Séquence et structures de contrôle

1 Les séquences

Les séquences sont une catégorie de type de variable. Elles regroupent notamment les chaînes de caractères et les listes. **Les éléments d'une séquence sont rangés dans un ordre déterminé** et sont repérés par leur indice.

Définition

Séquence

Une séquence est une collection ordonnée de valeurs. En Python, le premier élément d'une séquence possède l'indice 0.

Une séquence est dite **immuable** si les valeurs stockées dans la séquence ne peuvent pas être modifiées après sa création. Dans le cas contraire, on parle de séquence **muable** : il est possible de modifier les valeurs stockées après la déclaration et l'initialisation de la variable.

Type Python	Nature	Exemple
`str`	immuable	`"PCSI"`
`list`	muable	`["PCSI", 2026, True]`
`tuple`[1]	immuable	`("PCSI", 2026, True)`

Tableau 1 – Quelques types de séquences en Python

1.1 Les chaînes de caractères

Une **chaîne de caractères** est une **suite ordonnée de caractères**. Son type Python est `str` (*string of characters*). Une chaîne est délimitée par des apostrophes ou par des guillemets.

```
chaine_vide = ""
classe = "PCSI"
message = 'Bonjour !'
```

Remarque : L'utilisation de `'exemple de chaîne'` ou `"exemple de chaîne"` est identique. Python laisse la possibilité de choisir entre les deux pour prendre en compte le cas où l'un des deux caractères est présent dans la chaîne. En effet, si la valeur de la variable est `l'exemple de chaîne`, il faut utiliser `"l'exemple de chaîne"` car une apostrophe est présente dans la chaîne.

Comment faire si les deux caractères sont présents dans la chaîne (apostrophe et guillemet) ? Il faut alors utiliser un **caractère d'échappement**. Il est introduit par le caractère `'\'` (*backslash* ou *anti-slash*). Le Tableau 2 présente quelques caractères d'échappement courants.

Écriture	Signification	Résultat dans une chaîne
\'	apostrophe	'
\"	guillemet	"
\n	retour à la ligne	passage à la ligne suivante
\t	tabulation	l'équivalent de 4 espaces

Tableau 2 – Quelques caractères d'échappement

Exemple

Caractères spéciaux

Pour insérer une apostrophe et un retour à la ligne :

```
ma_str = 'L\'exemple est sur\ndeux lignes.'
print(ma_str)
```

La console affiche :

```
L'exemple est sur
deux lignes.
```

1.2 Les listes

On peut voir les **listes** comme des **contenants** (des cartons, par exemple) dans lesquels on range des **objets** de manière **ordonnée**. Les types de ces objets ne sont pas forcément identiques au sein d'une même liste. Une liste est **muable** et est caractérisée par les crochets.

```
ma_liste = []
```

Cette instruction déclare une variable nommée `ma_liste`, de type `list`, et vide. *J'ai sorti un carton pour y ranger des objets. Pour l'instant, il est vide mais je le remplirai plus tard.*
On peut bien sûr déclarer des listes non vides :

```
ma_liste = ['PCSI', 0.6, 2026, True]
zeros = [0] * 5
```

Ici, la variable `zeros` contient alors `[0, 0, 0, 0, 0]`.

Attention –

```
maListe = ['PCSI', 0.6, 2026, True]
memeListe = maListe
```

Au contraire des variables déjà vues en cours, `maListe` et `memeListe` ont toutes deux **le même espace mémoire**, ce qui signifie qu'**un changement sur l'une implique un changement sur l'autre**. La section 1.3.2 montre des exemples de comment faire une copie indépendante d'une liste.

1.3 Opérations sur les séquences

Les opérations du Tableau 3 sont communes aux chaînes de caractères et aux listes (entre autres, elles sont aussi valides pour les tuples par exemple). On note `seq` une séquence.

Opération	Syntaxe	Exemple ou résultat
Longueur	`len(seq)`	`len("PCSI")` renvoie
Accès à un élément	`seq[i]`	`"PCSI"[1]` renvoie
Extraction	`seq[debut:fin:pas]`	l'élément d'indice `fin` est exclu
Concaténation	`seq1 + seq2`	crée une nouvelle séquence
Répétition	`seq * n`	répète `seq` `n` fois
Appartenance	`element in seq`	renvoie un booléen
Recherche d'indice	`seq.index(element)`	indice de la première occurrence

Tableau 3 – Opérations usuelles sur les séquences

Question 1 : Pour la séquence `seq = 'PCSI'` que renvoie `seq[0:2]` ? Cette instruction est équivalente à `seq[0:2:1]` ; que renvoie `seq[0:4:2]` ?

1.3.1 Indices et extraction

Comme énoncé précedemment, l'**indice du premier élément est** 0. Les indices négatifs permettent de compter depuis la fin : `-1` désigne le dernier élément, `-2` l'avant-dernier, etc.

Pour une extraction `seq[debut:fin:pas]`, les valeurs non spécifiée prennent des valeurs par défaut : le début ou la fin de la séquence, et un pas égal à 1. **L'élément d'indice `fin` n'est jamais inclus.**

Exemple

Extraction dans une liste

```
ma_liste = ['PCSI', 0.6, 2026, True]
sous_liste = ma_liste[1:3]
dernier = ma_liste[-1]
```

La valeur de `sous_liste` est `[0.6, 2026]` et celle de `dernier` est `True`.

Question 2 : On considère la chaîne `mot = "Python"`. Déterminer les valeurs de `mot[0]`, `mot[-1]`, `mot[1:4]`, `mot[:2]` et `mot[::2]`.

Question 3 : Compléter le tableau suivant.

Expression	Valeur renvoyée
`"PC" + "SI"`	
`[1, 2] * 2`	
`3 in [1, 2, 3]`	
`"a" in "Python"`	
`"abracadabra".index("a")`	

1.3.2 Opérations spécifiques aux listes

Contrairement à une chaîne de caractères, **une liste est muable (mutable)** : ses éléments peuvent être remplacés, ajoutés ou supprimés. Le Tableau 4 présente les opérations et méthodes principales des listes. (Les méthodes sont des fonctions réservées à un type d'objet, en l'occurrence ici aux listes. Elles sont présentées à la section 3.4.)

Action	Syntaxe	Effet
Modifier un élément	`liste[i] = valeur`	remplace l'élément d'indice `i`
Ajouter à la fin	`liste.append(x)`	ajoute l'élément `x`
Ajouter plusieurs éléments	`liste.extend(seq)`	ajoute les éléments de `seq`
Insérer	`liste.insert(i, x)`	insère `x` à l'indice `i`
Retirer par valeur	`liste.remove(x)`	retire la **première** occurrence de `x`
Retirer par indice	`liste.pop(i)`	retire et renvoie l'élément d'indice `i` (-1 par défaut)
Supprimer	`del liste[i]`	supprime l'élément d'indice `i`
Inverser	`liste.reverse()`	inverse les index de la liste

Tableau 4 – Principales opérations spécifiques aux listes

Deux constructions permettent de créer simplement une copie indépendante d'une liste :

```
copie_1 = ma_liste[:]
copie_2 = list(ma_liste)
```

Remarque : La fonction `list()` est ce qu'on appelle un constructeur. Les constructeurs sont des fonctions qui permettent de construire les objets ; il y a bien cette idée de création d'une nouvelle variable, donc d'un espace mémoire dédié.

Question 4 : Déterminer la valeur de `liste` à la fin du programme suivant.

```
liste = [1, 2, 3]
liste.append(4)
liste.insert(1, 0)
liste.pop(3)
liste.reverse()
```

1.4 D'autres collections de données

Nous verrons d'autres collections (toutes des séquences, sauf les dictionnaires) plus tard :

- les **tuples**, séquences immuables délimitées par des parenthèses ;
- les **dictionnaires**, qui associent des clés à des valeurs ;
- les tableaux `array` du module `numpy`, notamment adaptés au calcul numérique ;
- les **piles** et les **files**, qui imposent un ordre particulier pour ajouter et retirer les données.

2 Structures de contrôle

Définition

Structure de contrôle

Une structure de contrôle permet de modifier le déroulement séquentiel d'un programme, par exemple en faisant des retours ou des sauts dans les lignes du programme.

En Python, une structure de contrôle respecte trois règles de syntaxe essentielles :
- la ligne d'introduction se termine par deux-points : ;
- le bloc d'instructions associé est **indenté de façon constante**.

2.1 Structure conditionnelle – `if` / `elif` / `else`

Une structure conditionnelle exécute un bloc d'instructions seulement si une condition est vraie. **La condition est un booléen ou une expression booléenne**, souvent obtenue à l'aide d'une comparaison.

```
if condition_1:
    instructions_1
elif condition_2:
    instructions_2
else:
    instructions_3
```

Les conditions sont testées dans l'ordre. Dès qu'une condition est vraie, le bloc correspondant est exécuté et les branches suivantes sont ignorées. Les branches `elif` et `else` sont facultatives.

Exemple

Valeur absolue

```
x = -3
if x >= 0:
    valeur_absolue = x
else:
    valeur_absolue = -x
```

À la fin du programme, `valeur_absolue` vaut 3.

Question 5 : Écrire une structure conditionnelle qui affecte à la variable `mention` la chaîne `"admis"` si `note` est supérieure ou égale à 10, et `"refuse"` sinon.

2.2 Les boucles

2.2.1 Structure itérative conditionnelle – `while`

La boucle `while` permet de répéter un bloc d'instructions tant qu'une condition est vraie.

```
while condition:
    instructions
```

La condition est évaluée **avant chaque répétition**. Si elle est fausse dès le départ, le bloc n'est jamais exécuté. Une variable intervenant dans la condition doit généralement être modifiée dans la boucle afin d'éviter une boucle infinie.

Exemple

Compteur

```
compteur = 0
while compteur < 3:
    print(compteur)
    compteur = compteur + 1
```

Le programme affiche successivement 0, 1 et 2.

2.2.2 Structure itérative – `for`

La boucle `for` parcourt successivement les valeurs d'une séquence ou, plus généralement, d'un objet itérable.

```
for variable in sequence:
    instructions
```

Ainsi :

- le bloc d'instructions est répété autant de fois qu'il y a de valeurs dans la séquence ;
- la variable prend successivement chacune des valeurs de la séquence ;
- cette variable peut, sans obligation, être utilisée dans le bloc d'instructions.

Exemple

Parcours d'une chaîne

```
for caractere in "PCSI":
    print(caractere)
```

Les quatre caractères sont affichés successivement.

2.2.3 La fonction `range`

La fonction range produit une suite d'entiers, fréquemment utilisée avec une boucle `for`. Comme pour les extractions, la borne de fin est exclue.

Syntaxe	Entiers produits
`range(fin)`	de 0 à `fin - 1`
`range(debut, fin)`	de debut à `fin - 1`
`range(debut, fin, pas)`	de debut vers `fin`, avec le pas indiqué

Tableau 5 – Les trois syntaxes usuelles de range

On peut aussi, par exemple, se servir de la suite pour initialiser une liste :

Exemple

```
list(range(5))        # [0, 1, 2, 3, 4]
list(range(2, 6))     # [2, 3, 4, 5]
list(range(10, 3, -2)) # [10, 8, 6, 4]
```

Question 6 : Déterminer la valeur de somme à la fin du programme suivant.

```
somme = 0
for i in range(1, 5):
    somme = somme + i
```

Remarque : Quand on écrit `for i in range(1,5):`, en fait, on déclare une nouvelle variable de type `int`, de nom `i` qui prendra les valeurs de l'objet itérable (la suite d'entier renvoyée par range).

2.2.4 Création d'une liste par compréhension

Une liste peut être construite à partir d'un objet itérable en écrivant directement l'expression qui produit chacun de ses éléments.

```
carres = [i**2 for i in range(6)]
pairs = [i for i in range(10) if i % 2 == 0]
```

La première liste vaut `[0, 1, 4, 9, 16, 25]`. La seconde ne conserve que les valeurs qui vérifient la condition et vaut `[0, 2, 4, 6, 8]`.

Remarque : Une création par compréhension est adaptée aux constructions simples. Dès que le traitement devient difficile à lire, il est préférable d'utiliser une boucle classique.

2.3 Mots-clés `break` et `continue`

Mot-clé	Effet
`break`	interrompt immédiatement la boucle et poursuit le programme après celle-ci
`continue`	interrompt uniquement l'itération en cours et passe à la suivante

Exemple

Recherche d'une valeur

```
valeurs = [4, 7, -2, 9, -8]
for valeur in valeurs:
    if valeur < 0:
        print("Valeur négative trouvée")
        break
```

La boucle s'arrête dès que la première valeur négative est rencontrée.

Remarque : L'utilisation de ces deux mots-clés n'est pas courante.

3 Les fonctions

Une fonction permet de regrouper des instructions afin de les réutiliser. Elle facilite la factorisation du code, sa lecture, ses tests et son évolution.

Remarques :

- Une grande partie des questions de sujet consistent à écrire des fonctions. En effet, il est courant de découper un programme en fonctions très simples puis de construire des fonctions de plus en plus complexes à partir des précédentes.
- On retrouve l'idée des sciences de l'ingénieur d'adresser des problèmes simples en décomposant un problème complexe.

Définition

Fonction

Une fonction est un bloc d'instructions identifié par un nom. Elle peut recevoir des valeurs, appelées arguments lors de l'appel, et peut renvoyer un résultat à l'aide du mot-clé `return`.

Il faut distinguer la **définition** d'une fonction, qui crée un nouvel outil, et son **appel**, qui utilise cet outil. Les parenthèses indiquent l'appel d'une fonction, même lorsqu'aucun argument n'est nécessaire.

3.1 Définition et appel

```
def nom_fonction(parametre_1, parametre_2):
    """Description de la fonction."""
    instructions
    return resultat

valeur = nom_fonction(argument_1, argument_2)
```

Terme	Signification
Paramètre	nom utilisé dans la définition de la fonction
Argument	valeur fournie à la fonction lors de son appel
Valeur de retour	résultat transmis par la fonction avec `return`
Docstring	texte placé au début de la fonction pour décrire son rôle
Prototype ou signature	décrit comment appeler la fonction et éventuellement ce qu'elle retourne

Exemple

Fonction avec valeur de retour

```
def aire_rectangle(longueur, largeur):
    """Renvoie l'aire d'un rectangle."""
    aire = longueur * largeur
    return aire

surfaceChambre1 = aire_rectangle(5, 2)
surfaceChambre2 = aire_rectangle(4, 3)
```

Lors du premier appel, `longueur` reçoit l'argument 5 et `largeur` reçoit l'argument 2. La fonction renvoie 10, qui est affecté à `surfaceChambre1`.

Question 7 : Dans l'exemple précédent, préciser un paramètre, un argument, la valeur de retour, la docstring ainsi que la signature.

Remarques :

- Les triples guillemets `"""` forment un bloc de commentaire – donc à mettre au début et à la fin. Pour une seule ligne, le code est # en début de ligne.
- **L'instruction `return` interrompt immédiatement l'exécution de la fonction**. Une fonction sans instruction `return` renvoie la valeur particulière `None`, *c'est à dire rien*. Afficher une valeur avec `print` ne signifie pas la renvoyer.

Question 8 : Écrire une fonction `maximum(a, b)` qui renvoie le plus grand des deux nombres a et b.

Question 9 : Écrire la ligne permettant d'affecter le maximum entre 3 et 9 à la variable `max1`. Faire de même avec la variable `max2` pour les arguments `4.5` et `8e1`.

3.2 Paramètres par défaut

Un paramètre peut recevoir une valeur par défaut. L'argument correspondant devient alors facultatif lors de l'appel.

```
def puissance(nombre, exposant=2):
    return nombre**exposant

carre = puissance(3)       # renvoie 9
cube = puissance(3, 3)     # renvoie 27
```

3.3 Variables locales et globales

L'appel d'une fonction crée un espace mémoire dédié. Les variables définies dans cet espace sont des **variables locales** : elles n'existent que pendant l'**exécution** de la fonction. Une variable définie en dehors des fonctions est dite **globale**.

```
x = 10                  # variable globale

def calcul(a):
    resultat = a + 1    # variable locale
    return resultat

y = calcul(x)
```

La variable `resultat` n'est pas accessible après l'appel car il s'agit d'une variable locale. Autrement dit, à la fin de l'appel, l'espace mémoire de la fonction est libérée et la variable `resultat` disparait avec. En revanche, la valeur qu'elle contenait a été renvoyée puis affectée à `y`.

> **Remarque :** Une fonction doit autant que possible communiquer avec le reste du programme par ses paramètres et sa valeur de retour. La modification directe de variables globales rend généralement le programme plus difficile à comprendre et à tester.

3.4 Les méthodes

Les méthodes sont des fonctions spécifiques à certains types d'objets. On retrouve les mêmes propriétés que pour les fonctions : arguments, parenthèses et éventuelle valeur de retour. **Une méthode s'utilise avec un point après l'objet**. On a déjà vu plusieurs méthodes :

```
ma_liste = [1, 2]
ma_liste.append(5)
position = ma_liste.index(2)
```

Dans cet exemple, `append` modifie directement la liste, alors que `index` renvoie une valeur qui est affectée à `position`.

> **Remarque :** Il faut consulter la documentation pour savoir si une méthode modifie l'objet en question, renvoie un nouvel objet, renvoie une valeur, etc.

3.5 Les modules et bibliothèques

Pour ne pas avoir besoin de réinventer la roue à chaque fois, il existe des modules qui peuvent être importés. Dans ces modules, on retrouve des fonctions déjà implémentées ainsi que des constantes, comme π par exemple.

La connaissance exhaustive des modules n'est pas attendue, mais il faut savoir importer un module et utiliser sa documentation. Si, pendant un devoir, il est demandé d'utiliser une fonction particulière, la documentation nécessaire devrait être fournie.

Façon d'importer	**Effet**	**Utilisation**
`from math import pi`	Importe uniquement `pi` de la bibliothèque `math`	
`from math import *`	Importe tout ce qui est contenu dans `math`	
`import math`	Importe tout ce qui est contenu dans `math`	
`import math as mth`	Importe tout ce qui est contenu dans `math`	

Tableau 6 – Différentes syntaxes d'importation

> **Remarque :** L'écriture `from module import *` est déconseillée : elle ne permet pas d'identifier facilement l'origine des noms importés et peut créer des conflits de noms.

3.5.1 Le module `math`

Le module `math` fournit des constantes et des fonctions mathématiques pour des valeurs numériques scalaires.

Élément	Rôle
`math.pi`	valeur approchée de π
`math.sqrt(x)`	racine carrée de `x`
`math.sin(x)`, `math.cos(x)`	fonctions trigonométriques, avec `x` en **radians**
`math.exp(x)`, `math.log(x)`	exponentielle et logarithme népérien

3.5.2 Le module `numpy`

Le module `numpy` est principalement utilisé pour créer et manipuler efficacement des tableaux numériques, on y retrouve aussi des fonctions mathématiques, souvent capable de fonctionner avec des listes. Il est usuellement importé sous l'alias `np`.

```
1 import numpy as np
2
3 x = np.linspace(0, 10, 101)
4 y = np.sin(x)
```

La fonction `linspace(0, 10, 101)` crée un tableau(liste, nous les verrons plus tard) de 101 valeurs régulièrement espacées entre 0 et 10, bornes incluses. L'expression `np.sin(x)` calcule le sinus de chaque valeur du tableau `x`.

3.5.3 Le module `matplotlib.pyplot`

Le module `matplotlib.pyplot` permet de tracer des graphiques. Il est usuellement importé sous l'alias `plt`.

```
import matplotlib.pyplot as plt

plt.plot(x, y, label="sin(x)")
plt.xlabel("x (rad)")
plt.ylabel("sin(x)")
plt.grid()
plt.legend()
plt.show()
```

Question 10 : Compléter les principales étapes permettant de tracer la fonction cosinus sur l'intervalle $[0;2\pi]$.

Étape	**Instruction**
Importer module	
Créer 201 abscisses sur l'intervalle $[0;2\pi]$	
Calculer les ordonnées	
Tracer la courbe	
Afficher la grille	
Afficher la légende	
Afficher la figure	