Chapitre 2. Notions de base

1. Introduction au langage Python

Python est un langage de programmation interprété, orienté objet, multiplateforme et open source. Il est largement utilisé en :

- Science des données et calcul scientifique (NumPy, SciPy, Pandas, Matplotlib)
- Intelligence artificielle et machine learning (TensorFlow, PyTorch)
- Traitement du signal et d'images
- Développement web, scripting, automatisation, simulation numérique, etc.

2. Caractéristiques principales

Caractéristique	Description
Interprété	Pas de compilation : le code est exécuté ligne par ligne
Typage dynamique	Le type d'une variable est déterminé automatiquement
Portable	Fonctionne sous Windows, Linux, macOS
Multi-paradigme	Supporte impératif, fonctionnel, orienté objet
Extensible	Peut être intégré à C, C++, Fortran
Grande bibliothèque standard	Modules pour math, fichiers, réseau, etc.

3. Premier programme Python

Exemple :

```
print("Bonjour, Python !")
```

Sortie :

```
Bonjour, Python !
```

Le mot-clé print() affiche un texte à l'écran.

Python reconnaît automatiquement les chaînes de caractères entre guillemets "..." ou '...'.

4. Variables et affectation

Une variable sert à stocker une valeur. En Python, il n'est pas nécessaire de déclarer le type.

Exemple :

```
x = 5
y = 3.14
nom = "Alice"
print(x, y, nom)
```

Sortie :

```
5 3.14 Alice
```

Python détermine les types automatiquement :

- int pour les entiers
- float pour les réels
- str pour les chaînes de caractères

5. Types d'objets Python

Tout est objet en Python : chaque élément possède un type et des méthodes associées.

5.1. Types numériques

Type	Description	Exemple
int	Entiers relatifs	x = 42
float	Nombres à virgule flottante	y = 3.1415
complex	Nombres complexes	z = 2 + 3j

Exemple :

```
a = 7
b = 2.5
c = 1 + 4j
print(type(a), type(b), type(c))
```

Sortie :

```
<class 'int'> <class 'float'> <class 'complex'>
```

Python gère automatiquement les conversions :

Exemple :

```
x = 3
y = 2.0
z = x + y  # z devient float
print(z, type(z))
```

Sortie :

```
5.0 <class 'float'>
```

5.2. Type booléen (bool)

Les valeurs booléennes sont :

- True
- False

Elles sont souvent le résultat de comparaisons.

Exemple :

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

5.3 Type chaîne de caractères (str)

Les chaînes peuvent être définies avec '...' ou "...".

Exemple :

```
nom = "Python"
print(nom[0])     # P
print(nom[-1])    # n
print(nom[1:4])   # yth
```

Les chaînes sont immuables **:** on ne peut pas modifier un caractère directement.

Exemple :

```
# Erreur :
nom[0] = 'p'
# TypeError: 'str' object does not support item assignment
```

5.4 Type None

None représente l'absence de valeur (équivalent du null en C/Java).

Exemple :

```
x = None
print(x)
```

Sortie :

```
None
```

5.5 Conversion de types

Python permet de convertir explicitement les types.

Exemple :

```
x = 3.7
y = int(x)
z = str(x)
print(y, type(y))
print(z, type(z))
```

Sortie :

```
3 <class 'int'>
3.7 <class 'str'>
```

6. Les opérateurs

Les opérateurs sont des symboles qui permettent d'effectuer des opérations sur des variables ou des valeurs. Python supporte plusieurs catégories d'opérateurs :

1. Arithmétiques
2. De comparaison
3. Logiques
4. D'affectation

5. D'appartenance
6. D'identité
7. Binaires

6.1.Opérateurs arithmétiques

Ces opérateurs réalisent les opérations mathématiques de base.

Opérateur	Signification	Exemple	Résultat
+	Addition	3 + 2	5
-	Soustraction	3 - 2	1
*	Multiplication	3 * 2	6
/	Division réelle	3 / 2	1.5
//	Division entière	3 // 2	1
%	Modulo (reste)	3 % 2	1
**	Puissance	3 ** 2	9

Exemple :

```
a = 10
b = 3
print(a + b)
print(a / b)
print(a // b)
print(a % b)
print(a ** b)
```

Sortie :

```
13
3.3333333333333335
3
1
1000
```

⚠ Attention :

- / renvoie toujours un float
- // renvoie la partie entière

6.2. Opérateurs de comparaison

Ils permettent de comparer deux valeurs et renvoient un résultat booléen (True ou False).

Opérateur	Signification	Exemple	Résultat

==	Égal à	5 == 5	True
!=	Différent de	5 != 3	True
>	Supérieur à	5 > 3	True
<	Inférieur à	5 < 3	False
>=	Supérieur ou égal	5 >= 5	True
<=	Inférieur ou égal	3 <= 5	True

Exemple :

```
x = 10
y = 20
print(x == y) #Égal à
print(x != y) #Différent de
print(x > y)  #Supérieur à
print(x < y)  #Inférieur à
print(x >= y)  #Supérieur ou égal
print(x <=y)  #Inférieur ou égal
```

Sortie :

```
False
True
False
True
False
True
```

6.3. Opérateurs logiques

Les opérateurs logiques servent à combiner des conditions booléennes.

Opérateur	**Signification**	**Exemple**	**Résultat**
and	Vrai si les deux conditions sont vraies	(x > 0) and (x < 10)	True si x entre 0 et 10
or	Vrai si au moins une condition est vraie	(x < 0) or (x > 10)	True si x est hors [0,10]
not	Inverse la condition	not(x > 0)	True si x <= 0

Exemple :

```
x = 7
print((x > 0) and (x < 10))   # True
```

```
print((x < 0) or (x > 10))    # False
print(not(x == 7))            # False
```

Sortie

```
True
False
False
```

6.4. Opérateurs d'affectation

Ils servent à assigner ou mettre à jour la valeur d'une variable.

Opérateur	Exemple	Équivalent à
=	x = 5	affectation simple
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
//=	x //= 3	x = x // 3
%=	x %= 3	x = x % 3
**=	x **= 3	x = x ** 3

Exemple :

```
x = 5
print(x)
x += 1
print(x)
x **= 2
print(x)
x -= 2
print(x)
x *= 3
print(x)
x /= 3
print(x)
x //= 2
print(x)
x %= 2
print(x)
```

Sortie :

```
5
6
36
34
102
34.0
17.0
1.0
```

6.5. Opérateurs d'appartenance

Ces opérateurs testent si un élément appartient à une séquence (chaîne, liste, tuple...).

Opérateur	Signification	Exemple	Résultat
in	Élément présent	'a' in 'python'	False
not in	Élément absent	'y' not in 'python'	False

Exemple :

```
texte = "Bonjour Python"
print("Python" in texte)
print("Java" not in texte)
```

Sortie :

```
True
True
```

6.6. Opérateurs d'identité

Ils testent si deux variables pointent vers le même objet (même adresse mémoire).

Opérateur	Signification	Exemple	Résultat
is	Même objet	a is b	True si identiques
is not	Objets différents	a is not b	True si distincts

Exemple :

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b)     # True
print(a is c)     # False
```

6.7. Opérateurs binaires

Ils s'appliquent sur des entiers et manipulent les bits.

Opérateur	Signification	Exemple	Résultat (en binaire)
&	ET binaire	5 & 3	1 (0101 & 0011 = 0001)
|	OU binaire	5 | 3	7 (0111)
^	OU exclusif	5 ^ 3	6 (0101 ^ 0011 = 0110)
~	Négation binaire	~5	-6
<<	Décalage à gauche	5 << 1	10
>>	Décalage à droite	5 >> 1	2

Exemple :

```
a = 5    # 0101
b = 3    # 0011
print(a & b)
print(a | b)
print(a ^ b)
print(~a)
print(a << 1)
print(a >> 1)
```

Sortie :

```
1
7
6
-6
10
2
```

7. Les structures de données

Les structures de données sont des objets Python capables de contenir plusieurs valeurs. Elles permettent d'organiser, de manipuler et d'accéder efficacement à l'information.

Python dispose nativement de plusieurs types de structures très puissantes :

1. Les listes
2. Les tuples
3. Les dictionnaires
4. Les ensembles (sets)
5. Les chaînes de caractères (qui sont aussi des séquences)

7.1.Les listes

Une liste est une séquence ordonnée et modifiable d'éléments, potentiellement de types différents. Elle est définie entre crochets [].

Exemple :

```
notes = [15, 12, 18, 9]
melange = [10, "Python", True, 3.14]
print(notes)
print(melange)
```

a. Accès aux éléments

Les éléments sont indexés à partir de **0**.

Exemple :

```
notes = [15, 12, 18, 9]
print(notes[0])    # Premier élément
print(notes[-1])   # Dernier élément
print(notes[1:3])  # Sous-liste (indices 1 et 2)
```

b. Modification

Les listes sont mutables, donc on peut changer leurs éléments :

Exemple :

```
notes[0] = 17
print(notes)
```

On peut aussi ajouter ou supprimer des éléments :

Méthode	**Fonction**	**Exemple**
append(x)	Ajoute un élément à la fin	notes.append(20)
insert(i, x)	Insère à la position i	notes.insert(1, 14)
remove(x)	Supprime la première occurrence de x	notes.remove(12)
pop(i)	Supprime et renvoie l'élément d'indice i	notes.pop(0)
sort()	Trie la liste	notes.sort()
reverse()	Inverse la liste	notes.reverse()

Exemple :

```
notes = [15, 12, 18, 9]
print(notes)
notes.append(14)
print(notes)
notes.insert(1, 13)
print(notes)
notes.remove(12)
print(notes)
notes.pop(0)
print(notes)
notes.sort()
print(notes)
notes.reverse()
print(notes)
```

Sortie :

```
[15, 12, 18, 9]
[15, 12, 18, 9, 14]
[15, 13, 12, 18, 9, 14]
[15, 13, 18, 9, 14]
[13, 18, 9, 14]
[9, 13, 14, 18]
[18, 14, 13, 9]
```

c. Parcours d'une liste

Exemple :

```
notes = [9, 12, 14, 15, 18]
for n in notes:
    print(n)
```

Sortie

```
9
12
14
15
18
```

d. Compréhensions de listes

Une liste par compréhension permet de créer une nouvelle liste à partir d'une expression.

Exemple :

```
carres = [x**2 for x in range(6)]
print(carres)
```

Sortie :

```
[0, 1, 4, 9, 16, 25]
```

7.2. Les tuples

Un tuple est une séquence ordonnée mais immuable (on ne peut pas la modifier après création). Défini avec des parenthèses ().

Exemple :

```
coord = (4, 5)
print(coord[0])   # 4
```

Avantages

- Moins de mémoire qu'une liste
- Plus rapide à parcourir
- Peut être utilisé comme **clé** dans un dictionnaire (contrairement aux listes)

7.3. Les dictionnaires

Un dictionnaire est une collection non ordonnée d'éléments sous forme clé → valeur. Il est défini avec des accolades {}.

Exemple :

```
etudiant = {"nom": "Alice", "age": 22, "note": 15.5}
print(etudiant["nom"])
```

Sortie

Alice

a. Modification

Méthode	Description	Exemple
d[key] = value	Ajoute/modifie une paire clé-valeur	etudiant["age"] = 23
pop(key)	Supprime une entrée	etudiant.pop("note")
keys()	Retourne les clés	etudiant.keys()
values()	Retourne les valeurs	etudiant.values()
items()	Retourne paires clé-valeur	etudiant.items()

Exemple :

```
etudiant["age"] = 23
for cle, valeur in etudiant.items():
    print(cle, ":", valeur)
```

Sortie :

```
nom : Alice
age : 23
note : 15.5
```

7.4. Les ensembles (sets)

Un ensemble est une collection non ordonnée et sans doublons, défini par {}.

Exemple :

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A | B)  # Union
print(A & B)  # Intersection
print(A - B)  # Différence
print(B - A)  # Différence
print(A ^ B)  # Différence symétrique
```

Sortie :

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{5, 6}
{1, 2, 5, 6}
```

7.5. Les chaînes de caractères

Les chaînes sont des séquences immuables de caractères (str).

a. Indexation et slicing

Exemple :

```
texte = "Python"
print(texte[0])
print(texte[-1])
print(texte[1:4])
```

sortie

```
P
n
yth
```

b. Fonctions utiles

Méthode	Fonction
len(texte)	longueur
upper()	majuscules
lower()	minuscules
replace(a, b)	remplace
split()	découpe en liste
join(liste)	joint une liste en chaîne
find()	trouve une sous-chaîne

Exemple :

```
s = "Bonjour Python"
print(s.upper())
print(s.split())
print("-".join(["un", "deux", "trois"]))
```

Sortie

```
BONJOUR PYTHON
['Bonjour', 'Python']
un-deux-trois
```

8. Contrôle du flux d'exécution

Les programmes ne se contentent pas d'exécuter les instructions les unes après les autres :

Ils doivent choisir ou répéter certaines opérations selon les conditions**.**

Python offre deux grands types d'instructions pour cela :

a. **Les instructions conditionnelles** (if, elif, else)

b. **Les instructions répétitives** (for, while)

8.1.Les instructions conditionnelles

a. Syntaxe générale

```
if condition:
    bloc_d_instructions_1
elif autre_condition:
    bloc_d_instructions_2
else:
    bloc_d_instructions_3
```

⚠ Les **indentations** (retraits) sont essentielles :

Elles indiquent quelles instructions appartiennent à quel bloc. Par convention, l'indentation est de 4 espaces**.**

b. Exemple simple

```
x = 10
if x > 0:
    print("x est positif")
elif x == 0:
    print("x est nul")
else:
    print("x est négatif")
```

Sortie :

```
x est positif
```

c. Condition multiple

```
age = 25
if age >= 18 and age < 60:
    print("Adulte")
else:
    print("Non adulte")
```

Sortie

```
Adulte
```

d. Instruction conditionnelle compacte

Python permet d'écrire une condition sur une seule ligne (opérateur ternaire) :

Example

```
message = "Pair" if x % 2 == 0 else "Impair"
```

Sortie

```
Pair
```

e. Imbrication de conditions

Exemple :

```
x = 15
if x > 0:
    if x % 2 == 0:
        print("x est positif et pair")
    else:
        print("x est positif et impair")
```

Sortie

```
x est positif et impair
```

8.2. Les instructions répétitives

Les boucles permettent de répéter une ou plusieurs instructions. Python possède deux boucles principales : for et while.

a. Boucle for

Elle s'utilise pour **parcourir une séquence** (liste, chaîne, tuple, etc.).

Exemple :

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

Sortie :

```
0
1
2
3
4
```

Ici, range(5) génère la séquence [0, 1, 2, 3, 4].

b. Boucle for sur une liste

Exemple :

```
noms = ["Ali", "Mina", "Sami"]
for k in noms:
    print("Bonjour", k)
```

Sortie :

```
Bonjour Ali
Bonjour Mina
Bonjour Sami
```

c. Boucle for avec indices

Exemple :

```
for i, val in enumerate(["a", "b", "c"]):
```

```
    print(i, val)
```

Sortie :

```
0 a
1 b
2 c
```

d. Boucle while

Répète les instructions **tant qu'une condition est vraie**.

Exemple :

```
x = 0
while x < 5:
    print("x =", x)
    x += 1
```

Sortie :

```
x = 0
x = 1
x = 2
x = 3
x = 4
```

e. Instructions de contrôle dans les boucles

Instruction	Description
break	Interrompt la boucle
continue	Passe à l'itération suivante
pass	Ne fait rien (utilisé comme placeholder)

Exemple :

```
for i in range(10):
    if i == 5:
        break
    if i % 2 == 0:
        continue
    print(i)
```

Sortie :

```
1
3
```

f. Boucles imbriquées

Exemple :

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Sortie:

```
i=0, j=0
i=0, j=1
i=1, j=0
i=1, j=1
i=2, j=0
i=2, j=1
```

8.3. Fonctions utiles avec les boucles

Fonction	Description	Exemple
range(n)	Génère une séquence d'entiers de 0 à n-1	range(5)
len(seq)	Retourne la longueur d'une séquence	len("Python")
enumerate(seq)	Donne indice + valeur	enumerate(["a","b"])
zip(a,b)	Parcourt deux séquences simultanément	zip([1,2],[3,4])

Exemple :

```
noms = ["Ali", "Mina"]
notes = [15, 18]
for nom, note in zip(noms, notes):
    print(nom, ":", note)
```

Sortie

```
Ali : 15
Mina : 18
```

9. Les fonctions

Une fonction est un bloc d'instructions qui réalise une tâche spécifique. Elle peut recevoir des paramètres et retourner une valeur**.** L'objectif principal est d'éviter la répétition du code et d'améliorer la lisibilité**.**

Syntaxe générale

```
def nom_de_fonction(param1, param2, ...):
    """Documentation éventuelle"""
```

```
    bloc_d_instructions
    return valeur
```

- def : mot-clé pour définir une fonction
- nom_de_fonction : identifiant choisi par le programmeur
- paramètres : données d'entrée (facultatives)
- return : valeur renvoyée (facultative)

Exemple simple

```
def saluer():
    print("Bonjour et bienvenue en Python !")
saluer()
```

Sortie :

```
Bonjour et bienvenue en Python !
```

Exemple avec paramètres

```
def bonjour(nom):
    print("Bonjour", nom)
bonjour("Ali")
bonjour("Mina")
```

Sortie :

```
Bonjour Ali
Bonjour Mina
```

Exemple avec retour de valeur

```
def somme(a, b):
    return a + b
resultat = somme(5, 3)
print("Résultat :", resultat)
```

Sortie :

```
Résultat : 8
```

9.1.Les types de paramètres

Python offre plusieurs façons de passer des arguments à une fonction.

Type de paramètre	Exemple	Description
Positionnel	somme(3, 4)	Les valeurs sont passées selon l'ordre
Nomé	somme(a=3, b=4)	Les valeurs sont identifiées par le nom
Valeur par défaut	def somme(a, b=5)	Si b n'est pas fourni, prend 5
Variable (*args)	def f(*args)	Reçoit un nombre variable d'arguments
Variable clé (**kwargs)	def f(**kwargs)	Reçoit des arguments nommés variables

Remarques

a. args = "arguments" : permet de recevoir plusieurs arguments positionnels (sans nom).

b. kwargs = "keyword arguments" (arguments à mots-clés) : permet de recevoir plusieurs arguments nommés (avec un nom).

c. Python regroupe les arguments de cette façon :

- Tous les arguments sans nom vont dans args (sous forme de tuple)
- Tous les arguments avec nom vont dans kwargs (sous forme de dictionnaire)

Exemple de valeur par défaut

```
def puissance(x, n=2):
    return x ** n
print(puissance(3))    # carré
print(puissance(3, 3)) # cube
```

Sortie :

```
9
27
```

Exemple avec *args et **kwargs

```
def infos(*args, **kwargs):
    print("Arguments positionnels :", args)
    print("Arguments nommés :", kwargs)
infos(1, 2, 3, nom="Ali", age=22)
```

Sortie :

```
Arguments positionnels : (1, 2, 3)
Arguments nommés : {'nom': 'Ali', 'age': 22}
```

9.2. Les variables locales et globales

Une variable définie dans une fonction est locale à celle-ci. Une variable définie hors des fonctions est globale**.**

Exemple :

```
x = 10   # variable globale
def afficher():
    x = 5   # variable locale
    print("x local =", x)
afficher()
print("x global =", x)
```

Sortie :

```
x local = 5
x global = 10
```

⚠ Pour modifier une variable globale à l'intérieur d'une fonction, on doit utiliser le mot-clé global.

Exemple :

```
x = 10
def incrementer():
    global x
    x += 1
incrementer()
print(x)
```

Sortie :

```
11
```

9.3. Fonctions anonymes (lambda)

Les fonctions lambda sont des fonctions courtes et sans nom**.**

Syntaxe :

lambda arguments : expression

Exemple :

```
carre = lambda x: x**2
print(carre(4))
```

Cette ligne équivaut exactement à :

```
def carre(x):
    return x ** 2
```

Sortie :

```
16
```

Souvent utilisées avec :

- map():map() applique une fonction à chaque élément d'un itérable (liste, tuple, etc.) et renvoie un nouvel itérateur (qu'on peut convertir en liste).
- filter():filter() filtre les éléments d'un itérable selon une condition (booléenne) :il garde seulement ceux pour lesquels la fonction renvoie True.
- sorted() :sorted() trie un itérable et renvoie une nouvelle liste triée (sans modifier l'original).

Exemple 1:

```
liste = [1, 2, 3, 4, 5]
carres = list(map(lambda x: x**2, liste))
print(carres)
```

Sortie :

```
[1, 4, 9, 16, 25]
```

Ce code équivaut exactement à :

```
liste = [1, 2, 3, 4, 5]
carres = [x**2 for x in liste]
print(carres)
```

Sortie :

```
[1, 4, 9, 16, 25]
```

Exemple 2

```
nombres = [1, 2, 3, 4, 5, 6]
pairs = filter(lambda x: x % 2 == 0, nombres)
print(pairs)  #  Ce n'est pas le contenu, mais l'adresse mémoire de l'itérateur.
#Pour afficher les valeurs, il faut convertir l'itérateur en liste :
pairs = list(filter(lambda x: x % 2 == 0, nombres))
print(pairs)
```

Sortie :

```
<filter object at 0x0000000003085FD0>
[2, 4, 6]
```

Exemple 3

```
nombres = [5, 2, 9, 1]
print(sorted(nombres))
```

Sortie :

```
[2, 4, 6]
[1, 2, 5, 9]
```

9.4. Documentation des fonctions

Une bonne pratique consiste à documenter ses fonctions avec une **docstring**. Une docstring (abréviation de "documentation string") est une chaîne de caractères placée au tout début d'une fonction, d'une classe ou d'un module, pour décrire ce qu'elle fait**.** C'est une forme de commentaire interne**, mais** structurée **et** reconnue par Python**.**

Exemple :

```
def addition(a, b):
    """Retourne la somme de a et b."""
    return a + b
help(addition)
```

- Les trois guillemets `"""  . . .  """` permettent d'écrire une docstring.
- Python enregistre cette chaîne et la rend accessible avec help() ou .__doc__.

Sortie :

```
Help on function addition in module __main__:
addition(a, b)
    Retourne la somme de a et b.
```

10. La récursivité

Une fonction récursive est une fonction qui s'appelle elle-même pour résoudre un sous-problème plus petit du problème initial. Chaque appel crée une nouvelle instance de la fonction avec ses propres variables locales. Pour éviter une boucle infinie, une fonction récursive doit comporter deux parties essentielles :

1. **Cas de base (condition d'arrêt)** → définit quand la récursion s'arrête.
2. **Appel récursif** → la fonction s'appelle elle-même sur un sous-problème plus simple.

10.1. Exemple simple : décompte

```
def compte_a_rebours(n):
    if n == 0:  # Cas de base
        print("Décollage ")
    else:
        print(n)
        compte_a_rebours(n - 1)  # Appel récursif
compte_a_rebours(5)
```

Sortie :

```
5
4
3
2
1
Décollage
```

10.2. Exemple : Factorielle

La **factorielle** d'un entier n, notée n!, est définie comme :

$n!=n\times(n-1)\times(n-2)\times...\times1$

et par convention, $0!=1$.

Code

```
def factorielle(n):
    if n == 0:
        return 1
    else:
```

```
        return n * factorielle(n - 1)
print(factorielle(5))
```

Ce code est équivalent à :

```
def fact(n):
    """Retourne la factorielle de n."""
    return 1 if n == 0 else n * fact(n - 1)

print(fact(5))  # 120
```

Sortie :

```
120
```

Déroulement de l'exécution

factorielle(5)
= 5 * factorielle(4)
= 5 * 4 * factorielle(3)
= 5 * 4 * 3 * factorielle(2)
= 5 * 4 * 3 * 2 * factorielle(1)
= 5 * 4 * 3 * 2 * 1 * factorielle(0)
= 5 * 4 * 3 * 2 * 1 * 1
= 120

10.3. Exemple: Suite de Fibonacci

La **suite de Fibonacci** est définie par :

$F(0)=0, \quad F(1)=1$

$F(n)=F(n-1)+F(n-2)$

Code

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
for i in range(10):
    print(fibonacci(i), end=" ")
```

Sortie :

0 1 1 2 3 5 8 13 21 34

10.4. Somme d'une liste (exemple récursif)

Exemple :

```
def somme_liste(L):
    if len(L) == 0:
        return 0
    else:
        return L[0] + somme_liste(L[1:])
print(somme_liste([2, 4, 6, 8]))
```

Sortie :

20

10.5. Récursivité vs Itération

Aspect	Récursivité	Itération
Mécanisme	La fonction s'appelle elle-même	Utilise une boucle (for, while)
Lisibilité	Plus naturelle pour problèmes hiérarchiques	Plus efficace pour calculs simples
Consommation mémoire	Plus élevée (pile d'appels)	Faible
Risque	Dépassement de la pile (RecursionError)	Aucun

Exemple comparatif : somme des entiers 1 à n

Version récursive :

```
def somme_rec(n):
    if n == 0:
        return 0
    else:
        return n + somme_rec(n - 1)
```

Version iterative:

```
def somme_iter(n):
    s = 0
    for i in range(n + 1):
        s += i
```

```
return s
```

10.6. Limite de récursion

Par défaut, Python limite la profondeur de récursion à environ 1000 appels imbriqués pour éviter un débordement de pile.

Exemple :

```
import sys
print(sys.getrecursionlimit())
```

Sortie :

```
1000
```

a. import sys

- Le module sys (abréviation de *system*) donne accès à certaines fonctions et variables internes de l'interpréteur Python.
- Il permet par exemple de :
  - gérer les arguments de la ligne de commande,
  - interagir avec la sortie ou l'entrée standard,
  - contrôler des paramètres internes du moteur Python.

b. sys.getrecursionlimit()

Cette fonction renvoie la profondeur maximale de récursion autorisée par Python. Autrement dit : combien de fois une fonction peut s'appeler elle-même avant de provoquer une erreur.

Par défaut, cette limite est généralement 1000.

Cette fonction retourne la profondeur maximale de récursion autorisée par Python. C'est-à-dire : le nombre maximal d'appels de fonction imbriqués qu'un programme peut effectuer avant de provoquer une erreur (RecursionError).

Pourquoi cette limite existe-t-elle ?

En Python, chaque appel de fonction consomme un peu de **mémoire (pile d'exécution)**. Si une fonction récursive s'appelle indéfiniment, elle remplirait la mémoire du système → plantage.

On peut (avec prudence) modifier cette limite :

```
import sys
sys.setrecursionlimit(2000)
print(sys.getrecursionlimit())
```

Sortie

```
2000
```

11. Les modules et les paquets

11.1. Qu'est-ce qu'un module ?

Un module est simplement un **fichier Python (.py)** contenant des fonctions, classes ou variables que l'on peut importer dans un autre programme. L'idée est de réutiliser du code sans le recopier.

Exemple : un module simple

Crée un fichier appelé mon_module.py :

```
# mon_module.py
def saluer(nom):
    print(f"Bonjour, {nom} !")

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

PI = 3.14159
```

Utilisation du module

Dans un autre fichier (ex. main.py) :

```
import mon_module
mon_module.saluer("Ali")
print("Résultat :", mon_module.addition(3, 5))
print("Valeur de PI :", mon_module.PI)
```

Sortie :

```
Bonjour, Ali !
Résultat : 8
Valeur de PI : 3.14159
```

11.2. Différentes façons d'importer

Syntaxe	Description	Exemple
import module	Importe tout le module	import math
from module import fonction	Importe une fonction spécifique	from math import sqrt
from module import *	Importe tout (déconseillé)	from math import *
import module as alias	Donne un alias au module	import numpy as np

Exemple1

```
import math
print(math.sqrt(16))
print(math.pi)
```

Sortie

```
4.0
3.141592653589793
```

Exemple2

```
from math import sqrt, pi
print(sqrt(25))
print(pi)
```

Sortie

```
5.0
3.141592653589793
```

Exemple3

Importation sans alias :

```
import math
print(math.sin(math.pi / 2))
```

Sortie

```
1.0
```

Importation avec alias :

```
import math as m
print(m.sin(m.pi / 2))
```

Sortie

```
1.0
```

11.3. Modules intégrés de Python

Python possède de nombreux modules standards livrés avec l'interpréteur.

Quelques exemples utiles :

Module	**Utilisation**	**Exemple**
math	Fonctions mathématiques	math.sqrt(9)
random	Génération de nombres aléatoires	random.randint(1,10)
datetime	Gestion des dates et heures	datetime.datetime.now()
os	Interaction avec le système d'exploitation	os.listdir()
sys	Gestion de l'interpréteur	sys.exit()
statistics	Moyennes, médianes, etc.	statistics.mean([1,2,3])

Exemple : module random

```
import random
print(random.randint(1, 6))       # entier aléatoire
print(random.choice(["pile", "face"]))  # choix aléatoire
print(random.random())          # nombre flottant [0,1)
```

Sortie

```
6
face
0.10037068240440172
```

Ou

```
1
face
0.6932150549320091
```

Ou

```
2
pile
0.9820948878478784
```

Ou

```
1
face
0.23676559322097024
```

…..etc

11.4. Création de ses propres modules

Créer un module consiste simplement à écrire ton code Python dans un fichier .py.

Exemple :

```
# outils_math.py
def carre(x):
    return x ** 2

def cube(x):
    return x ** 3
```

Puis, dans ton programme principal :

```
import outils_math
```

```
print(outils_math.carre(5))
print(outils_math.cube(3))
```

Sortie :

```
25
27
```

11.5. Le répertoire __pycache__

Lorsque Python importe un module, il crée un fichier compilé (.pyc) dans un dossier __pycache__ pour accélérer les futurs chargements. C'est un comportement normal.

Quand tu exécutes un programme Python qui importe un module, Python ne lit pas le fichier `.py` à chaque fois depuis zéro. Il compile d'abord ton fichier `.py` (le code source) en bytecode — un format intermédiaire plus rapide à exécuter. Ce bytecode est ensuite enregistré automatiquement dans un dossier nommé : __pycache__. C'est à dire ce dossier contient des fichiers **.pyc**, c'est-à-dire des fichiers Python compilés.

C'est une optimisation automatique du moteur Python :

- La première fois, Python lit et compile ton code en bytecode.
- Les fois suivantes, il recharge directement le fichier `.pyc`, ce qui accélère considérablement le démarrage **des** programmes.

Autrement dit :

Le dossier `__pycache__` sert à garder une version compilée du code pour éviter de recompiler à chaque import**.**

11.6. Les paquets Python

Un paquet (package) est un ensemble de modules organisés dans un répertoire**.** Un paquet contient :

- Un dossier (nom du paquet)
- Un fichier spécial __init__.py (même vide)
- Plusieurs modules .py

Exemple de structure

```
mon_projet/
│
├── mon_package/
│   ├── __init__.py
│   ├── calculs.py
│   └── utils.py
│
```

└── main.py

Exemple

calculs.py :

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

utils.py :

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

main.py :

```
from mon_package.calculs import addition
from mon_package.utils import saluer

print(addition(2, 3))
saluer("Fatima")
```

Sortie :

```
5
Bonjour, Fatima !
```

11.7. Installation de paquets externes

Pour installer des bibliothèques externes (comme numpy, matplotlib, pandas, etc.), on utilise l'outil **pip** :

```
pip install numpy
pip install matplotlib
```

Une fois installées, tu peux les importer :

```
import numpy as np
import matplotlib.pyplot as plt
```

12. Les tableaux NumPy

NumPy (*Numerical Python*) est une bibliothèque Python conçue pour le **calcul scientifique**. Elle permet de manipuler efficacement de grands ensembles de données sous forme de **tableaux multidimensionnels (arrays)**.

Ses points forts :

- Rapidité (implémentation en C)
- Calcul vectorisé (sans boucles)
- Outils mathématiques puissants
- Support des matrices, tenseurs, FFT, algèbre linéaire, etc.

12.1. Installation

Si NumPy n'est pas déjà installé :

```
pip install numpy
```

Puis, dans ton programme :

```
import numpy as np
```

Convention : on utilise presque toujours l'abréviation np.

12.2. Création de tableaux

À partir d'une liste

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
print(type(a))
```

Sortie :

```
[1 2 3 4]
<class 'numpy.ndarray'>
```

Nous avons créé un tableau NumPy (ou *ndarray*). Contrairement à une liste Python, ce tableau :

- Est plus rapide**,**
- Consomme moins de mémoire**,**
- Et permet des opérations vectorisées (sur tous les éléments à la fois).

Tableaux multidimensionnels

```
b = np.array([[1, 2, 3], [4, 5, 6]])
print(b)
```

Sortie :

```
[[1 2 3]
 [4 5 6]]
```

12.3. Création rapide avec fonctions NumPy

Fonction	**Description**	**Exemple**
np.zeros(n)	tableau rempli de 0	np.zeros(5)
np.ones(n)	tableau rempli de 1	np.ones((2,3))
np.arange(start, stop, step)	suite d'entiers	np.arange(0, 10, 2)
np.linspace(start, stop, n)	n valeurs également espacées	np.linspace(0, 1, 5)
np.eye(n)	matrice identité	np.eye(3)
np.random.rand(n)	valeurs aléatoires	np.random.rand(4)

12.4. Propriétés des tableaux

```
a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.ndim)   # nombre de dimensions
print(a.shape)  # forme (lignes, colonnes)
print(a.size)   # nombre total d'éléments
print(a.dtype)  # type des éléments
```

Sortie :

```
2
(2, 3)
6
int64
```

12.5. Indexation et slicing

```
a = np.array([[10, 20, 30],
              [40, 50, 60],
              [70, 80, 90]])

print(a[0, 1])     # élément ligne 0, colonne 1 → 20
print(a[1])        # ligne 1 complète
print(a[:, 2])     # toutes les lignes, colonne 2
print(a[0:2, 1:3])  # sous-tableau
```

Sortie :

```
20
[40 50 60]
[30 60 90]
[[20 30]
 [50 60]]
```

12.6. Opérations vectorisées

Les opérations sont effectuées élément par élément**,** beaucoup plus rapidement que les boucles classiques.

```
x = np.array([1, 2, 3, 4])
y = np.array([10, 20, 30, 40])
```

```
print(x + y)
print(x * y)
print(x ** 2)
print(np.sqrt(x))
```

Sortie :

```
[11 22 33 44]
[ 10  40  90 160]
[ 1  4  9 16]
[1.         1.41421356 1.73205081 2.        ]
```

12.7. Fonctions mathématiques courantes

```
a = np.array([1, 2, 3, 4, 5])

print(np.mean(a))    # moyenne
print(np.median(a))  # médiane
print(np.std(a))     # écart-type
print(np.sum(a))     # somme
print(np.min(a))     # minimum
print(np.max(a))     # maximum
```

12.8. Produits matriciels

a. Produit scalaire

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.dot(a, b))
```

Sortie:

```
32
```

b. Produit de matrices

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(A @ B)
```

Sortie :

```
[[19 22]
 [43 50]]
```

12.9. Reshape et Transposition

La méthode .reshape((lignes, colonnes)) permet de **changer la forme** du tableau sans modifier ses données.

Exemple :

```
a = np.arange(6)
b = a.reshape((2, 3))
print(b)

print(b.T)  # transposée
```

Sortie :

```
[[0 1 2]
 [3 4 5]]
[[0 3]
 [1 4]
 [2 5]]
```

12.10. Filtres et conditions logiques

```
a = np.array([10, 15, 20, 25, 30])
print(a[a > 20])  # filtre
```

Sortie :

```
[25 30]
```

12.11. Sauvegarde et chargement de tableaux

Sauvegarde au format NumPy :

```
np.save('donnees.npy', a)
```

Chargement :

```
b = np.load('donnees.npy')
print(b)
```