiques.
- Parcourir des collections de données (listes, chaînes de caractères, etc.) et appliquer des opérations sur chaque élément.
- Faciliter la gestion et l'analyse de grandes quantités de données grâce à des itérations efficaces.
- Améliorer la lisibilité et la maintenance du code en évitant la redondance.

Les types de boucles en Python incluent principalement les boucles **for** et **while**, chacune ayant ses cas d'utilisation spécifiques.

1. La Boucle For

La boucle **for** en Python est utilisée pour itérer sur une séquence (qui peut être une liste, un tuple, un dictionnaire, un ensemble ou une chaîne de caractères).

Syntaxe :

```
for valeur in sequence:
    # Instructions a executer
```

Exemple :

```
for i in range(5):
    print(i)
```

Cet exemple affiche les nombres de 0 à 4.

2. La Boucle While

La boucle while permet d'exécuter un ensemble d'instructions aussi longtemps qu'une condition est vraie.

Syntaxe :

```
while condition:
    # Instructions a executer
```

Exemple :

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Cet exemple affiche les nombres de 0 à 4.

Contrôle des Boucles

Python propose des instructions pour contrôler le flux des boucles :

- **break** : Permet de sortir immédiatement de la boucle la plus interne.
- **continue** : Ignore le reste du code à l'intérieur de la boucle pour l'itération courante et passe à l'itération suivante.
- **else** : Bloc de code exécuté après la fin de la boucle, sauf si la boucle est terminée par un **break**.

Quelques Exemples

Exemple 1 :

```
i = 0
while i < 10:
    print("{:d} times me".format(i + 1))
    i += 1
print("Fin")
```

Sortie :

```
1 times me
2 times me
...
10 times me
Fin
```

Exemples de Boucles For

Exemple 2 :

```
for _ in range(0, 11, 1):
    print("nta/i nadi/a")
```

Sortie : Affiche 11 fois "nta/i nadi/a".

Exemple 3 :

```
for _ in range(1, 101, 1):
    print(_, end="\t")
    if _ % 10 == 0:
        print()
```

Sortie : Affiche les nombres de 1 à 100 par blocs de 10.

Exemples Supplémentaires

Exemple 4 :

```
for _ in range(1, 101, 2):
    print(_, end="\t")
    if _ % 9 == 0:
        print()
```

Sortie : Affiche les nombres impairs de 1 à 99, avec saut de ligne lorsque le nombre est un multiple de 9.

Exemple 5 :

```
for _ in range(0, 101, 2):
    print(_, end="\t")
    if _ % 10 == 0:
        print()
```

Sortie : Affiche les nombres pairs de 0 à 100 par blocs de 10.

Jeu : Deviner le Nombre

Exemple 6 :

```
n_secret = 10
inp = int(input("Entree un numero si tu trouves le
    numero secret tu gagnes"))
while inp != n_secret:
    print("Le numero entre : {:d} n'est pas 10".format
        (inp))
    inp = int(input("Veuillez entrer un nombre a
        nouveau :"))
print("Bravooooo, le numero entre est bien 10 : {:d}".
    format(n_secret))
```

Jeu : Deviner un Nombre Aleatoire

Exemple 7 :

```
import random as rd
n_secret = rd.randint(0, 10)
inp = int(input("Entree un numero si tu trouves le
    numero secret tu gagnes"))
while inp != n_secret:
    print("Le numero entre : {:d} n'est pas le secret"
        .format(inp))
    inp = int(input("Veuillez entrer un nombre a
        nouveau :"))
print("Bravooooo, le numero entre est bien le numero
    secret : {:d}".format(n_secret))
```

Introduction aux Fonctions

- **Qu'est-ce qu'une fonction ?**
  - Une fonction est un bloc de code réutilisable qui effectue une tâche spécifique.
  - Elle peut prendre des paramètres en entrée et retourner une valeur.
- **Pourquoi utiliser des fonctions ?**
  - Pour éviter la redondance du code.
  - Pour améliorer la lisibilité et la maintenance du code.
  - Pour faciliter le débogage et les tests.

Définition d'une Fonction en Python

- **Syntaxe de base**

```
def nom_de_la_fonction(param1, param2):
    # Bloc de code return resultat
```

- **Exemple simple**

```
def saluer(nom):
    print("Bonjour,", nom)
```

Appel d'une Fonction

- **Comment appeler une fonction ?**
  - Utiliser le nom de la fonction suivi de parenthèses.
- **Exemple**

```
saluer("Alice") # Affiche: Bonjour, Alice
```

- **Fonction avec retour de valeur**

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

resultat = addition(5, 3)
print("Le resultat est:", resultat) # Affiche: Le
    resultat est: 8
```

Paramètres et Arguments

- **Paramètres**
  - Variables définies dans la déclaration de la fonction.
  - Exemples : `nom`, `a`, `b` dans les fonctions précédentes.
- **Arguments**
  - Valeurs réelles passées à la fonction lors de son appel.
  - Exemples : `"Alice"`, `5`, `3`.
- **Types de paramètres**
  - *Paramètres positionnels* : Basés sur l'ordre.
  - *Paramètres nommés* : Basés sur le nom du paramètre.

Fonctions avec Paramètres par Défaut

- **Définition**
  - Les paramètres peuvent avoir des valeurs par défaut.
  - Si aucun argument n'est fourni, la valeur par défaut est utilisée.
- **Exemple**

```
def saluer(nom, message="Bonjour"):
    print(message, nom)

saluer("Alice") # Affiche: Bonjour Alice
saluer("Bob", "Salut") # Affiche: Salut Bob
```

Portée des Variables

- **Variables locales**
  - Définies à l'intérieur d'une fonction.
  - Accessibles uniquement dans cette fonction.
- **Variables globales**
  - Définies en dehors de toutes les fonctions.
  - Accessibles dans tout le programme.
- **Exemple**

```
def fonction():
    y = 5 # Variable locale
    print("y =", y)

fonction()
print("x =", x) #print("y =", y)
# Erreur: y n'est pas accessible ici
```

Introduction aux Bibliothèques

- **Qu'est-ce qu'une bibliothèque ?**
  - Un ensemble de modules et de fonctions pré-écrits.
  - Permet de réutiliser du code pour des tâches courantes.
- **Pourquoi utiliser des bibliothèques ?**
  - Gain de temps en évitant de réinventer la roue.
  - Bénéficier de code optimisé et testé par la communauté.
- **Exemples de bibliothèques**
  - `math` pour les fonctions mathématiques.
  - `random` pour les nombres aléatoires.
  - `datetime` pour la manipulation des dates et heures.

Utilisation de Bibliothèques en Python

- **Importer une bibliothèque**

```
import math
```

- **Utiliser une fonction d'une bibliothèque**

```
import math

resultat = math.sqrt(16)
print("La racine carree de 16 est:", resultat) #
    Affiche: 4.0
```

- **Importer une fonction spécifique**

```
from math import sqrt

resultat = sqrt(25)
print("La racine carree de 25 est:", resultat) #
    Affiche: 5.0
```

Les Modules en Python

- **Qu'est-ce qu'un module ?**
  - Un fichier Python contenant des définitions et du code.
  - Permet d'organiser le code en le répartissant sur plusieurs fichiers.
- **Créer un module**
  - Créer un fichier `mon_module.py` avec des fonctions.
- **Importer un module personnalisé**

```
import mon_module

mon_module.ma_fonction()
```

Introduction aux Paquets

- **Qu'est-ce qu'un paquet ?**
  - Un ensemble de modules organisés en arborescence de répertoires.
  - Chaque répertoire contient un fichier `__init__.py`.
- **Pourquoi utiliser des paquets ?**
  - Pour structurer des projets complexes.
  - Pour éviter les conflits de noms entre modules.
- **Exemple d'importation depuis un paquet**

```
from mon_paquet.mon_module import ma_fonction

ma_fonction()
```

Installer des Paquets Externes

- **Utilisation de `pip`**
  - `pip` est le gestionnaire de paquets de Python.
  - Permet d'installer des paquets tiers depuis le Python Package Index (PyPI).
- **Commande d'installation**

```
pip install nom_du_paquet
```

- **Exemple**
  - Installer la bibliothèque `requests` pour les requêtes HTTP.
  - `pip install requests`

Utilisation d'un Paquet Externe

- **Exemple avec `requests`**

```
import requests

reponse = requests.get("https://api.github.com")
print("Statut:", reponse.status_code)
```

- **Explication**
  - Importation du paquet `requests`.
  - Envoi d'une requête GET à l'API de GitHub.
  - Affichage du code de statut de la réponse.

Résumé

- Les **fonctions** permettent de structurer le code et de le rendre réutilisable.
- Les **bibliothèques** offrent des fonctionnalités supplémentaires prêtes à l'emploi.
- Les **modules** et **paquets** aident à organiser le code en projets de grande taille.
- L'utilisation de **paquets externes** via `pip` permet d'étendre les capacités de Python.

Fonctions Lambda

- **Qu'est-ce qu'une fonction lambda ?**
  - Une fonction anonyme définie avec le mot-clé `lambda`.
  - Utilisée pour des opérations simples et rapides.
- **Syntaxe**

```
lambda arguments: expression
```

- **Exemple**

```
addition = lambda x, y: x + y
print(addition(5, 3)) # Affiche: 8
```

Ouverture sur les Structures de Données

- Jusqu'ici, nous avons manipulé des variables simples.
- Les prochaines sections aborderont les **structures de données** plus complexes :
  - **Listes**
  - **Tuples**
  - **Dictionnaires**
- Ces structures permettent de stocker et manipuler des collections de données.
- Elles sont essentielles pour écrire des programmes efficaces et flexibles.

Structures de Données Avancées

Introduction aux Structures de Données en Python

- Une structure de données est un moyen d'organiser et de gérer des données pour les manipuler efficacement.
- Python offre plusieurs types de structures de données intégrées :
  - Listes
  - Tuples
  - Ensembles (Sets)
  - Dictionnaires
- Chaque structure a des caractéristiques et des usages spécifiques.

Les Listes

- Définition :
  - Une liste est une collection ordonnée et modifiable.
  - Peut contenir des éléments de types variés.
- Syntaxe : `ma_liste = [1, 2, 3, "Python", 3.14]`
- Méthodes principales :
  - `append(x)`, `remove(x)`, `sort()`, `reverse()`
- Exemples : `ma_liste.append(42)`
  `print(ma_liste[2])`

Structures de Données en Python

- Introduction aux Types de Données
- Listes et Opérations
- Tuples et leurs Caractéristiques
- Ensembles et leurs Opérations
- Dictionnaires et leur Utilisation

Introduction aux Listes Python

Caractéristiques Fondamentales

- **Définition :** Collection ordonnée et mutable
- **Création :**
  - Crochets : `[1, 2, 3]`
  - Constructeur : `list()`
  - Conversion : `list("abc")`
- **Types Supportés :** Tout type de données

Avantages

- Flexibilité maximale
- Modification facile
- Indexation rapide

Syntaxe de Base

```
ma_liste = [1, 2, 3, 4, 5]
```

Opérations Principales sur les Listes

Méthodes de Base

- `append(x)` - Ajout à la fin
- `insert(i, x)` - Insertion à l'index i
- `remove(x)` - Supprime la première occurrence
- `pop([i])` - Retire et renvoie l'élément
- `clear()` - Vide la liste

Méthodes Avancées

- `sort(key=None, reverse=False)`
- `reverse()` - Inverse l'ordre
- `copy()` - Copie superficielle
- `extend(iterable)` - Fusion de listes
- `count(x)` - Compte les occurrences

Techniques de Découpage (Slicing)

Syntaxe de Base

```
liste[début:fin:pas]  # début inclus, fin exclue
```

Exemples Pratiques

```
nombres = [0, 1, 2, 3, 4, 5]
print(nombres[2:4])    # [2, 3]
print(nombres[::2])    # [0, 2, 4]
print(nombres[::-1])   # [5, 4, 3, 2, 1, 0]
print(nombres[-3:])    # [3, 4, 5]
```

Patterns et Bonnes Pratiques

Patterns Courants

- **Compréhension de liste**
- **Filter/Map patterns**
- **Stack/Queue utilisation**
- **Buffer circulaire**

À Éviter

- Modification pendant l'itération
- Copies inutiles de grandes listes
- Recherches linéaires fréquentes
- Croissance non contrôlée

Optimisations

- Préallouer pour les grandes listes
- Utiliser deque pour les files

Exemple de Liste

```
# Création d'une liste
ma_liste = [valeur1, valeur2, valeur3, valeur4, valeur5]

# Accès aux éléments
print(ma_liste[0])  # Sortie: 1
print(ma_liste[-1]) # Sortie: 5

# Ajout d'un élément
ma_liste.append(6)  # Ajoute 6 à la fin

# Suppression d'un élément
ma_liste.remove(3)  # Supprime la première occurrence de 3
# Tri d'une liste
ma_liste.sort()     # Trie la liste par ordre croissant
# Inversion d'une liste
ma_liste.reverse()  # Inverse l'ordre des éléments
```

Tuples en Python

- **Définition :** Collection ordonnée et immuable
- **Création :**
  - Parenthèses : `(1, 2, 3)`
  - Tuple simple : `1,` ou `(1,)`
- **Avantages :**
  - Plus rapide que les listes
  - Protection contre les modifications
  - Parfait pour les données fixes

Cas d'Usage

- Coordonnées (x, y)
- Constantes
- Retours multiples

Syntaxe de Base

```
mon_tuple = (valeur1, valeur2, valeur3)
```

Opérations Avancées sur les Tuples

Manipulation des Tuples

```
# Création et unpacking
coords = (3, 4)
x, y = coords

# Tuple comme clé de dictionnaire
points = {(0, 0): 'origine',
          (1, 0): 'unité x'}

# Concaténation
t1 = (1, 2)
t2 = (3, 4)
t3 = t1 + t2  # (1, 2, 3, 4)
```

Exemple de Tuple

```
# Création d'un tuple
mon_tuple = (1, 2, 3, 4, 5)
# Accès aux éléments
print(mon_tuple[1])  # Sortie: 2
# Découpage d'un tuple
print(mon_tuple[1:4])  # Sortie: (2, 3, 4)
```

Ensembles (Sets) en Python

Syntaxe de Base

```
mon_ensemble = {valeur1, valeur2, valeur3}
```

Caractéristiques Principales

- **Non-ordonnés**
- **Éléments uniques**
- **Mutables**
- **Hashables uniquement**

Opérations Ensemblistes

- `union()` ou `|`
- `intersection()` ou `&`
- `difference()`
- `symmetric_difference()`

Applications Courantes

- Élimination des doublons
- Tests d'appartenance rapides
- Comparaison de collections

Exemple d'Ensemble

```
# Création d'un ensemble
mon_ensemble = {1, 2, 3, 4, 5}
# Ajout d'un élément
mon_ensemble.add(6)
# Suppression d'un élément
mon_ensemble.remove(2)
# Opérations sur les ensembles
autre_ensemble = {4, 5, 6, 7, 8}
print(mon_ensemble.union(autre_ensemble))
# Sortie: {1, 3, 4, 5, 6, 7, 8}
```

Dictionnaires en Python

Syntaxe de Base

```
mon_dict = {'key1': 'value1', 'key2': 'value2'}
```

Structure et Caractéristiques

- **Structure :** Collection de paires clé-valeur
- **Clés :** Doivent être uniques et hashables
- **Valeurs :** Peuvent être de tout type

Méthodes Importantes

- `.get(key, default)`
- `.setdefault(key, default)`
- `.update(other_dict)`
- `.pop(key, default)`

Pattern Courants

- Compteur de fréquence
- Cache de données
- Table de mappage
- Index inversé

Exemples Pratiques

```
# Dictionnaire par compréhension
carres = {x: x**2 for x in range(5)}

# Fusion de dictionnaires (Python 3.9+)
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = dict1 | dict2

# DefaultDict pattern
from collections import defaultdict
freq = defaultdict(int)
for mot in texte.split():
    freq[mot] += 1
```

Exemple de Dictionnaire

```
# Création d'un dictionnaire
mon_dict = {'pomme': 10, 'banane': 5, 'orange': 8}
# Accès à une valeur par clé
print(mon_dict['pomme'])  # Sortie: 10
# Ajout d'une nouvelle paire clé-valeur
mon_dict['raisin'] = 15
# Mise à jour d'une clé existante
mon_dict['banane'] = 6
```

Vue d'Ensemble des Itérations

Types d'Itération

- **Boucle For :**
  - Itération sur séquences
  - Énumération avec `enumerate()`
  - Parcours parallèle avec `zip()`
- **Boucle While :**
  - Conditions de sortie
  - Compteurs et accumulateurs

Points Clés

- Lisibilité du code
- Performance
- Gestion mémoire

Itérateurs Personnalisés

```
class MonIterateur:
    def __init__(self, limite):
        self.limite = limite
        self.compteur = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.compteur < self.limite:
            self.compteur += 1
            return self.compteur
        raise StopIteration
```

Compréhensions et Expressions Générateurs

Compréhensions

```
# Liste
[x**2 for x in range(10)
if x % 2 == 0]

# Dictionnaire
{x: x**2 for x in range(5)}

# Ensemble
{x%3 for x in range(10)}
```

Générateurs

```
# Expression générateur
sum(x**2 for x in range(10))

# Fonction générateur
def gen_carres(n):
    for i in range(n):
        yield i**2
```

Itération sur Différentes Structures

Listes et Tuples

```
# Énumération
for i, val in enumerate(liste):
    print(f"Index {i}: {val}")

# Parcours parallèle
for x, y in zip(liste1, liste2):
    print(f"{x} correspond à {y}")
```

Dictionnaires

```
# Différentes méthodes
for cle in dict.keys():
    print(cle)

for valeur in dict.values():
    print(valeur)

for cle, val in dict.items():
    print(f"{cle}: {val}")
```

Bonnes Pratiques et Optimisations

À Faire

- Utiliser `enumerate()` au lieu des compteurs manuels
- Préférer les compréhensions aux boucles simples
- Employer `itertools` pour les cas complexes
- Utiliser les générateurs pour les grandes séquences

À Éviter

- Modification pendant l'itération
- Création de listes temporaires inutiles
- Boucles imbriquées inefficaces
- Répétition de calculs coûteux

Exemples Pratiques Avancés

Traitement de Données

```
from itertools import groupby
from operator import itemgetter
# Groupement de données
données = [(1, 'A'), (1, 'B'), (2, 'C')]
for clé, groupe in groupby(données,
                           key=itemgetter(0)):
    print(f"Groupe {clé}:")
    for item in groupe:
        print(f"  {item}")
```

Application Réelle

- Traitement de fichiers volumineux
- Analyse de données en temps réel
- Génération de rapports

Comparaison des Structures de Données Python

Structure	Ordonnée	Modifiable	Doublons	Accès
Liste	Oui	Oui	Oui	Index
Tuple	Oui	Non	Oui	Index
Ensemble	Non	Oui	Non	Valeur
Dictionnaire	Non*	Oui	Non (Clés)	Clé

Note

*Depuis Python 3.7, les dictionnaires préservent l'ordre d'insertion

Conclusion

- Récapitulation des Structures de Données
- Importance dans la Programmation Python
- À venir : Travail avec les Fichiers et Gestion des Exceptions

Manipulation des Fichiers

Opérations de Base

- Lecture de Fichiers
- Écriture dans les Fichiers
- Modes d'Ouverture : 'r', 'w', 'a'
- Utilisation du Gestionnaire de Contexte (with)

Points Importants

- Toujours fermer les fichiers après utilisation
- Gérer les exceptions lors des opérations sur les fichiers
- Utiliser les chemins relatifs et absolus correctement

Exemple de Manipulation de Fichiers

```
# Lecture d'un fichier
with open('mon_fichier.txt', 'r') as f:
    contenu = f.read()
    print(contenu)

# Écriture dans un fichier
with open('sortie.txt', 'w') as f:
    f.write('Bonjour, monde!')

# Ajout à un fichier
with open('log.txt', 'a') as f:
    f.write('Nouvelle entrée\n')
```