DEVENIR DÉVELOPPEUR PYTHON

MODULE 1 : LES FONDATIONS DU LANGAGE

Objectif : Maîtriser la syntaxe et la logique algorithmique de base.

I. Introduction et Environnement

1. Fonctionnement d'un langage interprété vs compilé
2. Installation de Python (Windows, macOS, Linux)
3. Découverte du Shell interactif (REPL)
4. Configuration de l'IDE (VS Code ou PyCharm)
5. Premier script : `print("Hello World")` et exécution

II. Variables et Types Primitifs

1. Le typage dynamique en Python
2. Nombres : `int` , `float` , `complex` et opérations arithmétiques
3. Booléens (`bool`) et logique binaire
4. Chaînes de caractères (`str`) : méthodes de base, concaténation, échappement
5. Formatage de chaînes : f-strings (moderne), `.format()` , `%`
6. Conversion de types (Casting)
7. Saisie utilisateur (`input`)

III. Contrôle du Flux et Logique

1. Opérateurs de comparaison et logiques (`and` , `or` , `not`)
2. Structures conditionnelles (`if` , `elif` , `else`)
3. L'opérateur ternaire
4. Boucles `while` (Tant que) et boucles infinies
5. Boucles `for` (Pour chaque) et itérables
6. Contrôle de boucle : `break` , `continue` , `pass`
7. La fonction `range()` et `enumerate()`
8. Gestion des erreurs de base (`try` , `except`)

IV. Structures de Données Built-in (Le Cœur de Python)

1. **Listes** : Création, indexation, slicing, méthodes (`append` , `pop` , `sort`)
2. **Tuples** : Immutabilité, unpacking, usage
3. **Dictionnaires** : Clés/valeurs, méthodes (`get` , `items` , `keys`), performance (Hash tables)
4. **Sets (Ensembles)** : Unicité, opérations (union, intersection, différence)
5. Copie superficielle vs Copie profonde (`shallow copy` vs `deepcopy`)

V. Fonctions et Modularité

1. Définition (`def`) et appel de fonctions
2. Paramètres positionnels vs nommés (keyword arguments)
3. Valeurs par défaut et pièges des objets mutables
4. Arguments arbitraires (`*args` et `**kwargs`)

5. Scope : Variables locales, globales et `nonlocal`
6. Docstrings et documentation
7. Fonctions Lambda (anonymes)

C'est parti pour le premier véritable cours technique.

Ce module est le plus important. Si votre environnement est mal configuré, rien ne fonctionnera et vous serez frustré avant même de commencer. Prenez le temps de bien lire chaque étape.

MODULE 1 : INTRODUCTION ET ENVIRONNEMENT

1. Fonctionnement d'un langage : Interprété vs Compilé

Avant d'installer quoi que ce soit, vous devez comprendre comment Python parle à votre ordinateur (le processeur). Le processeur ne comprend que des `0` et des `1` . Python est de l'anglais. Il faut donc un **traducteur**.

Il existe deux types de traducteurs en informatique :

A. Le Langage Compilé (Exemple : C, C++)

Imaginez que vous avez un livre écrit en français à traduire en chinois.

- **Le processus :** Le traducteur prend tout le livre, s'enferme dans son bureau, traduit l'intégralité, et vous donne un nouveau livre entièrement en chinois.
- **Avantage :** La lecture du livre final est très rapide (l'ordinateur exécute vite).
- **Inconvénient :** Si vous voulez changer une seule phrase à la page 10, le traducteur doit tout retraiter et réimprimer le livre.
- *En code :* On écrit le code $\rightarrow$ On compile (création d'un `.exe`) $\rightarrow$ On exécute.

B. Le Langage Interprété (Exemple : Python, JavaScript)

Imaginez un interprète simultané à l'ONU.

- **Le processus :** L'interprète lit votre phrase, la traduit immédiatement, l'ordinateur l'exécute, puis il passe à la phrase suivante.
- **Avantage :** C'est flexible ! On peut modifier le code et voir le résultat instantanément. Si ça plante à la ligne 10, on s'arrête à la ligne 10.
- **Inconvénient :** C'est un peu plus lent à l'exécution qu'un langage compilé (car il traduit "en direct").
- *En code :* On écrit le code $\rightarrow$ On lance Python $\rightarrow$ Python lit et exécute ligne par ligne.

Conclusion : Python est **interprété**. C'est pour cela qu'il est idéal pour apprendre : le retour est immédiat.

---

2. Installation de Python

Python n'est pas un simple logiciel comme Word. C'est un interpréteur qui s'installe dans le système.

Sur Windows (Attention danger !)

C'est ici que 50% des débutants échouent. Suivez ceci à la lettre.

1. Allez sur python.org/downloads.
2. Téléchargez la dernière version (bouton jaune "Download Python 3.x.x").
3. Lancez l'installateur.
4. **STOP ! NE CLIQUEZ PAS ENCORE SUR INSTALLER.**
5. En bas de la fenêtre, **COCHEZ LA CASE** : ☑ Add python.exe to PATH.
    - *Pourquoi ?* Le "PATH" est une liste d'adresses que Windows connaît. Si vous ne cochez pas ça, quand vous taperez `python` dans le terminal, Windows dira "C'est qui lui ? Je ne le connais pas".
6. Cliquez sur **"Install Now"**.

Sur macOS

Les Mac ont souvent une version de Python préinstallée, mais elle est vieille ou réservée au système. N'y touchez pas.

1. Allez sur python.org/downloads.
2. Téléchargez le "macOS 64-bit universal2 installer".
3. Installez comme une application classique.
4. Sur Mac, la commande sera souvent `python3` au lieu de `python` .

Sur Linux (Ubuntu/Debian)

C'est souvent déjà là. Pour être sûr d'être à jour : Ouvrez votre terminal et tapez :

```
sudo apt update
sudo apt install python3 python3-pip
```

Vérification (Le Test de Vérité)

1. Ouvrez votre terminal (Invite de commande / PowerShell / Terminal).
2. Tapez : `python --version` (ou `python3 --version`).
3. Si vous voyez `Python 3.12.x` , **BRAVO**, c'est installé.

---

3. Découverte du Shell Interactif (REPL)

Le **REPL** (Read-Eval-Print Loop) est un mode "conversationnel". Vous parlez à Python, il répond, puis il attend la suite. C'est parfait pour tester une idée en 2 secondes.

Comment l'utiliser ?

1. Dans votre terminal, tapez juste `python` (ou `python3`).
2. Vous verrez apparaître trois chevrons : `>>>` . Cela signifie : *"Je suis prêt, donne-moi une instruction"*.

Exemples à tester :

Mathématiques :

```
>>> 5 + 5
10
>>> 10 / 2
5.0
```

Texte :

```
>>> print("Je suis un hacker")
Je suis un hacker
```

Le piège du REPL (Comment sortir ?) : Beaucoup de débutants restent coincés ici. Le REPL n'est pas le terminal système. `cd` ou `ls` ne marchent pas ici. Pour quitter et revenir au terminal normal :

- Tapez `exit()` puis Entrée.
- Ou faites `Ctrl + Z` (Windows) / `Ctrl + D` (Mac).

---

4. Configuration de l'IDE (VS Code)

Le REPL, c'est bien pour tester `1 + 1` . Mais pour écrire des programmes, il faut enregistrer le code dans des fichiers. Pour cela, on utilise un **IDE** (Environnement de Développement Intégré).

Nous allons utiliser **Visual Studio Code (VS Code)**. C'est le standard de l'industrie : gratuit, léger et puissant.

1. Installation :

- Téléchargez-le sur code.visualstudio.com.
- Installez-le (les options par défaut sont très bien).

2. Configuration pour Python (Indispensable) : VS Code est un éditeur de texte générique. Il faut lui "apprendre" le Python.

1. Ouvrez VS Code.
2. À gauche, cliquez sur l'icône des **Extensions** (les 4 carrés) ou `Ctrl+Shift+X` .
3. Dans la barre de recherche, tapez `Python` .
4. Choisissez la première (celle créée par **Microsoft**, avec des millions de téléchargements).
5. Cliquez sur **Install**.

Ce que ça change : Désormais, quand vous écrirez du Python, VS Code mettra des couleurs (coloration syntaxique), vous suggérera la fin des mots (autocomplétion) et soulignera vos erreurs en rouge.

---

5. Premier script : "Hello World"

C'est une tradition en informatique depuis les années 70. Le premier programme qu'on écrit sert juste à vérifier que tout fonctionne.

Étape A : Préparer le terrain

Ne mettez pas votre code n'importe où. Soyez organisé.

1. Créez un dossier `FormationPython` sur votre Bureau ou dans Documents.
2. Dans VS Code, faites **File > Open Folder** (Fichier > Ouvrir le dossier).
3. Sélectionnez votre dossier `FormationPython` .

Étape B : Créer le fichier

1. Dans la barre latérale gauche de VS Code (l'explorateur), faites un clic droit dans le vide.
2. Choisissez "New File" (Nouveau fichier).
3. Nommez-le `main.py` .
    - **Très important :** L'extension `.py` dit à l'ordinateur "Ceci est un script Python". Si vous l'appelez `main.txt` , ça ne marchera pas.

Étape C : Écrire le code

Dans le grand espace vide à droite, écrivez :

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

Analyse du code :

- `print` : C'est une **fonction**. Un ordre qu'on donne à Python ("Affiche quelque chose").
- `()` : Les parenthèses contiennent ce qu'on veut afficher.
- `""` : Les guillemets disent que `Hello World` est du **texte** (une chaîne de caractères), pas une commande mathématique.

Étape D : Exécuter (Le moment de vérité)

Il y a deux façons de lancer votre script :

Méthode 1 : Le bouton Play (Facile) En haut à droite de VS Code, il y a un petit triangle "Play" ▷. Cliquez dessus. Un terminal va s'ouvrir en bas de l'écran et vous devriez voir :

```
Hello World
```

Méthode 2 : Le Terminal (Professionnel) Les pros utilisent le terminal intégré.

1. Dans VS Code, faites `Ctrl + ù` (ou Terminal > New Terminal).
2. Vous êtes déjà dans le bon dossier. Tapez :

    ```
    python main.py
    ```

3. Entrée. Vous verrez `Hello World` .

En cas d'erreur fréquente

- **Erreur :** `SyntaxError: Missing parentheses in call to 'print'`
  - *Cause :* Vous utilisez Python 3 mais vous avez écrit `print "Hello"` (syntaxe de vieux Python 2). Mettez des parenthèses !
- **Erreur :** `Python was not found...`
  - *Cause :* Vous avez oublié de cocher "Add to PATH" à l'installation. Désinstallez et réinstallez Python en cochant la case.
- **Erreur :** Rien ne se passe.
  - *Cause :* Avez-vous sauvegardé le fichier ? (`Ctrl + S`). Dans VS Code, un rond blanc à côté du nom du fichier veut dire "non sauvegardé".

Bravo ! Vous avez écrit et exécuté votre premier programme. Vous êtes officiellement prêt à apprendre le langage.

Voici la suite logique : **Le Module 2**.

Maintenant que votre environnement est prêt, nous allons apprendre à manipuler la matière première de la programmation : **les Données**.

MODULE 2 : VARIABLES ET TYPES PRIMITIFS

1. Le Typage Dynamique et les Variables

Une **variable** est comme une étiquette que vous collez sur une boîte pour stocker une information. Contrairement à des langages comme le C ou Java, Python est **dynamiquement typé**. Cela signifie que vous n'avez pas besoin de dire à l'ordinateur "Cette boîte contiendra un nombre". Python le devine tout seul.

Créer une variable (L'assignation)

Le signe `=` ne veut pas dire "égal" (mathématique), il veut dire **"met ce qu'il y a à droite dans la variable de gauche"**.

```
# Syntaxe : nom_variable = valeur

age = 25           # Python devine que c'est un Entier
nom = "Sophie"     # Python devine que c'est du Texte
prix = 19.99       # Python devine que c'est un Nombre à virgule
est_etudiant = True # Python devine que c'est un Booléen (Vrai/Faux)

print(age)
print(nom)
```

Le concept de "Dynamique"

Vous pouvez changer le contenu et le *type* d'une variable n'importe quand (même si c'est déconseillé pour la clarté).

```
x = 10          # x est un entier
print(x)

x = "Banane"    # x devient une chaîne de caractères (Aucune erreur !)
print(x)
```

> ** Règle d'or du nommage (Snake Case) 🤫* En Python, on écrit les variables en minuscules avec des underscores.
>
> - Bien : `mon_super_score`, `vitesse_lumiere`
> - Pas bien : `monSuperScore` (c'est du Java), `MonScore` (c'est une Classe), `vitesse` (trop vague).

---

2. Les Nombres (`int` , `float` , `complex`)

Python est très fort en maths. Il gère trois types de nombres principaux.

A. Les Types

1. `int` **(Integer / Entier)** : Nombres sans virgule (ex: `1` , `-5` , `42`). Ils ont une précision illimitée en Python (vous pouvez stocker un nombre avec 1000 zéros).
2. `float` **(Floating point / Flottant)** : Nombres à virgule (ex: `3.14` , `-0.01`).
   - *Attention :* On utilise un **point** `.` , pas une virgule `,` .
3. `complex` : Nombres complexes utilisés en sciences (ex: `3 + 4j`). *Rarement utilisé par les débutants.*

B. Les Opérations Mathématiques

Ouvrez votre REPL ou VS Code et testez :

```
a = 10
b = 3

print(a + b)  # Addition (13)
print(a - b)  # Soustraction (7)
print(a * b)  # Multiplication (30)
print(a / b)  # Division classique (3.3333...) -> Donne toujours un float !

# Les opérateurs spéciaux (À connaître absolument)

print(a // b) # Division Entière (3) -> Coupe la partie décimale
print(a % b)  # Modulo (1) -> Le "Reste" de la division (10 = 3*3 + 1)
print(a ** b) # Puissance (1000) -> 10 puissance 3
```

> **Astuce :** Le Modulo `%` est très utile pour savoir si un nombre est pair. Si `nombre % 2` vaut `0`, c'est pair. Sinon, c'est impair.

---

3. Les Booléens (`bool`)

C'est la base de toute logique informatique. Un booléen ne peut avoir que deux valeurs :

- `True` (Vrai, Allumé, 1)
- `False` (Faux, Éteint, 0)

Attention à la majuscule : `true` ne marche pas, il faut `True` .

Logique Binaire (Opérateurs de comparaison)

```
age = 18

print(age > 10)   # True (Strictement plus grand)
print(age >= 18)  # True (Plus grand ou égal)
print(age < 5)    # False
print(age == 18)  # True (Est égal à... Notez le double égal ==)
print(age != 20)  # True (Est différent de...)
```

---

4. Les Chaînes de Caractères (`str`)

Du texte, tout simplement. On l'entoure de guillemets simples `'` ou doubles `"` .

```
texte1 = "Bonjour"
texte2 = 'Le monde'
```

Méthodes de base (Les outils intégrés)

Les chaînes sont des objets, elles ont des "pouvoirs" (méthodes) attachés.

```
message = "  python est GENIAL  "

print(message.upper())      # "  PYTHON EST GENIAL  " (Majuscule)
print(message.lower())      # "  python est genial  " (Minuscule)
print(message.strip())      # "python est GENIAL" (Enlève les espaces
inutiles autour)
print(message.replace("GENIAL", "nul")) # Remplace un morceau
print(len(message))         # 21 (Compte le nombre de caractères, espaces
inclus)
```

L'Échappement

Comment écrire : *L'avion* si on utilise des apostrophes ?

```
# Erreur :
# phrase = 'L'avion'  -> Python croit que la chaîne finit après L

# Solution 1 : Utiliser des guillemets doubles
phrase = "L'avion"

# Solution 2 : Le caractère d'échappement antislash \
phrase = 'L\'avion'
```

---

5. Le Formatage de Chaînes (Indispensable)

Comment insérer une variable au milieu d'un texte ?

La méthode Moderne : f-strings (Python 3.6+)

C'est celle que vous devez utiliser tout le temps. Mettez un petit `f` devant les guillemets.

```
nom = "Alice"
age = 30

# Regardez la magie des accolades {}
phrase = f"Bonjour, je m'appelle {nom} et j'ai {age} ans."
print(phrase)
# Résultat : Bonjour, je m'appelle Alice et j'ai 30 ans.
```

Les vieilles méthodes (Pour comprendre le vieux code)

- `.format()` : `"Je suis {}".format(nom)`
- `%` (Style C) : `"Je suis %s" % nom`

---

6. La Conversion de Types (Casting)

C'est une source d'erreur fréquente. Python ne mélange pas les torchons et les serviettes.

```
chiffre = 5
texte = "10"

# print(chiffre + texte)
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Il faut transformer (caster) l'un vers l'autre.

- `int()` : Force en Entier.

- `float()` : Force en Flottant.
- `str()` : Force en Texte.

Correction de l'exemple :

```
print(chiffre + int(texte))  # 5 + 10 = 15 (Maths)
print(str(chiffre) + texte)  # "5" + "10" = "510" (Concaténation de texte)
```

---

7. Interaction Utilisateur (`input`)

Pour rendre vos programmes interactifs.

> **LE PIÈGE DU DÉBUTANT :** La fonction `input()` renvoie **TOUJOURS du texte (`str`)**, même si l'utilisateur tape un chiffre.

Exemple : Un mini calculateur d'âge

Créez un fichier `age.py` et testez ceci :

```
# 1. On demande l'année de naissance
annee_naissance = input("En quelle année es-tu né ? ")

# À ce stade, si je tape 1990, annee_naissance vaut "1990" (Texte)

# 2. On calcule l'âge
# age = 2023 - annee_naissance  <-- ERREUR ! On ne peut pas faire 2023 -
"Texte"

# 3. La correction : On convertit (cast) le texte en entier
annee_convertie = int(annee_naissance)

age = 2025 - annee_convertie

# 4. On affiche le résultat avec une f-string
print(f"Tu as (ou auras) {age} ans cette année.")
```

Résumé compact pour vos fiches :

Type	Exemple	Note
`int`	`42`	Entier
`float`	`3.14`	Virgule avec un point
`bool`	`True`	Majuscule obligatoire
`str`	`"Salut"`	Entre guillemets
`input()`	`x = input()`	Renvoie toujours un `str`

Type	Exemple	Note
`int("5")`	`5`	Convertit le texte en nombre

Vous maîtrisez maintenant les briques élémentaires. Le prochain niveau, c'est la **Logique** (Si... Alors...).

MODULE 3 : CONTRÔLE DU FLUX ET LOGIQUE

1. Opérateurs de comparaison et logiques

Pour prendre une décision, l'ordinateur doit répondre à une question par OUI (`True`) ou NON (`False`).

Les Opérateurs Logiques (`and` , `or` , `not`)

Imaginez un système de sécurité pour entrer dans une boîte de nuit.

- `and` **(ET)** : Les DEUX conditions doivent être vraies.

```
age = 20
a_billet = True

# Est-ce qu'il a l'âge ET le billet ?
peut_entrer = (age >= 18) and (a_billet == True)
print(peut_entrer) # True
```

- `or` **(OU)** : AU MOINS UNE des conditions doit être vraie.

```
est_vip = False
connait_le_patron = True

# Est-ce qu'il est VIP OU qu'il connaît le patron ?
entree_speciale = est_vip or connait_le_patron
print(entree_speciale) # True
```

- `not` **(NON)** : Inverse le résultat.

```
est_fatigue = True
print(not est_fatigue) # False
```

---

2. Structures conditionnelles (`if` , `elif` , `else`)

C'est l'aiguillage du train.

Règle Absolue : L'Indentation

En Python, il n'y a pas d'accolades `{}` comme en C ou Java. On utilise **l'indentation** (le décalage vers la droite).

- Tout ce qui est décalé (4 espaces ou 1 tabulation) fait partie du bloc.
- Si vous oubliez d'indenter, Python vous criera dessus (`IndentationError`).

La structure complète

```
note = 14

if note >= 18:
    print("Excellent !")         # Ce bloc s'exécute si condition 1 est
Vraie
    print("Tu es un génie.")     # Toujours dans le bloc "if"

elif note >= 10:                 # Sinon, Si...
    print("Passable.")           # Ce bloc s'exécute si condition 1 Fausse
ET condition 2 Vraie

else:                            # Sinon (tous les autres cas)
    print("Recalé.")             # Ce bloc s'exécute si tout le reste est
Faux

print("Fin du bulletin.")        # Ce code n'est plus indenté, il s'exécute
TOUJOURS.
```

---

3. L'Opérateur Ternaire (Le raccourci Pro)

Parfois, un `if/else` prend trop de place pour une simple assignation de variable. On peut le faire en une seule ligne.

Version longue (Classique) :

```
age = 15
status = ""

if age >= 18:
    status = "Majeur"
else:
    status = "Mineur"
```

Version courte (Ternaire) :

```
# Syntaxe : VALEUR_SI_VRAI if CONDITION else VALEUR_SI_FAUX
```

```
status = "Majeur" if age >= 18 else "Mineur"
```

C'est très "Pythonic" (élégant).

---

4. Boucles `while` (Tant que...)

On utilise `while` quand **on ne sait pas à l'avance combien de fois** on va répéter l'action (ex: "Tant que l'utilisateur n'a pas deviné le mot de passe").

```
compteur = 0

while compteur < 5:
    print(f"Le compteur est à {compteur}")
    compteur = compteur + 1  # INDISPENSABLE : Sinon boucle infinie !
```

Danger : La Boucle Infinie

Si vous oubliez d'incrémenter `compteur`, la condition `0 < 5` sera toujours vraie. Votre ordinateur va chauffer et le programme ne s'arrêtera jamais.

- **Solution d'urgence :** Faites `Ctrl + C` dans le terminal pour tuer le programme.

---

5. Boucles `for` (Pour chaque...)

C'est la boucle la plus utilisée en Python. Contrairement au C (où on compte `i=0; i<10; i++`), le `for` de Python **parcourt une collection** (liste, texte, etc.).

Parcourir une liste :

```
fruits = ["Pomme", "Banane", "Cerise"]

for fruit in fruits:
    # À chaque tour, la variable 'fruit' prend la valeur suivante
    print(f"J'aime la {fruit}")
```

Parcourir du texte :

```
nom = "Python"
for lettre in nom:
    print(lettre.upper())
# Affiche P, puis Y, puis T...
```

---

6. Contrôle de boucle : `break` , `continue` , `pass`

Parfois, on veut modifier le comportement naturel de la boucle.

- `break` : **STOP TOUT !** On sort immédiatement de la boucle.

```
# Chercher un intrus
panier = ["Pomme", "Poire", "Ver de terre", "Banane"]

for item in panier:
    if item == "Ver de terre":
        print("Trouvé ! J'arrête de chercher.")
        break # On ne regarde pas "Banane", on sort direct.
    print(f"Miam, une {item}")
```

- `continue` : **PASSE CE TOUR**, mais continue la boucle.

```
# Afficher seulement les nombres pairs
for i in [1, 2, 3, 4]:
    if i % 2 != 0: # Si c'est impair
        continue   # On zappe ce tour, on ne print pas
    print(i)
```

- `pass` : **JE NE FAIS RIEN**. Python déteste les blocs vides. Si vous créez un `if` ou une boucle mais que vous n'avez pas encore écrit le code dedans, utilisez `pass` pour éviter une erreur.

```
if age < 10:
    pass  # TODO: Gérer ça plus tard
```

---

7. `range()` et `enumerate()` (Les super-outils)

`range(début, fin, pas)`

Génère une suite de nombres. **ATTENTION :** La borne de fin est **exclue** !

```
# De 0 à 4 (5 est exclu)
for i in range(5):
    print(i) # 0, 1, 2, 3, 4

# De 2 à 8, par pas de 2
for i in range(2, 9, 2):
    print(i) # 2, 4, 6, 8
```

`enumerate()`

Souvent, on veut la valeur **ET** sa position (index) dans la liste.

La mauvaise méthode (style C/Java) :

```
fruits = ["A", "B", "C"]
for i in range(len(fruits)):
    print(i, fruits[i]) # Lourd et moche
```

La méthode Python Pro (`enumerate`) :

```
fruits = ["Pomme", "Poire", "Banane"]

for index, valeur in enumerate(fruits):
    print(f"Le fruit n°{index} est {valeur}")
```

---

8. Gestion des erreurs (`try` , `except`)

Un bon programme ne plante pas (ne "crash" pas) quand l'utilisateur fait n'importe quoi.

Imaginez une calculatrice :

```
try:
    x = int(input("Entrez un nombre : "))
    y = int(input("Entrez le diviseur : "))
    resultat = x / y
    print(f"Résultat : {resultat}")

except ValueError:
    # Se déclenche si l'utilisateur tape "a" au lieu de "12"
    print("Erreur : Vous devez entrer des chiffres !")

except ZeroDivisionError:
    # Se déclenche si l'utilisateur tape 0 pour le diviseur
    print("Erreur : Division par zéro impossible, trou noir imminent.")

except Exception as e:
    # Attrape TOUTES les autres erreurs (filet de sécurité)
    print(f"Une erreur inconnue est survenue : {e}")

else:
    # S'exécute SI tout s'est bien passé
    print("Bravo, calcul réussi.")

finally:
```

```
    # S'exécute DANS TOUS LES CAS (erreur ou pas)
    print("Fin du programme.")
```

Résumé pour vos fiches

Mot-clé	Action
`if`	Si condition vraie
`elif`	Sinon si...
`else`	Sinon (par défaut)
`while`	Répète tant que c'est vrai
`for`	Parcourt une collection
`break`	Stoppe la boucle
`continue`	Passe au tour suivant
`range(5)`	Crée une séquence 0, 1, 2, 3, 4
`try/except`	Gère les crashs

Prochaine étape : **Les Structures de Données** (Listes, Dictionnaires...), là où on stocke vraiment l'information.

C'est ici que Python brille par rapport à des langages comme le C ou le Java. Python possède des **structures de données intégrées** (Built-in Data Structures) incroyablement puissantes et faciles à utiliser.

Comprendre ces 4 structures, c'est comprendre 80% de la programmation Python quotidienne.

MODULE 4 : STRUCTURES DE DONNÉES (Le Cœur)

1. Les Listes (`list`) : Le couteau suisse

Une liste est une **collection ordonnée** et **mutable** (modifiable). C'est l'équivalent d'un tableau dynamique. Elle peut contenir n'importe quoi (des entiers, des chaînes, et même d'autres listes).

A. Création et Accès

```
# Création avec des crochets []
panier = ["Pomme", "Banane", "Cerise", 42, 3.14]

# Indexation (Ça commence à 0 !)
print(panier[0])  # "Pomme"
print(panier[2])  # "Cerise"
```

```
# La magie de l'indexation négative (Spécifique à Python)
# -1 veut dire "le dernier", -2 "l'avant-dernier"
print(panier[-1]) # 3.14
```

B. Le Slicing (Découpage chirurgical)

La syntaxe est `liste[début : fin : pas]` .

- **Rappel :** La borne de fin est toujours **exclue**.

```
nombres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(nombres[0:3])   # [0, 1, 2] (De l'index 0 à 2)
print(nombres[:3])    # [0, 1, 2] (Si on omet le début, ça commence à 0)
print(nombres[5:])    # [5, 6, 7, 8, 9] (De 5 jusqu'à la fin)
print(nombres[::2])   # [0, 2, 4, 6, 8] (Tout, par pas de 2)

# L'astuce d'entretien d'embauche : Inverser une liste
print(nombres[::-1])  # [9, 8, 7..., 0]
```

C. Méthodes essentielles

Les listes sont des objets, elles ont des méthodes.

```
todo = ["Manger", "Dormir"]

# Ajouter
todo.append("Coder")       # Ajoute à la FIN -> ["Manger", "Dormir",
"Coder"]
todo.insert(0, "Réveil")   # Insère à l'index 0 -> ["Réveil", "Manger"...]

# Retirer
item = todo.pop()          # Retire et RENVOIE le dernier élément ("Coder")
todo.remove("Manger")      # Cherche "Manger" et le supprime (Pas d'index
ici)

# Trier
chiffres = [5, 1, 8, 3]
chiffres.sort()            # Modifie la liste EN PLACE -> [1, 3, 5, 8]
# vs
nouveau = sorted(chiffres) # Crée une NOUVELLE liste triée, ne touche pas
l'originale
```

---

2. Les Tuples (`tuple`) : Le coffre-fort

Un tuple est comme une liste, mais **IMMUABLE**. Une fois créé, on ne peut plus le modifier (ni ajouter, ni retirer, ni changer une valeur).

Pourquoi utiliser ça ?

1. **Sécurité :** Vous êtes sûr que les données ne bougeront pas (ex: coordonnées GPS, jours de la semaine).
2. **Performance :** C'est plus léger en mémoire qu'une liste.
3. **Dictionnaire :** Un tuple peut servir de clé dans un dictionnaire (pas une liste).

A. Création

```
# Avec des parenthèses ()
point = (10, 20)

# Sans parenthèses (Python comprend)
coordonnees = 48.85, 2.35

# Piège du tuple unique :
solo = (5)   # C'est un entier (int)
vrai_solo = (5,) # C'est un tuple (notez la virgule)
```

B. Unpacking (Déballage)

C'est une syntaxe très élégante pour extraire des valeurs.

```
user = ("John", "Doe", 30)

prenom, nom, age = user  # Hop ! 3 variables créées d'un coup.

print(prenom) # "John"

# Échanger deux variables sans variable temporaire (Magie Python)
a = 5
b = 10
a, b = b, a   # a vaut 10, b vaut 5
```

---

3. Les Dictionnaires (`dict`) : La base de données

C'est la structure la plus importante pour le Web et la Data. C'est un système de **Clé : Valeur**. Dans d'autres langages, on appelle ça *Hash Map*, *Map* ou *Tableau Associatif*.

- L'accès à une valeur est quasi-instantané $O(1)$, quelle que soit la taille du dictionnaire.

A. Création et Accès

```python
# Clé : Valeur
etudiant = {
    "nom": "Dupont",
    "age": 21,
    "notes": [12, 15, 9]  # On peut mettre une liste dans un dico
}

# Accès direct
print(etudiant["nom"])  # "Dupont"

# Le Danger : Accéder à une clé qui n'existe pas
# print(etudiant["adresse"])  -> KeyError: 'adresse' (Crash du programme)

# La solution Pro : .get()
print(etudiant.get("adresse"))         # Renvoie None (pas d'erreur)
print(etudiant.get("adresse", "Inconnu")) # Renvoie "Inconnu" par défaut
```

B. Modification

```python
etudiant["age"] = 22           # Modifie
etudiant["ville"] = "Paris"    # Ajoute une nouvelle paire
del etudiant["notes"]          # Supprime la clé
```

C. Itération (Boucler sur un dico)

```python
# Parcourir les clés et valeurs
for cle, valeur in etudiant.items():
    print(f"La clé {cle} contient {valeur}")

# Juste les clés : etudiant.keys()
# Juste les valeurs : etudiant.values()
```

---

4. Les Sets (`set`) : Les mathématiques

Un Set (Ensemble) est une collection **non ordonnée** d'éléments **uniques**.

- Pas de doublons.
- Pas d'index (on ne peut pas dire `set[0]`).

A. Usage principal : Nettoyer les doublons

```python
liste_sale = [1, 2, 2, 3, 3, 3, 4]

# On convertit en Set -> Doublons éliminés -> On remet en Liste
```

```
liste_propre = list(set(liste_sale))

print(liste_propre) # [1, 2, 3, 4]
```

B. Opérations mathématiques

Très puissant pour comparer des groupes de données.

```
devs_python = {"Alice", "Bob", "Charlie"}
devs_java = {"Bob", "David", "Eve"}

# Intersection (&) : Qui fait les DEUX ?
print(devs_python & devs_java) # {'Bob'}

# Union (|) : Tous les développeurs (sans doublons)
print(devs_python | devs_java) # {'Alice', 'Bob', 'Charlie', 'David',
'Eve'}

# Différence (-) : Qui fait du Python MAIS PAS de Java ?
print(devs_python - devs_java) # {'Alice', 'Charlie'}
```

---

5. Copie Superficielle vs Profonde (Le piège mortel)

En Python, les variables ne sont que des **étiquettes** (pointeurs) vers des objets en mémoire.

A. Le problème de l'assignation simple

```
a = [1, 2, 3]
b = a          # b est juste une AUTRE étiquette sur la MÊME liste

b.append(4)

print(a)       # [1, 2, 3, 4]  <-- A est modifié aussi !
```

B. La Copie Superficielle (Shallow Copy)

Pour créer une vraie copie indépendante d'une liste simple.

```
a = [1, 2, 3]
b = a.copy()   # Ou b = a[:]

b.append(4)
print(a)       # [1, 2, 3]     <-- A est sauf !
```

C. La Copie Profonde (Deep Copy)

Si vous avez des listes DANS des listes (imbrication), la copie superficielle ne suffit pas (elle copie la liste principale, mais les sous-listes restent partagées).

```
import copy

liste_complexe = [[1, 2], [3, 4]]

# Copie totale et récursive
liste_independante = copy.deepcopy(liste_complexe)
```

Résumé des Structures

Structure	Syntaxe	Ordonné ?	Mutable ?	Doublons ?	Usage Type
List	`[1, 2]`	Oui	Oui	Oui	Collection d'objets standards
Tuple	`(1, 2)`	Oui	Non	Oui	Coordonnées, constantes
Dict	`{'a': 1}`	(3.7+)	Oui	Clés Uniques	Base de données, JSON, Config
Set	`{1, 2}`	Non	Oui	Non	Maths, supprimer doublons

Vous avez maintenant les outils pour stocker vos données. La prochaine étape sera d'organiser votre code avec les **Fonctions** pour ne pas vous répéter.

C'est ici que l'on passe du statut de "bricoleur de scripts" à celui de **développeur**.

Jusqu'ici, vous écriviez du code séquentiel (ligne 1 à ligne 100). Mais si vous devez faire la même chose 10 fois, vous n'allez pas copier-coller 10 fois votre code. Vous allez créer une **Fonction**.

Le principe clé ici est **DRY : Don't Repeat Yourself** (Ne vous répétez pas).

MODULE 5 : FONCTIONS ET MODULARITÉ

1. Définition et Appel (`def` , `return`)

Une fonction est une mini-usine : on lui donne des ingrédients (arguments), elle fait un travail, et elle renvoie un produit fini (valeur de retour).

A. La Syntaxe

```python
def nom_de_la_fonction(parametre):
    # Bloc de code indenté
    resultat = parametre * 2
    return resultat
```

B. `return` vs `print` (La confusion n°1)

- `print` : Affiche du texte à l'écran pour l'humain. L'ordinateur "oublie" la valeur ensuite.
- `return` : Renvoie la valeur au programme pour qu'elle soit stockée dans une variable. `return` **arrête immédiatement la fonction.**

```python
def addition_inutile(a, b):
    print(a + b) # Affiche, mais renvoie "None"

def addition_utile(a, b):
    return a + b # Renvoie le résultat

# Test
x = addition_inutile(2, 3)
# x vaut None (vide), on ne peut rien faire avec.

y = addition_utile(2, 3)
# y vaut 5. On peut faire y * 2 ensuite.
```

---

2. Paramètres Positionnels vs Nommés

Python est très flexible sur la façon de passer des données.

```python
def presenter(prenom, age, ville):
    print(f"{prenom} a {age} ans et vit à {ville}.")
```

A. Arguments Positionnels (L'ordre compte)

C'est la méthode classique.

```python
presenter("Alice", 25, "Paris")
# Alice a 25 ans et vit à Paris.

presenter(25, "Paris", "Alice")
# 25 a Paris ans et vit à Alice. (Sensé pour Python, absurde pour nous)
```

B. Arguments Nommés (Keyword Arguments)

On précise le nom, l'ordre n'a plus d'importance. C'est **plus lisible**.

```
presenter(age=30, ville="Lyon", prenom="Bob")
# Bob a 30 ans et vit à Lyon.
```

---

3. Valeurs par défaut et le PIÈGE MORTEL

On peut rendre des paramètres optionnels en leur donnant une valeur par défaut.

```
def café(sucre=True, lait=False):
    if sucre: print("Avec sucre")
    if lait: print("Avec lait")

café()            # Utilise les défauts (Sucre oui, Lait non)
café(lait=True)   # Change juste le lait
```

Le Piège des Objets Mutables (Interview Question)

NE JAMAIS utiliser une liste ou un dictionnaire vide comme valeur par défaut.

Le code buggé :

```
def ajouter_item(item, liste=[]):  # DANGER
    liste.append(item)
    return liste

print(ajouter_item("Pomme"))  # ['Pomme'] -> OK
print(ajouter_item("Poire"))  # ['Pomme', 'Poire'] -> QUOI ?
```

Pourquoi ? La liste par défaut est créée **une seule fois** au démarrage du programme, pas à chaque appel. Tout le monde partage la même liste !

La correction (Pattern standard) : Utilisez `None` comme valeur par défaut.

```
def ajouter_item(item, liste=None):
    if liste is None:
        liste = []  # On crée une nouvelle liste fraîche à chaque appel
    liste.append(item)
    return liste
```

---

4. Arguments Arbitraires (`*args` , `**kwargs`)

Parfois, on ne sait pas combien d'arguments on va recevoir (ex: une fonction `somme` qui prend 2, 10 ou 100 nombres).

A. `*args` (Tuple d'arguments positionnels)

Le `*` "aspire" tous les arguments restants dans un Tuple.

```
def somme_tout(*nombres):
    total = 0
    for n in nombres:
        total += n
    return total

print(somme_tout(1, 2, 3, 4)) # 10
```

B. `**kwargs` (Dictionnaire d'arguments nommés)

Le `**` "aspire" tous les arguments nommés dans un Dictionnaire. `kwargs` veut dire *KeyWord Args*.

```
def creer_profil(**infos):
    for cle, valeur in infos.items():
        print(f"{cle}: {valeur}")

creer_profil(nom="Bond", prenom="James", code="007")
# Affiche :
# nom: Bond
# prenom: James
# code: 007
```

---

5. Scope : Portée des Variables (Qui voit quoi ?)

C'est une source fréquente d'erreurs. Une variable n'est pas visible partout.

A. Variable Locale

Une variable créée **DANS** une fonction n'existe **QUE** dans la fonction. Elle meurt quand la fonction se termine.

```
def ma_fonc():
    secret = 42

ma_fonc()
# print(secret) -> NameError: name 'secret' is not defined
```

B. Variable Globale

Une variable créée en dehors des fonctions est visible partout (en lecture seule par défaut).

```
VITESSE_MAX = 80 # Constante globale

def radar(vitesse):
    if vitesse > VITESSE_MAX: # On peut LIRE la globale
        print("Amende !")
```

C. Modifier une Globale (Attention)

Si vous voulez changer une variable globale depuis une fonction, il faut utiliser le mot-clé `global` . (C'est souvent une mauvaise pratique, évitez-le si possible).

```
compteur = 0

def incrementer():
    global compteur
    compteur += 1
```

---

6. Docstrings : La Documentation Pro

Un code sans documentation est un code mort. En Python, on documente DANS la fonction, avec des triples guillemets `"""` .

```
def calcul_surface(longueur, largeur):
    """
    Calcule l'aire d'un rectangle.

    Args:
        longueur (float): La longueur en mètres.
        largeur (float): La largeur en mètres.

    Returns:
        float: L'aire en mètres carrés.
    """
    return longueur * largeur
```

Avantage : Si vous tapez `help(calcul_surface)` dans le terminal, Python affichera votre manuel d'utilisation !

---

7. Fonctions Lambda (Anonymes)

Ce sont des fonctions "jetables", écrites en une seule ligne. On les utilise souvent quand une autre fonction a besoin d'une petite fonction en argument (comme pour un tri).

Syntaxe : `lambda arguments: expression` (Le `return` est implicite).

Exemple classique : Trier une liste complexe

Imaginez une liste de tuples (Nom, Âge). On veut trier par âge.

```
utilisateurs = [("Alice", 30), ("Bob", 20), ("Charlie", 25)]

# Méthode classique (longue)
def get_age(user):
    return user[1]
utilisateurs.sort(key=get_age)

# Méthode Lambda (Pro)
# "Pour chaque x, utilise x[1] comme critère de tri"
utilisateurs.sort(key=lambda x: x[1])

print(utilisateurs)
# [('Bob', 20), ('Charlie', 25), ('Alice', 30)]
```

Résumé Module 5

Concept	Syntaxe / Mot-clé	Rôle
Définition	`def ma_func():`	Crée la fonction
Retour	`return`	Renvoie le résultat et quitte
Défaut	`def f(a=10):`	Paramètre optionnel
Args variables	`*args`	Liste infinie d'arguments (Tuple)
Kwargs variables	`**kwargs`	Liste infinie d'options (Dict)
Scope	`global`	Modifie une variable hors fonction
Doc	`""" Texte """`	Documentation intégrée
Lambda	`lambda x: x+1`	Petite fonction anonyme

FIN
