Synthèse : Variables, nombres, listes et chaînes de caractères.

Variables et syntaxe de base

- Une **variable** est un espace mémoire nommé où l'on stocke une **valeur**.
- Un **nom de variable** peut contenir des lettres, chiffres (sauf au début) ou _ : `age, mon_age, dim3, faire_affichage…`
- **Affectation/Déclaration** de variable : `age = 18`
- **Afficher** : `print("Mon age est", age, "ans.")`
- **Type** d'une valeur (les variables n'ont pas de type) :
  `type(2.0), type('Bonjour'), type(dim3)`
- Une **instruction** par ligne
- Les **commentaires** commencent par un # et ne sont pas exécutés
  `# Ceci n'a aucune influence sur le programme`

Nombres (entiers ou en virgule flottante)

- **18** ou **-5** sont des entiers (**int**) ; **3.5, 1e15** ou **1.0** sont des « réels » ou plutôt des nombres en virgule flottante (**float**).
- Les opérations respectent les priorités (parenthèses, puissances, multiplication/division, addition/soustraction, de gauche à droite) :

Opération	Symbole	Exemple
Addition/Soustraction	+ ou -	2+16 == 18
Multiplication/Division	* ou /	15 / 6 == 2.5
Exponentiation	**	3**2 == 9
Reste (modulo)	%	15 % 6 == 3
Quotient entier	//	15 // 6 == 2

- Toutes les opérations ont une variante qui modifie la valeur d'une variable :
  `compte += 1 # compte = compte + 1`
- Fonctions et constantes Mathématiques dans le module math :

```
import math
aire_disque = math.pi * rayon**2
distanceAB = math.sqrt( (xB-xA)**2 + (yB-yA)**2 )
```

Listes

- Une **liste** (de type **list**) sert à stocker plusieurs éléments consécutivement, quelque soit leur type, on les note entre crochets :

```
[1, 2, "Bonjour", 42.0]
# 4 éléments : int,int,str,float
```

- On dit que la liste est une **structure de données** (elle contient et organise plusieurs valeurs)
- Chaque élément a un **indice**, le premier élément a l'indice **0**.
- On peut **indexer** la liste avec un indice entre crochet :

```
ma_liste = [1, 2, 3, 4]
print(ma_liste[0]) # ==> 1
print(ma_liste[3]) # ==> 4
print(ma_liste[4]) # Erreur
```

- Les **indices négatifs** commencent à la fin de la liste (par -1) :
  `print(ma_liste[-1]) # ==> 4`
- On peut **modifier un élément** : `ma_liste[2] = "Surprise"`
- Ou **ajouter à la fin** : `ma_liste.append("Nouveau")`
- Deux listes peuvent être **concaténées** (mises bout à bout) :
  `de_1_a_6 = [1,2,3] + [4,5,6]`
- Une liste peut être **répétée** : `[1,2]*3 # [1,2,1,2,1,2]`
- On peut prendre une **tranche** (slice) de la liste :

```
de_1_a_6[2:4] # coupe après 2 et 4 éléments: [3,4]
de_1_a_6[0:5:2] # va de 2 en 2: [1,3,5]
de_1_a_6[-2:] # 2 derniers: [5,6]
```

- Créer une liste de nombre : `range(debut, fin exclue, pas)`

```
range(1,10,2) # [1,3,5,7,9]
range(5) # [0,1,2,3,4]
```

- **Longueur** d'une liste : `len(ma_liste)`

Chaînes de caractères

- Les **chaînes de caractères** (type **str**) ou **string** sont des listes de lettres non-modifiables. Autrement dit du texte.
- Elles peuvent contenir n'importe quel caractère **Unicode**.
- On les note entre guillemets anglais ou apostrophes :

```
"Bonjour" == 'Bonjour'
"C'est pratique !" # notez l'apostrophe
```

- Toutes les opérations sur les listes marchent sur les chaînes sauf les modifications.
- **Demander une chaîne** à l'utilisateur : `input("Entrer un mot")`
- Saut de ligne : **\n** ou mettre la chaîne entre `"""`

Conversions

- `int("18"), float(2), str(3), int(input("Nombre :"))`

Synthèse : Dictionnaires, Tests, Structures de contrôle et Fonctions.

Dictionnaires

- Les **dictionnaires** (type **dict**) permettent d'indexer des valeurs par des chaînes plutôt que des entiers. On les note entre accolades :
```
moyennes = {"maths":12, "physique":13.5, "svt":"non !"}
```
- On peut donc faire : `print(moyennes["maths"])`
- **Modifier ou ajouter un élément** : `moyennes["anglais"] = 13`
- Les index sont appelés les **clés** du dictionnaire : `moyennes.keys()`

Tests et booléens

- Les **booléens** (type **bool**) sont soit vrai (**True**) soit faux (**False**)
- On les obtient généralement comme résultat d'un test

Opération	Symbole	Exemple
Strictement inférieur	<	2 < 16 == True
Inférieur ou égal	<=	3 <= 3 == True
Supérieur	> ou >=	3 > 3 == False
Egal (Attention !)	==	2 == 4 est False
Différent	!=	2 != 4 est True
Est dans	in	2 in [1,2,3] == True

- L'opérateur « in » marche avec les listes, chaînes de caractères ou dictionnaires (seul les clés comptent).
- On peut combiner des booléens avec les opérateurs logiques **et** (**and**), **ou** (**or**), **non** (**not**). Attention le ou n'est pas exclusif.
```
print( not (2 > 3 or 10 < 100) ) # ==> False
```

Conditionnelle

- De la forme **Si … Alors … Sinon si … Sinon**. (« Sinon si » et « Sinon » sont optionnels) :
```
if condition1:
      instruction1
      instruction2
elif condition2:
      instruction3
      instruction4
else:
      instruction5
```
- L'**indentation** regroupe les instructions en blocs : si la condition1 est vraie, les instructions 1 et 2 seront exécutées. Un bloc commence après un **deux-points**

Boucles

- La boucle **Pour** (**for**) permet de répéter des instructions un nombre de fois déterminé avant d'entrer dans la boucle
```
for i in range(10): # répète 10 fois
      instruction1 # i prendra les valeurs de 0 à 9
      instruction2
```
- Elle permet aussi de **parcourir** n'importe quelle liste ou chaîne :
```
for élément in séquence:
      bloc d'instructions
```
- La boucle **Tant que** (**while**) permet de répéter des instructions un nombre de fois indéterminé, tant qu'une condition est vraie
```
while condition:
      instruction tant que condition est vraie
# on sort de la boucle lorsque condition est fausse
```
- Pour sortir d'une boucle prématurément, on peut utiliser le mot clé **break** :
```
while True :
      if input("Quitter ? (oui/non)") == "oui" :
            break
```
- While peut tout faire mais utilisez For lorsque vous parcourez une séquence ou connaissez le nombre de répétitions pour plus de lisibilité.

Fonctions

- Les fonctions sont des morceaux de code nommés pour être réutilisés à plusieurs reprises ou simplement rendre la structure d'un programme plus lisible. On les définit avec le mot-clé **def** :
```
def nom_de_la_fonction():
      """Description renvoyée par help(nom_de_la_fonction)"""
      bloc d'instruction
```
- Définir une fonction n'exécute pas le code. Pour **appeler une fonction** on doit écrire son nom suivi par des parenthèses ouvrantes et fermantes :
`nom_de_la_fonction()`
- La fonction peut avoir des paramètres séparés par des virgules, parfois avec une valeur par défaut, et elle peut renvoyer une valeur avec **return** :
```
def f(x, y=0):
      return x*y
```
- Lorsqu'une fonction avec paramètres est appelée, son code est exécuté avec les paramètres prenant les valeurs passées en paramètre :
```
print( f(5, 10) ) # affiche 50
```
- Les variables sont **locales** à la définition de fonction sauf si on utilise le mot-clé **global**. Attention les listes passées en paramètres peuvent être modifiées.

Compréhension de listes : `[i*i for i in range(100)]`