

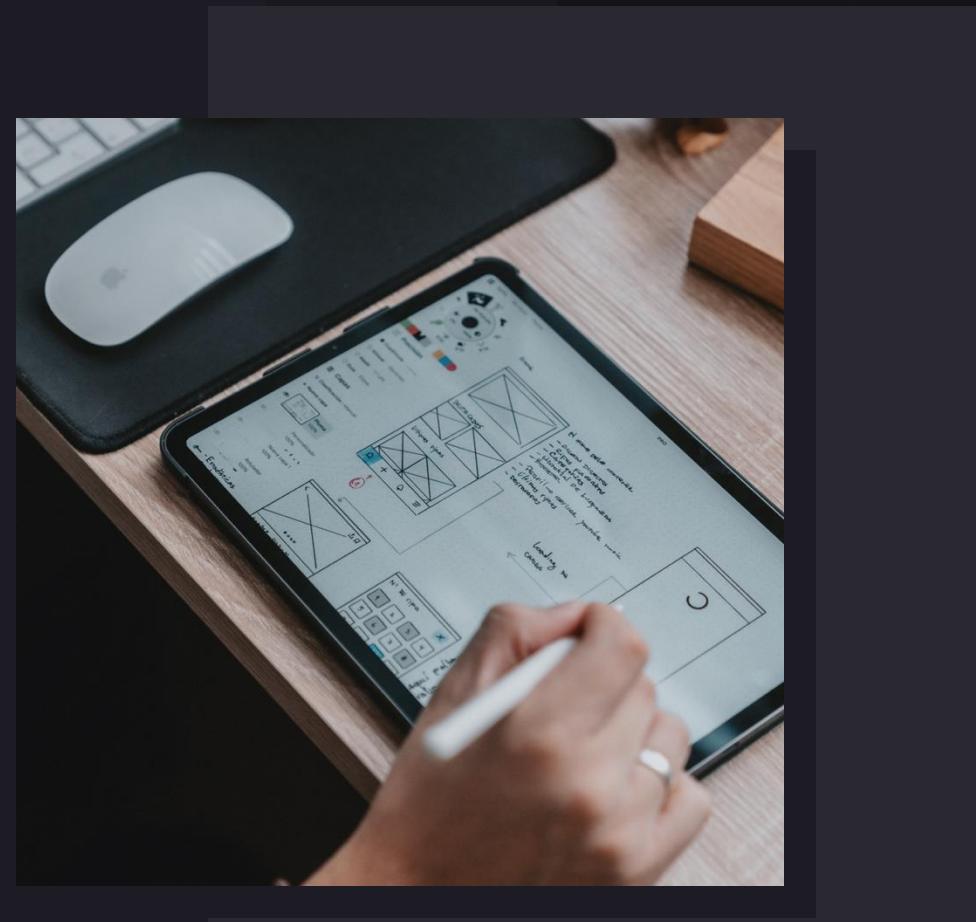


Python

DEVELOPPER AVEC UN LANGAGE PROCEDURAL

Objectifs

- Comprendre les bases de la programmation procédurale (variables, boucles, conditions, fonctions).
- Utiliser un environnement de développement pour écrire et déboguer du code.
- Compiler et exécuter un programme Maitriser les outils de gestion de version





Environnement.



IDE



Interpreter Python



Documentation

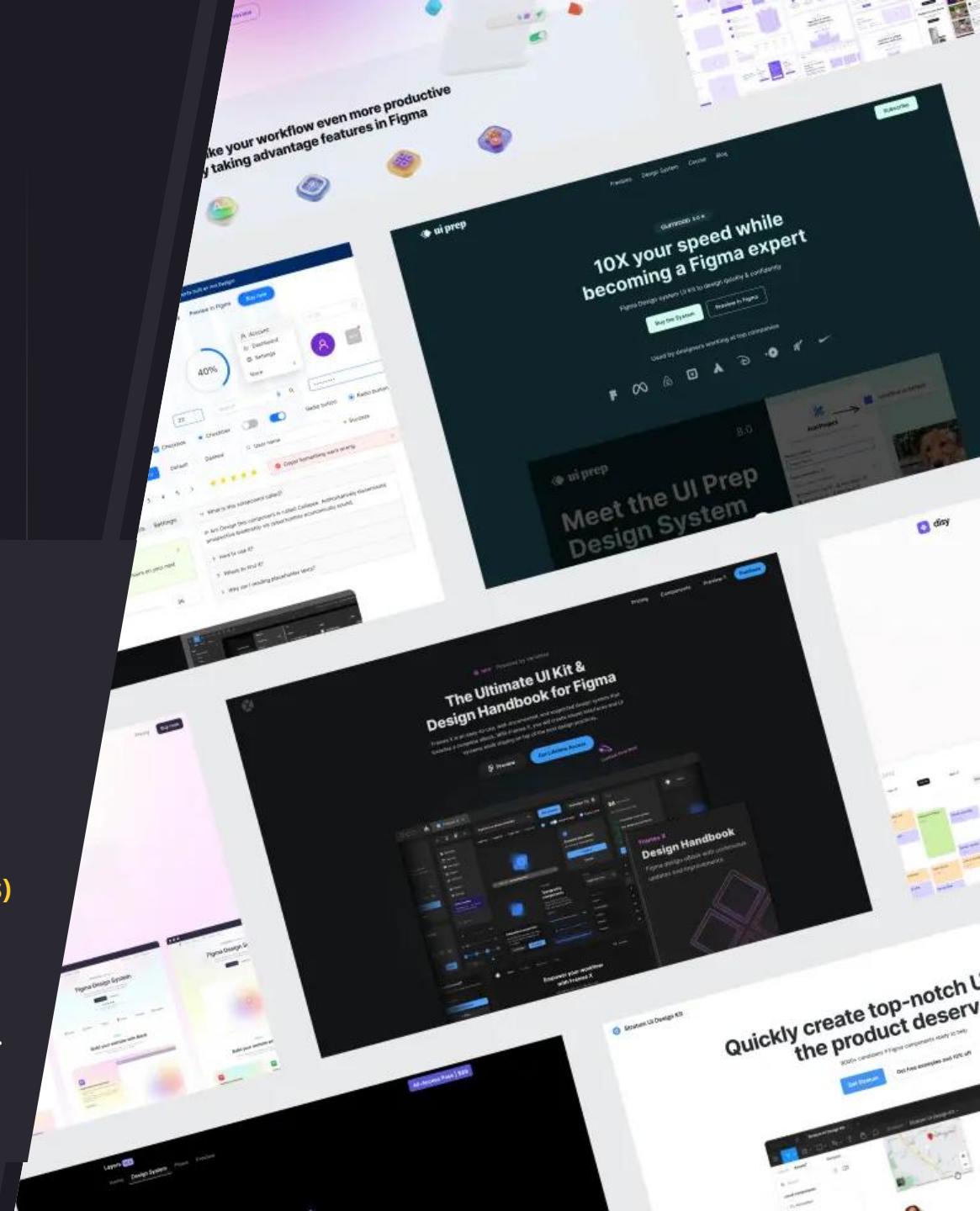
1. Python est un langage interprété

Inclus des gestionnaires de paquets (pour installer des bibliothèques)

Pas d'interpréteur Python = pas de Python exécuté.

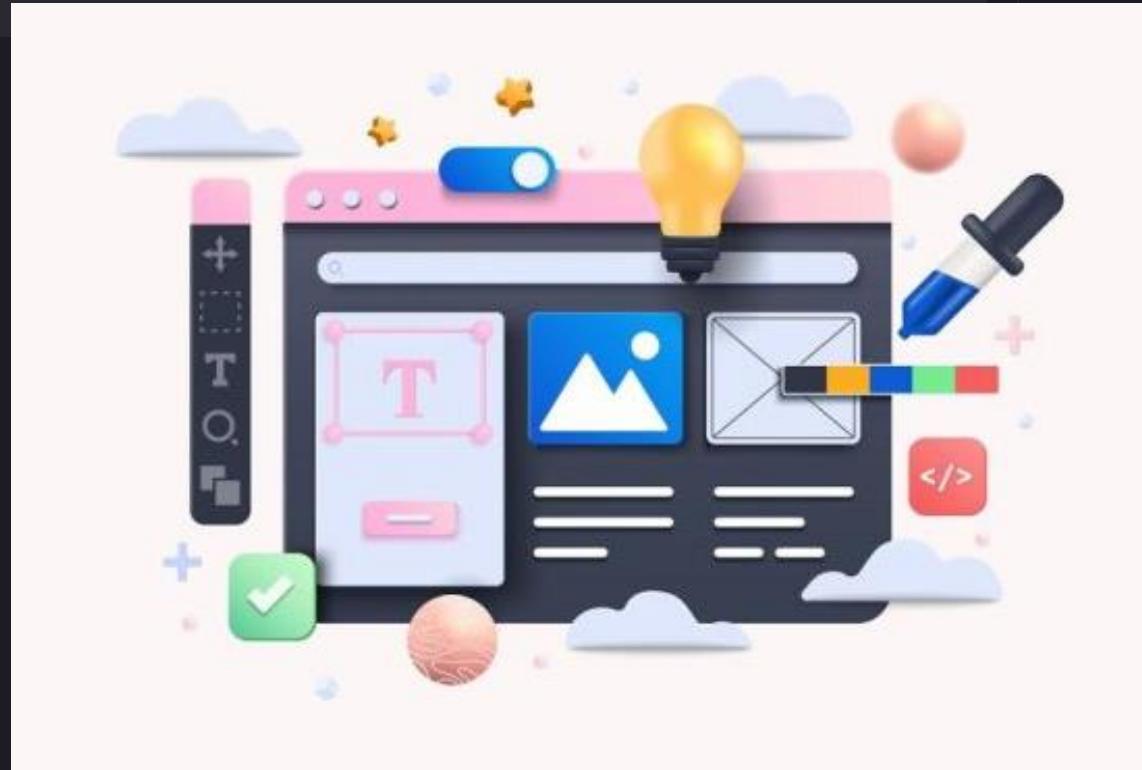
C'est lui qui traduit code Python en instructions que l'ordinateur comprend.

Sur Windows cocher la case « PATH »





Généralités



Commentaires avec le #

Print() pour afficher dans la console

Nécessité de Run le code



```
1 # Variable = A container for a value
2 #           A variable behaves as if
3
4 # Strings|      I
5 first_name = "Bro"
6 food = "pizza"
7 email = "Bro123@fake.com"
8
9 # Integers
10 age = 25
11 quantity = 3
12 num_of_students = 30
13
14 # Float
15 price = 10.99
16 gpa = 3.2
17 distance = 5.5
18
19 # Boolean
20 is_student = True
21 for_sale = False
```

Déclarations variables

En Python, pas besoin de déclarer le type : le langage est dynamique.

- Règles de nommage
 - Commence par une lettre ou _
 - Pas d'espace, pas de caractères spéciaux
 - Sensible à la casse (Age ≠ age)

Il est possible de faire des affectations multiples

```
a, b, c = 1, 2, 3
x = y = 0 # Même valeur
```



```

1 # Typecasting = the process of cor
2 # str(), int(), floa
3
4 name = "Bro Code"
5 age = 25
6 gpa = 3.2
7 is_student = True
8
9 age = str(age)
10
11 age += "1"
12
13 print(age)

```

TypeCasting

Le **typecasting** permet de convertir une variable d'un type à un autre.

Pour vérifier le type d'une variable on utilisera **type(maVar)**

int()	int("10")	→ 10 (entier)
float	float("3.14")	→ 3.14 (flottant)
str()	str(25)	→ "25" (chaîne)
bool()	bool(0), bool("salut")	→ False, True



```
# input() = A function that prompts the user for input  
#           Returns the entered data as a string  
  
name = input("What is your name?: ")  
age = int(input("How old are you?"))  
  
age = age + 1  
  
print(f"Hello {name}!")  
print("HAPPY BIRTHDAY!")  
print(f"You are {age} years old")
```

Input

Pour demander à l'utilisateur **de saisir des informations** depuis le clavier.

- **input()** renvoie toujours **une chaîne de caractères (str)**.
- Utiliser **int()** ou **float()** si besoin de conversion.

⚠ Attention

```
age = int(input("Ton âge ?"))
```

Si l'utilisateur tape du texte non numérique ✖ Erreur !



Exercice 1 : input & variables

```
/*
  1. Ecrire un programme qui demande la longueur puis la
  largeur d'un rectangle et calcule l'aire de ce rectangle puis
  l'affiche
*/
/*
  2. Ecrire un programme qui demande quel article acheter, son
  prix, puis la quantité. On affiche ensuite une phrase
  recapitulative du style « vous avez acheté 9 pizzas pour un
  total de 999€ »
*/
```

Exercice 1 : corrections

```
1 # Exercise 1 Rectangle Area Calc  
2  
3 length = float(input("Enter the length: "))  
4 width = float(input("Enter the width: "))  
5 area = length * width  
6  
7 print(f"The area is: {area}cm²")
```

```
# Exercise 2 Shopping Cart Program
```

```
item = input("What item would you like to buy?: ")  
price = float(input("What is the price?: "))  
quantity = int(input("How many would you like?: "))  
total = price * quantity  
  
print(f"You have bought {quantity} x {item}/s")  
print(f"Your total is: ${total}")
```



```
# if = Do some code only IF something is true  
# Else do something else  
  
age = int(input("Enter your age  
  
if age >= 100:  
    print("You are too old to sign up")  
elif age >= 18:  
    print("You are now signed up")  
elif age < 0:  
    print("You haven't been born yet")  
else:  
    print("You must be 18+ to sign up")  
  
if age >= 18
```

Condition if / else

Pour exécuter différents blocs de code selon des conditions logiques.

- **if**: teste une première condition.
- **elif**: (optionnel) teste d'autres conditions si la précédente est fausse
- **.else** : (optionnel) s'exécute si aucune condition n'est remplie.

⚠️ **Attention à l'indentation**

✓ Indenter le code (généralement avec **4 espaces ou Tab**)



Exercice 2 : input & conditions

```
/*
  1. Ecrire un programme qui demande l'opérateur (« + - * / »)
  2 chiffres, et opère le calcul en fonction de l'opérateur entré
*/
```



```
/*
  2. écrire un programme qui demande à l'utilisateur de saisir
  une note sur 20. Le programme doit ensuite afficher la mention
  associée ET indiquer si la note est valide.*/
*/
```

Coorection exercices 2

```
# Demander la note à l'utilisateur
note = float(input("Entrez votre note sur 20 : "))

# Vérification de la validité de la note
if note < 0 or note > 20:
    print("Erreur : note invalide (doit être entre 0 et 20)")
else:
    # Détermination de la mention selon la note
    if note >= 16:
        print("Mention : Très bien")
    elif note >= 14:
        print("Mention : Bien")
    elif note >= 12:
        print("Mention : Assez bien")
    elif note >= 10:
        print("Mention : Passable")
    else:
        print("Mention : Insuffisant")
```

Python - Logical O

- not

x	notx
False	True
True	False

- and

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

- or

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Operateurs logiques

Les opérateurs logiques permettent de combiner ou inverser des conditions.

and Vrai si les 2 conditions sont vraies

or Vrai si au moins une condition est vraie

not Inverse la valeur d'une condition

⚠ Eviter les if imbriqués

✓ Utilise les opérateurs logiques pour simplifier les conditions longues au lieu de les imbriquer.



```
age = 13
temperature = 20
user_role = "admin"

# print("Positive" if num > 0 else "Nega
# result = "EVEN" if num % 2 == 0 else "
# max_num = a if a > b else b
# min_num = a if a < b else b
# status = "Adult" if age >= 18 else "Ch
# weather = "HOT" if temperature > 20 el
access_level = "Full Access" if user_rol

print(access_level)
```

Condition ternaire

Permet d'écrire une **condition simple en une seule ligne**.

résultat = valeur_si_vrai if condition else valeur_si_faux

C'est une alternative plus compacte à if/else. Pratique pour les affectations rapides ou les affichages simples.

⚠ Ne pas l'utiliser pour des conditions complexes



Methode string

Méthode	Description	Exemple
<code>len()</code>	Longueur de la chaîne	<code>"Python".len()</code> → 6
<code>find(s)</code>	Position de la 1ère occurrence de <code>s</code>	<code>"bonjour".find("o")</code> → 1
<code>rfind(s)</code>	Position de la dernière occurrence de <code>s</code>	<code>"bonjour".rfind("o")</code> → 4
<code>capitalize()</code>	Met la 1re lettre en majuscule	<code>"python".capitalize()</code> → "Python"
<code>upper()</code>	Transforme en majuscules	<code>"hello".upper()</code> → "HELLO"
<code>lower()</code>	Transforme en minuscules	<code>"HELLO".lower()</code> → "hello"
<code>isdigit()</code>	Vérifie si la chaîne ne contient que des chiffres	<code>"123".isdigit()</code> → True
<code>isalpha()</code>	Vérifie si la chaîne ne contient que des lettres	<code>"abc".isalpha()</code> → True
<code>count(s)</code>	Compte le nombre d'occurrences de <code>s</code>	<code>"salut".count("s")</code> → 1
<code>replace(a, b)</code>	Remplace <code>a</code> par <code>b</code> dans la chaîne	<code>"pomme".replace("m", "n")</code> → "ponne"



Exercice 3 : chaines et if

```
/*
  1. Ecrire un programme qui valide l'entrée d'un nom
utilisateur. Il doit contenir moins de 12 caractères, pas
d'espaces et pas de chiffres.
*/
```

Correction Exercice 3

```
username = input("Enter a username: ")

if len(username) > 12:
    print("Your username can't be more than 12 characters")
elif not username.find(" ") == -1:
    print("Your username can't contain spaces")
elif not username.isalpha():
    print("Your username can't contain numbers")
else:
    print(f"Welcome {username}")
```

```
py
# indexing = accessing elements of a
#           [start : end : step]

credit_number = "1234-5678-9012-3456"

last_digits = credit_number[-4:]
print(f"XXXX-XXXX-XXXX-{last_digits}")
```

Syntaxe	Résultat
texte[0:4]	"Pyth"
texte[::-2]	"Pto" (1 sur 2)
texte[1:]	"ython" (à partir de l'index 1)
texte[:3]	"Pyt" (jusqu'à l'index 2 inclus)
texte[::-1]	"nohtyP" (chaîne inversée)

Indexation String

maVar [début : fin : step]

Les index commencent à 0

✖ Une erreur se produit si l'index dépasse la taille de la chaîne

- texte[a:b] inclut l'indice a, mais exclut b
- L'indexation négative permet de partir de la fin
- Le step (pas) est optionnel et sert à sauter des caractères



```

py
# indexing = accessing elements of a list
# [start : end : step]

credit_number = "1234-5678-9012-3456"

last_digits = credit_number[-4:]
print(f"XXXX-XXXX-XXXX-{last_digits}")

```

Syntaxe	Résultat
texte[0:4]	"Pyth"
texte[::-2]	"Pto" (1 sur 2)
texte[1:]	"ython" (à partir de l'index 1)
texte[:3]	"Pyt" (jusqu'à l'index 2 inclus)
texte[::-1]	"nohtyP" (chaîne inversée)

formatage (flags)

Pour afficher proprement du texte, des nombres ou des variables avec un format lisible et contrôlé.

Syntaxe	Description	Exemple de sortie
f"{val:5}"	Largeur fixe (aligné à droite)	' 42'
f"{val:<5}"	Aligné à gauche	'42 '
f"{val:^5}"	Centré	' 42 '
f"{val:05}"	Zéros en padding	'00042'
f"{val:.2f}"	2 décimales (nombre flottant)	'3.14'
f"{pourcentage:.1%}"	Format pourcentage	0.25 → '25.0%'

- Le formatage f-string est lisible, moderne, et très utilisé.
- Combine les {} avec des options de mise en forme : alignement, décimales, zéros...



```
.py <br/># while loop = execute some code WHILE<br/><br/>name = input("Enter your name: ")<br/><br/>-while name == "":<br/>    print("You did not enter your nam<br/>    name = input("Enter your name: ")<br/><br/>print(f"Hello {name}")<br/><br/>while name == ""<br/>r your name<br/>Bro<br/><br/>with exit code 0
```

boucle **while**

Une boucle qui répète un bloc de code tant qu'une condition est vraie.

while condition:

bloc de code à répéter

La condition est testée avant chaque itération. Si elle est fausse dès le départ, le code ne s'exécute pas.

⚠ Attention aux boucles infinies

Attente d'une bonne saisie utilisateur

Répétition d'un calcul jusqu'à une condition atteinte



Exercice 4 : while et formatage

```
/*
  1. Créer un petit jeu où l'utilisateur doit deviner un
  nombre secret. Le programme utilise une boucle while pour répéter
  les tentatives, et des f-strings avec flags pour formater les
  messages.
*/
```

Tentative n°01 : Entrez un nombre : 5
Trop bas !

Tentative n°02 : Entrez un nombre : 10
Trop haut !

Tentative n°03 : Entrez un nombre : 7
Bravo ! Vous avez trouvé en 03 tentatives.

Exercice 4 : correction

```
secret = 7
essai = 0
trouve = False

while not trouve:
    essai += 1
    guess = int(input(f"Tentative n°{essai:02} : Entrez un nombre : "))

    if guess < secret:
        print("Trop bas !")
    elif guess > secret:
        print("Trop haut !")
    else:
        print(f"Bravo ! Vous avez trouvé en {essai:02} tentatives.")
        trouve = True
```

```
# while loop = execute some code WHILE

name = input("Enter your name: ")

while name == "":
    print("You did not enter your name")
    name = input("Enter your name: ")

print(f"Hello {name}")

while name == "":
    print("Enter your name")
    name = input("Bro")

with exit code 0
```

boucle for

Une boucle qui permet de parcourir une séquence (liste, chaîne, plage de nombres, etc.)

for variable in séquence:

bloc de code à répéter

À chaque tour, la variable prend la valeur suivante de la séquence.

range(x) génère les nombres de 0 à x-1

Syntaxe

range(n)

range(a, b)

range(a, b, step)

Résultat généré

de 0 à n-1

de a à b-1

de a à b-1, par palier de step



Exercice 5 : boucle for

```
/*
 1. Demander deux nombres à l'utilisateur : un début et une
 fin. Pour chaque entier entre ces deux bornes (incluses),
 afficher : Le nombre s'il est pair ou impair */
```

Début : 3

Fin : 8

```
3 est impair
4 est pair
5 est impair
6 est pair
7 est impair
8 est pair
```

Exercice 5 : correction

```
# Demander les bornes à l'utilisateur
debut = int(input("Début : "))

fin = int(input("Fin : "))

# Parcourir la plage de nombres
for nombre in range(debut, fin + 1):
    if nombre % 2 == 0:
        print(f"{nombre} est pair")
    else:
        print(f"{nombre} est impair")
```

```
# -- Création d'un dictionnaire
```

```
utilisateur = {  
    "nom": "Alice",  
    "age": 25,  
    "ville": "Paris"  
}
```

```
# -- Accès aux valeurs
```

```
print(utilisateur["nom"]) # Alice  
print(utilisateur["age"]) # 25
```

Méthode	Description
dict.keys()	Liste des clés
dict.values()	Liste des valeurs
dict.items()	Liste (clé, valeur)
dict.get("clé")	Récupérer une valeur en sécurité
dict.pop("clé")	Supprimer une clé
dict.clear()	Vider le dictionnaire

dictionnaires (dict)

Un dictionnaire est une **collection de paires clé → valeur**.

C'est l'équivalent d'un objet en JavaScript.

```
# -- Création d'un dictionnaire
```

```
utilisateur = {  
    "nom": "Alice",  
    "age": 25,  
    "ville": "Paris"  
}
```

```
# -- Accès aux valeurs
```

```
print(utilisateur["nom"]) # Alice  
print(utilisateur["age"]) # 25
```

Ils disposent de méthodes utiles

Mutable (on peut le modifier).

Clés uniques et généralement de type str.

Très pratique pour représenter des objets, enregistrements ou configurations.



Exercice 6 : dictionnaires

Créer un petit carnet d'adresses en utilisant un dictionnaire pour stocker des informations sur des personnes.

Consignes : Crée un dictionnaire contact avec les clés suivantes : "nom" "telephone" "email"

Demande à l'utilisateur de saisir ces informations et remplis le dictionnaire.

Affiche le contact de façon lisible.

Bonus : Permettre à l'utilisateur de modifier le numéro de téléphone puis d'afficher à nouveau le dictionnaire.

Exercice 6 : correction

```
# Création d'un dictionnaire vide
contact = {}

# Remplissage avec des saisies utilisateur
contact["nom"] = input("Entrez le nom : ")
contact["telephone"] = input("Entrez le téléphone : ")
contact["email"] = input("Entrez l'email : ")

# Affichage du contact
print("\nContact enregistré :")
for cle, valeur in contact.items():
    print(f"{cle} : {valeur}")

# Bonus : modification du numéro
nouveau_tel = input("\nNouveau téléphone : ")
contact["telephone"] = nouveau_tel

# Réaffichage
print("\nContact mis à jour :")
for cle, valeur in contact.items():
    print(f"{cle} : {valeur}")
```

```
fruits = ["apple", "orange",  
# print(dir(fruits))  
# print(help(fruits))  
# print(len(fruits))  
# print("pineapple" in fruits)  
  
# fruits[0] = "pineapple"  
  
# print(fruits[0])  
for fruit in fruits:  
    print(fruit)
```

les tableaux (lists)

Un tableau (ou liste) est une structure **de données ordonnée** qui peut contenir plusieurs valeurs (même ou différents types).

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

```
fruits = ["pomme", "banane", "kiwi"]
```

```
melange = [True, 42, "salut"]
```

Accéder aux éléments

```
print(fruits[0]) # "pomme"
```

```
print(fruits[-1]) # "kiwi" (dernier élément)
```

On peut parcourir facilement un tableau avec **for**

```
for fruit in fruits:
```

```
    print(fruit)
```



```
fruits = ["apple", "orange",
# print(dir(fruits))
# print(help(fruits))
# print(len(fruits))
# print("pineapple" in fruits)

# fruits[0] = "pineapple"
|
# print(fruits[0])
for fruit in fruits:
    print(fruit)
```

les ensembles (set)

Un set est une collection **non ordonnée** d'éléments **uniques**.

On ne peut pas les modifier par contre on peut en ajouter / retirer

```
animaux = {"chat", "chien", "cheval"}
```

⚠ Caractéristiques d'un set

- Pas d'indexation (pas de set[0]) → non ordonné
- Chaque élément est unique automatiquement
- Utilisé pour les opérations ensemblistes (union, intersection, etc.)

Idéal pour éliminer les doublons

Très utile pour les comparaisons ou les vérifications d'appartenance rapide.



Exercice 6a : lists

```
/*
```

```
    1. Crée une liste vide courses.Ajoute "lait", "pain" et  
    "œufs".Demande à l'utilisateur un autre produit à ajouter à la  
    liste.Affiche tous les éléments un par un avec une boucle for.
```

```
*/
```

```
/*
```

```
    2. Crée une liste de notes (ex : [14, 12, 16, 8]).Calcule la  
    somme et la moyenne manuellement (sans sum()).Affiche la moyenne  
    avec 2 chiffres après la virgule.*/
```

```
/*
```

```
    3. Crée une liste de prénoms.Crée une nouvelle liste  
    inversée manuellement (sans .reverse() ni [::-1]).Affiche les  
    deux listes.
```

```
*/
```

Exercice 1 — Liste de courses

python

```
# Création de la liste vide
courses = []

# Ajout de produits
courses.append("lait")
courses.append("pain")
courses.append("œufs")

# Demander à l'utilisateur un produit
produit = input("Entrez un produit à ajouter : ")
courses.append(produit)

# Afficher tous les éléments
print("\nListe de courses :")
for item in courses:
    print(f"- {item}")
```

💡 Points clés à retenir :

- `append()` ajoute à la fin d'une liste.
- Boucle `for` simple pour parcourir la liste.

Exercice 6a : corrections

Exercice 2 — Moyenne de notes

python

```
# Liste de notes
notes = [14, 12, 16, 8]

# Calcul manuel de la somme
somme = 0
for note in notes:
    somme += note

# Calcul de la moyenne
moyenne = somme / len(notes)

# Affichage formaté à 2 décimales
print(f"Moyenne : {moyenne:.2f}")
```

💡 Points clés à retenir :

- `len(notes)` → nombre d'éléments.
- `{moyenne:.2f}` → formatage avec 2 décimales.

Exercice 3 — Inverser une liste

python

```
# Liste initiale
prenoms = ["Alice", "Bob", "Charlie", "David"]

# Nouvelle liste inversée
inversee = []
for i in range(len(prenoms) - 1, -1, -1):
    inversee.append(prenoms[i])

# Affichage
print("Liste originale : ", prenoms)
print("Liste inversée : ", inversee)
```

💡 Points clés à retenir :

- `range(len(prenoms) - 1, -1, -1)` → parcours à l'envers.
- On construit une **nouvelle liste** (pas de modification en place).

Exercice 6b : sets

```
/*
```

```
    1. Demande à l'utilisateur de saisir plusieurs prénoms,  
    séparés par des virgules.Exemple : "Alice,Bob,Alice,Tom,Bob"
```

```
Transforme la chaîne en liste avec .split(',').Convertis la  
liste en set pour éliminer les doublons.Affiche le set obtenu.
```

```
/*
```

```
    2. Crée deux sets :
```

```
groupe_a = {"Alice", "Bob", "Claire"}
```

```
groupe_b = {"Claire", "David", "Emma"}
```

```
Affiche :Les personnes dans les deux groupes.Les personnes  
uniques à chaque groupe.La liste totale sans doublons.
```

```
/*
```

```
    3. Crée une liste de mots interdits : ["spam", "arnaque",  
    "escroquerie"].Demande à l'utilisateur de saisir un message.Si  
    un mot interdit est trouvé dans le message, affiche "Message  
    rejeté", sinon "Message accepté".💡 Astuce : utiliser une boucle  
    + in.*/
```

Exercice 6b : sets

python

```
# Saisie de plusieurs prénoms séparés par des virgules
saisie = input("Entrez plusieurs prénoms séparés par des virgules : ")

# Conversion en liste
liste_prenoms = saisie.split(",")

# Nettoyage : supprimer les espaces autour des prénoms
liste_prenoms = [prenom.strip() for prenom in liste_prenoms]

# Conversion en set pour éliminer les doublons
ensemble_prenoms = set(liste_prenoms)

# Affichage
print("Prénoms uniques :", ensemble_prenoms)
```

💡 Points clés à retenir :

- `.split(",")` → découpe en liste.
- `.strip()` → supprime espaces en début/fin.
- `set(liste)` → supprime automatiquement les doublons.

✓ Exercice 5 — Comparaison de groupes

python

```
groupe_a = {"Alice", "Bob", "Claire"}
groupe_b = {"Claire", "David", "Emma"}

# Intersection (dans les deux groupes)
commun = groupe_a & groupe_b
print("Dans les deux groupes :", commun)

# Différence (uniques à A)
uniques_a = groupe_a - groupe_b
print("Uniquement dans le groupe A :", uniques_a)

# Différence (uniques à B)
uniques_b = groupe_b - groupe_a
print("Uniquement dans le groupe B :", uniques_b)

# Union (tous sans doublons)
tous = groupe_a | groupe_b
print("Tous les participants :", tous)
```

✓ Exercice 6 — Présence d'un mot interdit

python

```
mots_interdits = ["spam", "arnaque", "escroquerie"]

# Demander le message
message = input("Entrez votre message : ").lower() # minuscule pour simplifier la vérif

# Vérification
interdit = False
for mot in mots_interdits:
    if mot in message:
        interdit = True
        break # on peut arrêter dès qu'on trouve un mot interdit

# Résultat
if interdit:
    print("Message rejeté ❌")
else:
    print("Message accepté ✅")
```

les iterations

On peut parcourir ces différents objets :

Les ensembles (set)

Non ordonnés

Pas de doublons

Pas d'indexation

```
animaux = {"chat", "chien", "chat"}  
for animal in animaux:  
    print(animal)
```

Les listes (list)

Ordonnées

Autorisent les doublons

Indexées

```
fruits = ["pomme", "banane", "kiwi"]  
for fruit in fruits:  
    print(fruit)
```

Les dictionnaires (dict)

Stockent des paires clé → valeur

Pas de doublons de clés

→ Différentes façons de parcourir : `dict.keys()` → uniquement les clés

```
utilisateur = {"nom": "Alice", "age": 25}  
  
for cle in utilisateur.keys():  
    print(cle)
```

`dict.values()` → uniquement les valeurs

```
for valeur in utilisateur.values():  
    print(valeur)  
# Alice  
# 25
```

`dict.items()` → couples (clé, valeur)

```
for cle, valeur in utilisateur.items():  
    print(cle, ":", valeur)  
# nom : Alice  
# age : 25
```



Exercice 6c : iterations

```
/*
```

```
    Crée une liste eleves avec quelques prénoms, dont certains  
en doublon.
```

```
Crée cette liste en set pour supprimer les doublons, puis  
affiche le set.
```

```
Crée un dictionnaire notes où chaque élève a une note (clé =  
prénom, valeur = note).
```

```
Affiche :
```

```
Toutes les clés (noms des élèves).
```

```
Toutes les valeurs (notes).
```

```
Toutes les paires clé/valeur avec .items().
```

Exercice 6c : iterations

```
# 1. Création d'une liste avec doublons
eleves = ["Alice", "Bob", "Alice", "Claire"]

print("Liste originale :", eleves)

# 2. Transformation en set pour supprimer les doublons
eleves_set = set(eleves)
print("Set sans doublons :", eleves_set)

# 3. Création d'un dictionnaire avec les notes
notes = {
    "Alice": 15,
    "Bob": 12,
    "Claire": 18
}

# 4. Affichage des clés, valeurs et paires

print("\nClés (élèves) :")
for cle in notes.keys():
    print(cle)

print("\nValeurs (notes) :")
for valeur in notes.values():
    print(valeur)

print("\nPaires (clé, valeur) :")
for cle, valeur in notes.items():
    print(f"{cle} : {valeur}")
```

unction = A block of reusable code
place () after the colon

```
happy_birthday(name, age):  
    print(f"Happy birthday to {name}!")  
    print("You are old!")  
    print("Happy birthday to you!")  
    print()
```

```
py_birthday("Bro", 20)  
py_birthday("Steve", 30)  
py_birthday("Joe", 40)  
py_birthday()
```

les fonctions

Bloc de code réutilisable qui réalise une tâche précise. Elle permet de structurer le programme et d'éviter les répétitions.

Pour la **définir** :

```
def dire_bonjour():  
    print("Bonjour !")
```

Puis pour **l'appeler** :

```
dire_bonjour()
```

Avec paramètre(s)

```
def saluer(nom):  
    print(f"Bonjour {nom} !")  
  
saluer("Alice")
```

Avec return

```
def addition(a, b):  
    return a + b  
  
resultat = addition(5, 3)  
print(resultat) # 8
```



Exercice 7 fonctions

```
/*
```

Écrire un programme qui utilise plusieurs fonctions pour gérer des notes d'élèves et afficher la moyenne.

Crée une fonction `saisir_notes()` qui demande à l'utilisateur d'entrer plusieurs notes (saisie terminée quand l'utilisateur tape -1). La fonction renvoie la liste des notes.

Crée une fonction `moyenne(notes)` qui reçoit une liste de notes et renvoie la moyenne.

Crée une fonction `afficher_resultat(moyenne)` qui affiche un message formaté : "Très bien" si la moyenne ≥ 16 "Bien" si la moyenne ≥ 14 "Assez bien" si la moyenne ≥ 12 "Passable" si la moyenne ≥ 10 "Insuffisant" sinon

Dans le programme principal :Appeler `saisir_notes()`Calculer la moyenne avec `moyenne()`Afficher le résultat avec `afficher_resultat()`

Exercice 7 correction

```
# Fonction pour saisir les notes
def saisir_notes():
    notes = []
    while True:
        note = float(input("Entrez une note (ou -1 pour finir) : "))
        if note == -1: # condition d'arrêt
            break
        notes.append(note)
    return notes

# Fonction pour calculer la moyenne
def moyenne(notes):
    if len(notes) == 0: # éviter la division par zéro
        return 0
    return sum(notes) / len(notes)

# Fonction pour afficher le résultat avec mention
def afficher_resultat(moyenne):
    print(f"Moyenne : {moyenne:.2f}")
    if moyenne >= 16:
        print("Très bien")
    elif moyenne >= 14:
        print("Bien")
    elif moyenne >= 12:
        print("Assez bien")
    elif moyenne >= 10:
        print("Passable")
    else:
        print("Insuffisant")
```

```
# ----- Programme principal -----
notes = saisir_notes()          # étape 1 : saisie
moy = moyenne(notes)           # étape 2 : calcul
afficher_resultat(moy)         # étape 3 : affichage
```

unction = A block of reusable code
place () after the colon

```
happy_birthday(name, age):  
    print(f"Happy birthday to {name}!")  
    print("You are old!")  
    print("Happy birthday to you!")  
    print()
```

```
py_birthday("Bro", 20)  
py_birthday("Steve", 30)  
py_birthday("Joe", 40)  
py_birthday()
```

nommages arguments

Un keyword argument permet d'appeler une fonction en nommant les paramètres.

Cela rend l'appel plus lisible et permet de changer l'ordre des arguments.

```
1 def saluer(nom, age):  
2     print(f"Bonjour {nom}, tu as {age} ans.")  
3  
4 # Appel classique (positional arguments)  
5 saluer("Alice", 25)  
6  
7 # Appel avec keyword arguments  
8 saluer(age=25, nom="Alice")
```

Attention : Les arguments positionnels doivent venir avant les arguments nommés (si on a les 2 cas dans un appel)



Exercice 8 arguments

```
/*
```

Écrire une fonction qui utilise des arguments nommés et des valeurs par défaut.

Consignes : Crée une fonction `creer_utilisateur(nom, age, pays="France")`.

La fonction doit afficher un message du type :

Utilisateur : Alice, 25 ans, France

Dans le programme principal, appelle la fonction :

Une fois avec arguments positionnels

Une fois avec arguments nommés dans le désordre.

Une fois en utilisant la valeur par défaut pour pays

Exercice 8 arguments

```
# Définition de la fonction avec une valeur par défaut pour 'pays'  
def creer_utilisateur(nom, age, pays="France"):  
    print(f"Utilisateur : {nom}, {age} ans, {pays}")  
  
# ----- Programme principal -----  
  
# 1) Appel avec arguments positionnels  
creer_utilisateur("Alice", 25, "Canada")  
# → Utilisateur : Alice, 25 ans, Canada  
  
# 2) Appel avec arguments nommés dans le désordre  
creer_utilisateur(age=30, nom="Bob", pays="Suisse")  
# → Utilisateur : Bob, 30 ans, Suisse  
  
# 3) Appel en utilisant la valeur par défaut pour 'pays'  
creer_utilisateur("Charlie", 22)  
# → Utilisateur : Charlie, 22 ans, France
```

```
def add(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total  
  
print(add(1, 2, 3, 4,))
```

arguments multiples

*args permet de passer un nombre variable d'arguments ² à une fonction. Les arguments sont regroupés dans un **tuple**.

```
def ma_fonction(*args):  
    print(args) # args est de type tuple
```

*args est utilisé pour **gérer plusieurs arguments optionnels**.
Toujours placé après les paramètres classiques.
Donne de la flexibilité aux fonctions.

Tuple= méthodes associées

<https://docs.python.org/fr/3.13/tutorial/datastructures.html>



```
def add(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total  
  
print(add(1, 2, 3, 4,))
```

arguments nommées multiples

****kwargs** permet de passer un **nombre variable d'arguments nommés** à une fonction.

Les arguments sont regroupés dans un **dictionnaire** (clé: valeur).

```
1 def ma_fonction(**kwargs):  
2     print(kwargs) # kwargs est un dictionnaire
```

****kwargs = arguments nommés variables.**

Stockés dans un **dictionnaire**.

Très utile pour des fonctions avec **options flexibles**.

Dictionnaire = méthodes associées

<https://docs.python.org/fr/3.13/tutorial/datastructures.html>



Exercice 9 args & kwargs

```
/*
```

Créer une fonction `facture(*articles, **options)` qui :

- Calcule le prix total d'une série d'articles (passés en `*args`).
- Applique des options supplémentaires (passées en `**kwargs`) comme une remise ou une TVA.

1. La fonction prend :des prix d'articles via `*args` (ex:
`facture(10, 20, 30)`)
2. des options nommées via `**kwargs`, comme :
`:remise=10` (réduction en %)
`tva=20` (ajout en %)
3. La fonction affiche :Le sous-totalLe montant de la remise appliquée (si présente)Le montant de la TVA appliquée (si présente)Le total final

Exercice

args

```
def facture(*articles, **options):
    """
    Calcule une facture avec une liste d'articles (args)
    et des options nommées (kwargs) : remise, tva, devise.
    """

    # ----- Étape 1 : calcul du sous-total -----
    sous_total = sum(articles)
    print(f"Sous-total : {sous_total:.2f} €")

    # ----- Étape 2 : appliquer la remise si présente -----
    remise = options.get("remise", 0) # par défaut : 0
    montant_remise = sous_total * (remise / 100)
    if remise > 0:
        print(f"Remise ({remise}%) : -{montant_remise:.2f} €")

    # ----- Étape 3 : appliquer la TVA si présente -----
    tva = options.get("tva", 0) # par défaut : 0
    montant_tva = (sous_total - montant_remise) * (tva / 100)
    if tva > 0:
        print(f"TVA ({tva}%) : +{montant_tva:.2f} €")

    # ----- Étape 4 : total final -----
    total = sous_total - montant_remise + montant_tva
    print(f"Total final : {total:.2f} €")

# ----- Programme principal -----
facture(10, 20, 30, remise=10, tva=20)
```

```
word = "APPLE"  
  
letter = input("Guess a letter in the word : ")  
  
if letter in word:  
    print(f"There is a {letter}")  
else:  
    print(f"{letter} was not found")
```

the secret word: Z

opérateur **in** / **not in**

Les opérateurs **in** et **not in** permettent **de tester la présence d'un élément dans un itérable** (liste, set, dict, chaîne...).

```
fruits = ["pomme", "banane", "kiwi"]  
  
print("pomme" in fruits)      # True  
print("orange" not in fruits) # True
```

in = appartenance

not in = non-appartenance

Fonctionne avec : listes, sets, tuples, chaînes, dictionnaires (par clés)



```
# Créer une liste des carrés de 0 à 4 :
```

```
carrés = [x**2 for x in range(5)]  
print(carrés) # [0, 1, 4, 9, 16]
```

```
# Avec condition :
```

```
# Créer une liste des nombres pairs de 0 à 9 :
```

```
pairs = [x for x in range(10) if x % 2 == 0]  
print(pairs) # [0, 2, 4, 6, 8]
```

```
# 2 boucles imbriquées
```

```
pairs_produits = [x*y for x in range(3) for y in range(4)]  
print(pairs_produits) # [0, 0, 0, 0, 0, 1, 2, 3, 0, 2, 4, 6]
```

Constructeur de list

Une list comprehension permet de **créer une liste de manière concise et lisible à partir d'une autre liste ou d'un itérable.**

Elle remplace souvent une boucle for traditionnelle.



```
[nouvel_element for element in iterable if condition]
```

nouvel_element : ce que vous voulez mettre dans la nouvelle liste

element : chaque élément de l'itérable d'origine

if condition : optionnel, filtre les éléments selon une condition

Plus compact et lisible que les boucles for classiques

Permet de filtrer facilement avec if

Peut inclure des transformations ou calculs



Exercice 10 creation list

```
/*
```

Créez une liste nombres contenant les entiers de 1 à 10.

À partir de cette liste, créez :

- a) Une nouvelle liste carrés contenant les carrés de ces nombres.
- b) Une nouvelle liste pairs ne contenant que les nombres pairs.
- c) Une liste pairs_carrés contenant les carrés uniquement des nombres pairs.
- d) Bonus : créez une liste table_multiplication contenant tous les produits $x*y$ pour x et y allant de 1 à 3.

Exercice 10 creation list

```
# Créer une liste de nombres de 1 à 10
nombres = [x for x in range(1, 11)]
print(nombres) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Créer une liste des carrés des nombres
carrés = [x**2 for x in nombres]
print(carrés) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Créer une liste des nombres pairs
pairs = [x for x in nombres if x % 2 == 0]
print(pairs) # [2, 4, 6, 8, 10]

# Créer une liste des carrés uniquement des nombres pairs
pairs_carrés = [x**2 for x in nombres if x % 2 == 0]
print(pairs_carrés) # [4, 16, 36, 64, 100]

# Bonus : créer une liste avec tous les produits x*y pour x et y de 1 à
table_multiplication = [x*y for x in range(1, 4) for y in range(1, 4)]
print(table_multiplication) # [1, 2, 3, 2, 4, 6, 3, 6, 9]
```

```
def day_of_week(day):
    if day == 1:
        return "It is Sunday"
    elif day == 2:
        return "It is Monday"
    elif day == 3:
        return "It is Tuesday"
    elif day == 4:
        return "It is Wednesday"
    elif day == 5:
        return "It is Thursday"
    elif day == 6:
        return "It is Friday"
    elif day == 7:
        return "It is Saturday"
    else:
        return "Not a valid day"
```

Match...case (depuis v3.10)

Permet de comparer une variable à plusieurs cas possibles

Similaire au **switch** qu'on trouve dans d'autres langages

Plus puissant : supporte aussi la **déstructuration** et les **patterns**

```
match variable:
    case valeur1:
        # instructions si variable == valeur1
    case valeur2:
        # instructions si variable == valeur2
    case _:
        # cas par défaut (équivalent de "default")
```

Lisibilité accrue par rapport aux if/elif/else

Supporte les **motifs complexes** (tuples, listes, classes, etc.)

Le `_` permet de gérer le **cas par défaut**



Exercice 11 match case et lists

```
/*
```

Écris un programme qui analyse une liste de nombres et affiche un message différent selon son contenu :

Si la liste est vide → "La liste est vide"

Si la liste contient un seul élément → "Un seul élément : X"

Si la liste contient deux éléments → "Deux éléments : X et Y"

Si la liste contient au moins trois éléments → "Liste plus longue, premier élément : X"

Exercice 11 match case et lists

```
ma_liste = [1, 2, 3] # tu peux modifier pour tester différents cas

match ma_liste:
    case []:
        print("La liste est vide")
    case [x]:
        print(f"Un seul élément : {x}")
    case [x, y]:
        print(f"Deux éléments : {x} et {y}")
    case [x, *reste]:
        print(f"Liste plus longue, premier élément : {x}")
```

```

# -----
# mon_programme.py
# -----

# ❶ Définition des fonctions
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:
        return a / b
    else:
        return "Erreur : division par zéro"

# ❷ Code principal (point d'entrée)
if __name__ == "__main__":
    print("Programme principal lancé !\n")

    x, y = 10, 5
    print(f"{x} + {y} = {addition(x, y)}")
    print(f"{x} - {y} = {soustraction(x, y)}")
    print(f"{x} * {y} = {multiplication(x, y)}")
    print(f"{x} / {y} = {division(x, y)}")

```

Modules & main

Qu'est-ce que `__name__` ?

En Python, chaque fichier est un **module**.

La variable spéciale `__name__` contient :

- `"__main__"` si le fichier est exécuté directement.
- Le nom du module si le fichier est importé ailleurs.

Pourquoi utiliser `if __name__ == "__main__":` ?

Permet de définir un point d'entrée du programme.

Le code dans ce bloc ne s'exécute **que si le fichier est lancé directement**, pas lorsqu'il est importé.

Bonne pratique pour séparer :

- les fonctions / classes réutilisables
- le code d'exécution principal

```

def dire_bonjour():
    print("Bonjour !")

# S'exécute seulement si ce fichier est lancé directement
if __name__ == "__main__":
    dire_bonjour()

```

Si on lance le fichier → affiche Bonjour !

Si on importe ce fichier dans un autre module → rien ne s'affiche



Exercice 12 modules et main

```
/*
```

Créer un mini-projet avec deux fichiers :

utils.py

Contiendra des fonctions utilitaires :

carres_pairs(n) → renvoie la liste des carrés des nombres pairs de 0 à n. (Utiliser une compréhension de liste) analyse_liste(liste) → utilise match...case pour analyser la liste :

- liste vide → "La liste est vide"
- liste d'un seul élément → "Un seul élément : X"
- liste de deux éléments → "Deux éléments : X et Y"
- liste plus longue → "Liste plus longue, premier élément : X"

main.py

Importer les fonctions de utils.py

Dans le bloc if __name__ == "__main__":, il devra :

- Demander à l'utilisateur un nombre n
- Calculer et afficher les carrés pairs jusqu'à n
- Analyser la liste obtenue avec analyse_liste

Exercice 12 modules et main

```
# utils.py

def carres_pairs(n):
    """Renvoie les carrés des nombres pairs de 0 à n"""
    return [x**2 for x in range(n+1) if x % 2 == 0]

def analyse_liste(liste):
    """Analyse une liste avec match...case"""
    match liste:
        case []:
            return "La liste est vide"
        case [x]:
            return f"Un seul élément : {x}"
        case [x, y]:
            return f"Deux éléments : {x} et {y}"
        case [x, *reste]:
            return f"Liste plus longue, premier élément : {x}"
```

```
# main.py
from utils import carres_pairs, analyse_liste

if __name__ == "__main__":
    # Demander un nombre à l'utilisateur
    n = int(input("Entrez un nombre : "))

    # Générer les carrés pairs
    resultat = carres_pairs(n)
    print("Carrés pairs :", resultat)

    # Analyser la liste obtenue
    analyse = analyse_liste(resultat)
    print("Analyse :", analyse)
```