Mémento Python

Première et terminale NSI

① Variables et types

```
age = 17              # affectation
nom = "Ada"           # minuscules, sans accent
age = age + 1         # mise à jour
a, b = b, a           # échange en une ligne
```

Type	Exemple	Description
`int`	`42`, `-7`	entier, taille illimitée
`float`	`3.14`, `2e-3`	flottant (approché)
`bool`	`True`	booléen
`str`	`"NSI"`, `'a'`	chaîne, immuable
`list`	`[1, 2, 3]`	liste, mutable
`tuple`	`(1, 2)`	p-uplet, immuable
`dict`	`{"a": 1}`	dictionnaire, mutable
`NoneType`	`None`	absence de valeur

```
type(3.0)          # <class 'float'>
int("12") + 1      # 13
float("2.5")       # 2.5
str(7) + "h"       # '7h'
int(3.99)          # 3 (troncature)
round(2.675, 2)    # 2.67 (flottant approché)
bool(0), bool(""), bool([])  # False x3
```

Toute valeur non nulle, toute chaîne ou liste non vide vaut `True` dans un test.

② Opérateurs

Opérateur	Sens	Exemple
`+ - *`	somme, différence, produit	`7 * 3` → `21`
`/`	division réelle	`7 / 2` → `3.5`
`//`	quotient entier	`7 // 2` → `3`
`%`	reste (modulo)	`7 % 2` → `1`
`**`	puissance	`2 ** 10` → `1024`
`== !=`	égal, différent	`3 == 3.0` → `True`
`< <= > >=`	comparaisons	`"a" < "b"` → `True`
`and or not`	logique	`not (a and b)`
`in`	appartenance	`"s" in "nsi"` → `True`
`is`	même objet	`x is None`

Priorités, de la plus forte à la plus faible : `**`, puis `* / // %`, puis `+ -`, puis les comparaisons, puis `not`, `and`, `or`. En cas de doute, on met des parenthèses.

```
-7 // 2     # -4 (arrondi vers le bas)
-7 % 2      # 1  (reste toujours positif)
0.1 + 0.2 == 0.3            # False !
abs(0.1 + 0.2 - 0.3) < 1e-9  # True
x = 5 ; x += 2 ; x *= 3      # x vaut 21
```

Les opérateurs `and` et `or` sont paresseux : `n != 0 and 10 / n > 1` ne divise jamais par zéro.

③ Entrées et sorties

```
nom = input("Prénom ? ")   # toujours une chaîne
age = int(input("Âge ? ")) # conversion
    obligatoire
print("Bonjour", nom, age) # séparés par un
    espace
print("a", "b", sep="-", end="")  # a-b
print(f"{nom} a {age} ans")   # chaîne formatée
print(f"{3.14159:.2f}")       # 3.14
print(f"{7:03d}")             # 007
print(f"{255:b} {255:x}")     # 11111111 ff
```

④ Conditions

```
if note >= 16:
    mention = "TB"
elif note >= 14:
    mention = "B"
elif note >= 12:
    mention = "AB"
else:
    mention = ""
# expression conditionnelle
parite = "pair" if n % 2 == 0 else "impair"
```

Le bloc est délimité par l'indentation (quatre espaces). Les comparaisons s'enchaînent : `0 <= x < 10`.

⑤ Boucles

```
for i in range(5):          # 0, 1, 2, 3, 4
    print(i)
for i in range(2, 10, 3):   # 2, 5, 8
    print(i)
for i in range(10, 0, -1):  # 10, 9, ..., 1
    print(i)
for lettre in "nsi":        # parcours direct
    print(lettre)
for i, v in enumerate([8, 5]):  # (0,8) (1,5)
    print(i, v)
for a, b in zip([1, 2], "xy"):  # (1,x) (2,y)
    print(a, b)
```

```
n = 27
etapes = 0
while n != 1:
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3 * n + 1
    etapes += 1
```

`range(a, b)` va de `a` inclus à `b` exclu. `break` sort de la boucle, `continue` passe au tour suivant. Une boucle `while` doit faire évoluer sa condition, sinon elle ne termine pas.

6 Fonctions

```
def moyenne(notes, coef=1):
    """Renvoie la moyenne des notes.
    notes : liste de flottants non vide."""
    assert len(notes) > 0, "liste vide"
    total = 0
    for x in notes:
        total += x
    return total / len(notes) * coef

m = moyenne([12, 15])          # 13.5
m = moyenne([12, 15], coef=2)  # 27.0
```

- `return` interrompt la fonction et renvoie une valeur ; sans `return`, la fonction renvoie `None`.
- On peut renvoyer plusieurs valeurs : `return q, r` puis `q, r = division(17, 5)`.
- Les variables créées dans la fonction sont locales. Une variable globale est lisible dans la fonction, mais on évite de la modifier.
- Une liste passée en paramètre est partagée : la modifier dans la fonction la modifie aussi à l'extérieur.
- La docstring décrit le rôle, les paramètres et la valeur renvoyée. Les `assert` en début de fonction vérifient les préconditions.

```
def test_moyenne():
    assert moyenne([10, 20]) == 15
    assert moyenne([7]) == 7
```

Les sujets d'épreuve pratique annotent souvent les types, sans effet à l'exécution : `def moyenne(notes: list, coef: int = 1) -> float:`.

7 Chaînes de caractères

```
s = "Bonjour"
len(s)          # 7
s[0], s[-1]     # 'B', 'r'
s[0:3]          # 'Bon' (indice 3 exclu)
s[3:]           # 'jour'
s[::-1]         # 'ruojnoB' (renversée)
s + " !"        # concaténation
"ab" * 3        # 'ababab'
"o" in s        # True
```

Méthode	Résultat (nouvelle chaîne)
`s.upper()`, `s.lower()`	majuscules, minuscules
`s.capitalize()`	première lettre en majuscule
`s.strip()`	sans les espaces aux extrémités
`s.replace("o", "0")`	remplace toutes les occurrences
`s.split(";")`	liste des morceaux
`";".join(liste)`	chaîne à partir d'une liste
`s.find("jour")`	indice de la première occurrence, sinon `-1`
`s.count("o")`	nombre d'occurrences
`s.startswith("Bon")`	booléen
`s.isdigit()`, `s.isalpha()`	que des chiffres, que des lettres
`ord("A")`, `chr(66)`	`65`, `'B'` (code Unicode)

Une chaîne est immuable : `s[0] = "b"` provoque une erreur. On construit une nouvelle chaîne.

8 Listes

```
L = [3, 1, 4]
L[0], L[-1]     # 3, 4
L[1:3]          # [1, 4]
len(L)          # 3
L.append(1)     # [3, 1, 4, 1]
L.insert(0, 9)  # [9, 3, 1, 4, 1]
L.pop()         # renvoie 1 -> [9, 3, 1, 4]
L.pop(0)        # renvoie 9 -> [3, 1, 4]
L.remove(4)     # première occurrence -> [3, 1]
L.index(3)      # 0
L.count(1)      # 1
L.sort()        # sur place -> [1, 3]
sorted(L)       # nouvelle liste triée
L.reverse()     # sur place -> [3, 1]
L + [7, 8]      # nouvelle liste [3, 1, 7, 8]
[0] * 5         # [0, 0, 0, 0, 0]
sum(L), max(L), min(L)   # 4, 3, 1
```

```
# constructions par compréhension
carres = [x ** 2 for x in range(6)]
pairs = [x for x in L if x % 2 == 0]
grille = [[0] * 4 for _ in range(3)]
grille[1][2] = 5   # ligne 1, colonne 2
```

```
# pièges des listes
A = [1, 2, 3]
B = A           # même liste (alias)
B.append(4)     # A vaut aussi [1, 2, 3, 4]
C = list(A)     # vraie copie (ou A[:])
M = [[0] * 3] * 2  # 2 fois LA MÊME ligne
```

9 Tuples et dictionnaires

```
p = (3, 4)      # p-uplet, immuable
x, y = p        # déballage
p[0]            # 3
couleurs = [("rouge", 255), ("vert", 0)]
for nom, valeur in couleurs:
    print(nom, valeur)
```

```
d = {"nom": "Ada", "age": 36}
d["age"]                # 36
d["ville"] = "Londres"  # ajout ou modification
d.get("pays", "?")      # '?' si la clé manque
"nom" in d              # True (sur les clés)
del d["age"]
len(d)
for cle in d:               # parcours des clés
    print(cle, d[cle])
for cle, val in d.items():  # couples
    print(cle, val)
list(d.keys()), list(d.values())
```

Les clés sont uniques et immuables (chaînes, entiers, tuples). Un dictionnaire sert de table d'association : compter des occurrences, décrire un objet, mémoriser des résultats.

```
effectifs = {}
for lettre in "abracadabra":
    n = effectifs.get(lettre, 0)
    effectifs[lettre] = n + 1
# {'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

10 Modules utiles

```
from math import sqrt, pi, floor, ceil, gcd
sqrt(2), floor(2.7), ceil(2.1), gcd(12, 18)
from random import randint, random
from random import choice, shuffle
randint(1, 6)       # entier de 1 à 6 inclus
random()            # flottant dans [0, 1[
choice(["a", "b"])  # élément au hasard
shuffle(L)          # mélange L sur place
from time import perf_counter
debut = perf_counter()
# ... calcul ...
duree = perf_counter() - debut
```

11 Fichiers

```
with open("notes.txt", encoding="utf-8") as f:
    lignes = f.readlines()  # liste des lignes
with open("sortie.txt", "w") as f:
    f.write("Bonjour\n")
import csv
with open("eleves.csv", newline="") as f:
    lecteur = csv.DictReader(f, delimiter=";")
    table = list(lecteur)
# table : une liste de dictionnaires
```

12 Erreurs et mise au point

Message	Cause fréquente
`SyntaxError`	deux-points, parenthèse ou guillemet oublié
`IndentationError`	bloc mal indenté
`NameError`	variable jamais définie ou mal orthographiée
`TypeError`	types incompatibles, `"a" + 1`
`ValueError`	valeur inadaptée, `int("abc")`
`IndexError`	indice hors de la liste
`KeyError`	clé absente du dictionnaire
`ZeroDivisionError`	division par zéro
`AttributeError`	méthode inconnue, `L.push(1)`
`ModuleNotFoundError`	module absent ou mal orthographié
`RecursionError`	récursion sans cas de base

```
try:
    n = int(input("Entier ? "))
except ValueError:
    print("Ce n'est pas un entier")
```

Pour comprendre un programme : lire l'erreur de bas en haut, afficher les variables avec `print`, tester les fonctions une par une, écrire des `assert`.

13 Récursivité

```
def factorielle(n):
    if n == 0:      # cas de base
        return 1
    return n * factorielle(n - 1)

def somme(L):
    if L == []:
        return 0
    return L[0] + somme(L[1:])
```

Une fonction récursive a toujours au moins un cas de base, et chaque appel se rapproche de ce cas. La profondeur est limitée (environ mille appels).

14 Programmation orientée objet

```
class Compte:
    def __init__(self, titulaire, solde=0):
        self.titulaire = titulaire  # attribut
        self.solde = solde

    def deposer(self, montant):     # méthode
        self.solde += montant

    def __str__(self):
        return f"{self.titulaire} : {self.solde}"


c = Compte("Ada", 100)  # instanciation
c.deposer(50)
print(c)                # Ada : 150
c.solde                 # 150
```

Vocabulaire : la **classe** est le modèle, `c` en est une **instance** ; `titulaire` et `solde` sont ses **attributs**, `deposer` une **méthode**. `self` désigne l'objet courant, passé automatiquement à l'appel `c.deposer(50)`. `__init__` est le constructeur, `__str__` définit l'affichage par `print`.

15 Fonctions natives à connaître

Fonction	Rôle
`len(x)`	longueur d'une chaîne, liste, dictionnaire
`abs(x)`, `round(x, n)`	valeur absolue, arrondi
`min`, `max`, `sum`	sur une liste ou plusieurs arguments
`sorted(L)`, `reversed(L)`	copie triée, parcours à l'envers
`range(a, b, pas)`	suite d'entiers
`enumerate(L)`, `zip(A, B)`	indices et valeurs, parcours parallèle
`isinstance(x, int)`	test de type
`bin(10)`, `hex(255)`	`'0b1010'`, `'0xff'`
`int("1010", 2)`	`10` (lecture en base 2)
`divmod(17, 5)`	`(3, 2)`
`help(fonction)`	documentation intégrée