Sujet n°1

Ce sujet est composé de deux problèmes indépendants. Une annexe (en dernière page) rappelle les principaux éléments de syntaxe Python nécessaires. L'écriture « `machine à écrire` » représente du code Python.

Problème 1. Le jeu de la vie

Un automate cellulaire est constitué d'une grille régulière de *cellules* étant chacune dans un *état* (parmi un nombre fini d'états possibles), et d'une règle d'évolution *locale* et *synchrone* : à chaque étape de temps (discret), l'état de chaque cellule est mis à jour en fonction de l'état de ses voisines (et du sien).
Le mathématicien John Conway a défini en 1970 un automate cellulaire particulier qu'il a nommé « jeu de la vie ». Il s'agit d'un automate dont la grille est une grille bi-dimensionnelle infinie, et chaque cellule peut prendre deux états 0 ou 1 (qu'on appelle habituellement « état mort » et « état vivant »). Les voisines d'une cellule sont les huit cellules qui l'entourent. Et la règle d'évolution est la suivante :

— une cellule vivante au temps t meurt au temps $t+1$ si elle n'est entourée que de 0 ou 1 cellule vivante (*isolement*), ou si elle est entourée d'au moins 4 cellules vivantes (*surpopulation*) ; elle *survit* dans les autres cas ;

— une cellule morte au temps t devient vivante au temps $t+1$ si elle est entourée d'exactement 3 cellules vivantes (*naissance*) ; elle reste morte sinon.

Certaines de ces règles sont illustrées par la figure 1.

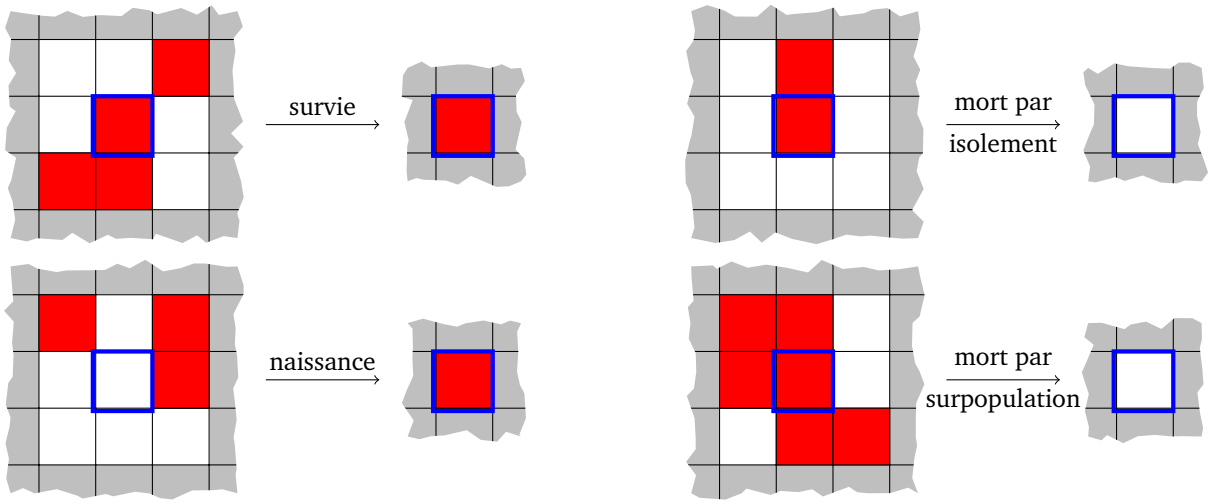


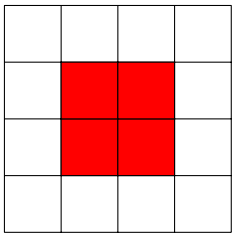
FIGURE 1 – Règles de survie, de mort et de naissance pour la cellule encadrée. Les cellules colorées sont les cellules vivantes. Le grisé qui entoure signifie qu'on ne connaît pas l'état des cellules voisines.

1.1 Simulation du jeu de la vie

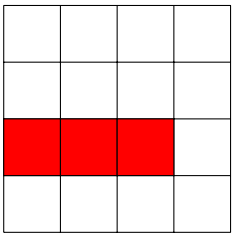
Pour pouvoir représenter de manière finie le jeu de la vie, on modifie la grille pour en faire un *tore de taille k* : au lieu d'être une grille infinie, c'est une grille $k \times k$ dont la colonne de gauche est voisine de la colonne de droite, et la ligne du haut voisine de la ligne du bas. De cette manière, la grille n'a pas de *bord* et toute cellule possède exactement 8 voisines. On peut représenter une grille par une *matrice* (tableau bi-dimensionnel) de taille $k \times k$, où 0 représente l'état mort et 1 l'état vivant. La représentation de la grille en Python sera une liste de listes. La numérotation des lignes et des colonnes de la matrice ira de 0 à $(k-1)$.

Question 1.1. Représentation de la grille

(a) On considère une grille de taille 4. Donner la représentation matricielle des deux configurations suivantes, et donner les configurations obtenues au temps $t+1$ et $t+2$ pour chacune. Que remarque-t-on ?



Configuration A



Configuration B

(b) Dans la grille de taille 4, quelles sont les coordonnées des cellules voisines de $(1,1)$? Et celles des voisines de $(0,3)$?

(c) Écrire une fonction `grille_vide` qui prend en entrée un entier n et renvoie une grille torique de dimension n vide (toutes les cellules sont mortes).

(d) Écrire une fonction `est_vide` qui prend en entrée une grille et renvoie un booléen, qui vaut `True` si toutes les cellules de la grille sont mortes.

(e) Écrire une fonction `afficher_grille` qui prend en entrée une grille et l'affiche ligne par ligne en représentant les 0 par le caractère `'-'` et les 1 par le caractère `'*'`. La fonction ne renverra rien. *Par exemple, les grilles correspondant aux configurations A et B de la question (a) seront affichées sous la forme ci-dessous.*

```
- - - -     - - - -
- * * -     - - - -
- * * -     * * * -
- - - -     - - - -
```

Question 1.2. Évolution de la grille

(a) Écrire deux fonctions `ind_suiv` et `ind_prec` qui prennent en entrée une grille et un indice i (de ligne ou de colonne) et renvoient l'indice de la ligne ou colonne suivante (resp. précédente) dans la grille. *Attention, il faut tenir compte du caractère torique de la grille !*

(b) Écrire une fonction `nombre_voisins` qui prend en entrée une grille et une cellule de la grille représentée par ses coordonnées (i,j), et renvoie le nombre de cellules *vivantes* parmi les voisines de (i,j), sans compter la cellule elle-même.

(c) Écrire une fonction `etat_suivant` qui prend en entrée une grille et une cellule, et renvoie l'état de la cellule (0 ou 1) au temps $t+1$.

(d) Écrire une fonction `configuration_suivante` qui prend en entrée une grille et renvoie la grille obtenue au temps $t+1$.

(e) Écrire une fonction `simulation` qui prend en entrée une grille et un entier N, et renvoie la grille obtenue après N étapes (au temps $t+N$).

Question 1.3. Sauvegarde d'une grille On peut sauvegarder une grille dans un fichier texte avec le format suivant. La première ligne contient un entier k représentant la dimension de la grille. Chacune des lignes suivantes contient deux entiers i et j séparés par un espace représentant les coordonnées d'une cellule vivante. *On donne ci-dessous des fonctions permettant de convertir une liste d'entiers en une chaîne de caractère contenant ces entiers séparés par des espaces, et inversement. Se référer à l'annexe pour les lectures et écritures dans un fichier.*

```python
def liste_vers_chaine(L):
    return ' '.join(str(n) for n in L)

def chaine_vers_liste(s):
    return [int(c) for c in s.split() if len(c)]
```

(a) Dessiner les configurations correspondant aux trois fichiers de la figure 2.

```
4
1 1
1 2
2 2
2 1
```

Automate1.txt

```
4
2 0
2 1
2 2
```

Automate2.txt

```
5
0 1
1 2
2 0
2 1
2 2
```

Automate3.txt

FIGURE 2 – Trois grilles sauvegardées sous forme de fichiers texte

(b) Écrire une fonction `ouvre_grille` qui reçoit en entrée une chaîne de caractère contenant un nom de fichier, et renvoie la grille (sous forme de matrice) représentée dans le fichier.

(c) Écrire une fonction `sauve_grille` qui reçoit en entrée une grille et une chaîne de caractère contenant un nom de fichier, et sauvegarde la grille dans le fichier dans le format décrit ci-dessus. *Si le fichier existe déjà, il est écrasé.*

1.2 Motifs et évolution

Question 1.4. Insertion et recherche de motifs Un motif est n'importe quel rectangle de cellules, certaines vivantes et certaines mortes. On appelle *cellule principale* d'un motif la cellule située en haut à gauche du motif. Des exemples de motifs sont représentés ci-dessous, avec la cellule principale encadrée.

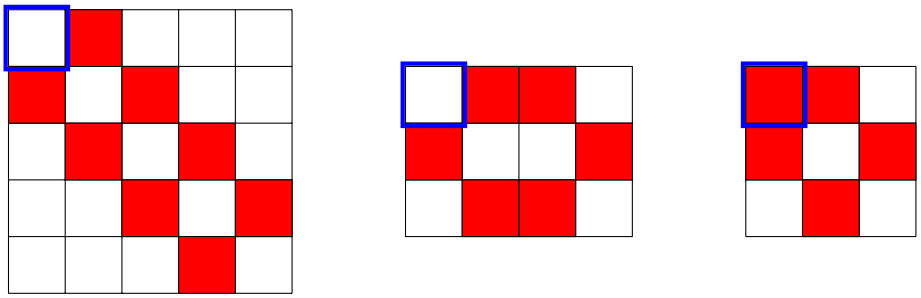


FIGURE 3 – Trois exemples de motifs

(a) Écrire une fonction `initialise` qui prend en entrée un motif et une dimension de grille m et renvoie la grille de dimension m contenant en son centre (approximatif) le motif. Si le motif est trop grand pour rentrer dans la grille, une exception sera soulevée. *On détaillera le choix effectué pour placer le motif approximativement au centre de la grille.*

(b) Écrire une fonction `extraire_motif` qui prend en entrée une grille, une cellule (représentée par ses coordonnées (i, j)) et des dimensions (m, n) et renvoie le motif de dimensions $m \times n$ dont la cellule principale a pour coordonnées (i, j). *L'extraction tient naturellement compte de la nature torique de la grille.*

(c) Écrire une fonction `appartient_grille` qui prend en entrée un motif et une grille et teste si la grille contient le motif. La fonction renvoie une liste de coordonnées : pour chaque occurence du motif dans la

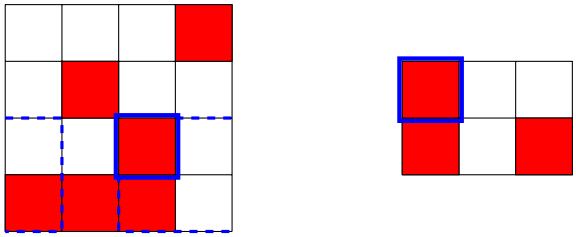


FIGURE 4 – Extraction d'un motif 2×3 depuis la cellule $(2,1)$ d'une grille.

grille, la liste contient les coordonnées de la cellule principale du motif dans la grille. Si le motif n'apparaît pas, la liste est naturellement vide. *Par exemple, avec en entrée le motif de taille* 1×1 , *la fonction renvoie les coordonnées de toutes les cellules vivantes.*

Question 1.5. Propriété des motifs et configurations Partant d'une grille G_0 au temps $t = 0$, on note G_i la grille obtenue au temps i. Une grille G_0 est un *point fixe* si $G_1 = G_0$. Une grille est dite *périodique de période p* (avec $p > 0$) si $G_p = G_0$.

(a) Vérifier que la grille du fichier `Automate1.txt` est un point fixe.

(b) Écrire une fonction `est_point_fixe` qui prend en entrée une grille et renvoie un booléen indiquant si la grille est un point fixe.

(c) Vérifier que la grille du fichier `Automate2.txt` est périodique. Préciser sa (plus petite) période.

(d) Écrire une fonction `est_periodique` qui prend en entrée une grille et un entier N et teste si la grille est périodique de période p pour un entier $p \leq N$.

(e) On veut modifier la fonction précédente pour ne pas avoir à donner de borne supérieure N. Pour une grille de dimension n, donner une borne supérieure sur la longueur de la plus petite période de la grille, s'il en existe une. Modifier en conséquence la fonction `est_periodique` pour qu'elle ne prenne en entrée qu'une grille.

Problème 2. Séquençage d'ADN

Ce problème s'intéresse au processus de séquençage de l'ADN. Dans ce contexte, l'une des étapes importantes du séquençage permet de déterminer quelles sont les parties de la séquence d'ADN recherchée mais dans un ordre indéterminé. Le problème consiste alors à **reconstruire** la séquence complète à partir de ces parties. La suite de ce sujet propose une résolution algorithmique à ce problème.

Pour cela, nous représentons **une séquence** S_n d'ADN de longueur n par une suite de n lettres, chaque lettre appartenant à l'ensemble $\{A, T, G, C\}$.
Exemple : $S_9 = AGGTCAGGT$ est une séquence d'ADN de 9 lettres.

2.1 Recherche de mots dans une séquence d'ADN

Dans un premier temps nous allons chercher à déterminer quelles sont les parties de longueur fixe d'une séquence, appelées **mots de la séquence** par la suite.
Formellement, soient S_n une séquence d'ADN et ℓ un entier tel que $0 < \ell \leq n$. On appelle *mot de S_n* de longueur ℓ toute suite de ℓ lettres contiguës contenue dans la séquence S_n.
Exemples : GTC est un mot de longueur 3 de la séquence $S_9 = AGGTCAGGT$, tandis que ATT n'en est pas un.

Question 2.1.

(a) Combien de mots de longueur ℓ existe-t-il dans une séquence de longueur n (en comptant les répétitions possibles) ?

(b) Proposer un algorithme qui, étant donnés un entier n, une séquence d'ADN S_n de longueur n et un entier strictement positif ℓ ($\ell \leq n$), calcule la liste de tous les mots de longueur ℓ de S_n.

Une telle liste peut contenir plusieurs fois le même mot. Vous supposerez que la séquence d'ADN est contenue dans un tableau de caractères. Il n'est pas demandé de décrire les manipulations de listes que vous aurez éventuellement besoin d'effectuer, comme l'ajout d'un élément dans une liste par exemple.

Exemple : l'algorithme appelé sur la séquence $S_9 = AGGTCAGGT$ et $\ell = 3$ doit calculer la liste $[AGG, GGT, GTC, TCA, CAG, AGG, GGT]$.

(c) Quelle est la complexité de votre algorithme en nombre d'opérations, en fonction de n et de ℓ ?

Question 2.2. On s'intéresse à la question de déterminer tous les mots distincts de longueur ℓ dans S_n. Nous cherchons ici à proposer une fonction Python. Une séquence S_n sera alors décrite comme une chaîne de caractères de longueur n.
Exemple : la liste de tous les mots distincts de longueur 3 dans la séquence $S_9 =$ "$AGGTCAGGT$" où $n = 9$ et $\ell = 3$ est ["AGG","GGT","GTC","TCA","CAG"].

(a) Soient n et ℓ deux entiers ($n \geq \ell > 0$). Combien de mots distincts de longueur ℓ existe-t-il au plus dans une séquence de longueur n ?

(b) Proposer une structure de données en Python adaptée pour stocker les mots de la séquence sans répétition.

(c) Proposer une fonction `liste_mots` en Python qui prend en arguments une séquence S_n, un entier n et un entier ℓ ($n \geq \ell > 0$), et qui renvoie la liste de tous les mots distincts de longueur ℓ de S_n.

2.2 Reconstruction d'une séquence d'ADN

L'une des difficultés lors du séquençage de l'ADN est qu'il faut reconstruire la séquence complète à partir de l'ensemble des mots de longueur ℓ qui la composent, sans qu'aucune information sur leur position dans la séquence ne soit donnée.
Le reste de cet exercice va se concentrer précisément sur la résolution algorithmique de ce problème.

Soient S_n une séquence d'ADN de longueur n et ℓ un entier tel que $0 < \ell \leq n$. On appelle spectre(S_n, ℓ) l'ensemble des mots de longueur ℓ distincts qui sont dans la séquence S_n.
Exemple : étant donnée la séquence $S_9 = AGGTCAGGT$, spectre$(S_9, 3) = \{AGG, CAG, GGT, GTC, TCA\}$.
Le problème de la **reconstruction** de la séquence d'ADN consiste, à partir de la connaissance d'un spectre SP contenant des mots de longueur ℓ, à déterminer une séquence d'ADN S compatible avec SP, c'est-à-dire

telle que spectre$(S,\ell) = \text{SP}$. Plus formellement, on peut formuler le problème RECONSTRUCTION de la manière suivante :

Données : un entier ℓ ; un ensemble SP de mots de longueur ℓ.
Sortie : une séquence d'ADN S telle que spectre$(S,\ell) = \text{SP}$.

Pour résoudre ce problème, nous allons nous intéresser aux recouvrements existant entre les mots d'un spectre. Soient M un mot de longueur $n > 0$ et k un entier tel que $0 < k \leq n$, on appelle *préfixe* de M de longueur k le mot constitué des k premières lettres de M. De manière similaire, on appelle *suffixe* de M de longueur k le mot constitué des k dernières lettres de M.

Soient deux mots M_1 et M_2 de longueur n_1 et n_2 respectivement et soit $k \leq \min(n_1, n_2)$ un entier strictement positif. Il existe un *recouvrement de longueur* k entre M_1 et M_2 si et seulement si le suffixe de M_1 de longueur k est identique au préfixe de M_2 de longueur k. Nous appellerons également *longueur du recouvrement maximal* entre M_1 et M_2, noté lrmax(M_1, M_2) dans la suite, le plus grand entier k tel qu'il existe un recouvrement de longueur k entre M_1 et M_2. Si aucun recouvrement n'existe entre M_1 et M_2, lrmax$(M_1, M_2) = 0$.
Exemple : lrmax$(ACGG, CTAG) = 0$ *et* lrmax$(ACGG, GGAT) = 2$.

Question 2.3. Que valent :

- lrmax$(ATGC, GGTA)$?
- lrmax$(TGGCGT, CGTAAATG)$?
- lrmax$(GCTAGGCTAA, AGGCTAAGTCGAT)$?
- lrmax$(TCTAGCCAGCTAGC, TAGCCAGCTAGCACT)$?

Question 2.4. Première modélisation Soient $\ell \geq 2$ un entier et SP un spectre contenant des mots de longueur ℓ. Nous allons modéliser notre problème par un graphe orienté $G = (V, E)$, dans lequel :

- Chaque sommet de V correspond à un mot du spectre et chaque mot du spectre est représenté par un et un seul sommet.
- L'ensemble E contient un arc entre $v_1 \in V$ et $v_2 \in V$ si et seulement si lrmax$(v_1, v_2) = \ell - 1$.

(a) Quelle est la modélisation obtenue sous forme de graphe pour les spectres et longueurs suivants :

- $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$ et $\ell = 4$?
- $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $\ell = 3$?

(b) Proposer une formulation du problème de reconstruction comme un parcours de graphe. De quel problème classique de théorie des graphes ce problème se rapproche-t-il ?

(c) Proposer pour chacun des graphes obtenus dans la question **(a)** une solution au problème RECONSTRUCTION.

Question 2.5. Deuxième modélisation Il est possible de modéliser autrement ce problème, toujours sous forme de graphe. Nous modélisons maintenant les données ($\ell \geq 2$ un entier et SP un spectre contenant des mots de longueur ℓ) par un graphe orienté $G = (V, E)$ où :

- V est l'ensemble de tous les préfixes de longueur $\ell - 1$ et de tous les suffixes de longueur $\ell - 1$ des mots du spectre.
- L'ensemble E contient un arc entre les sommets $v_1 \in V$ et $v_2 \in V$ si et seulement si le spectre SP contient un mot M de longueur ℓ tel que v_1 est le préfixe de M de longueur $\ell - 1$ et v_2 est le suffixe de M de longueur $\ell - 1$.

(a) Quelle est la modélisation obtenue sous forme de graphe pour les spectres suivants :

- $\{GTGA, ATGA, GACG, CGTG, ACGT, TGAC\}$ et $\ell = 4$?
- $\{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $\ell = 3$?

On rappelle que dans un graphe orienté $G = (V, E)$, un *chemin* d'origine u ($\in V$) et d'extrémité v ($\in V$) est défini par une suite d'arcs consécutifs reliant u à v. Un *circuit* est un chemin dont les deux extrémités sont identiques. Un chemin (resp. circuit) est dit *eulérien* s'il contient une fois et une seule chaque arc de G.

(b) Les graphes construits à l'aide de la modélisation de la question **(a)** contiennent-ils un circuit eulérien ? un chemin eulérien ?

(c) En quoi un circuit ou un chemin eulérien dans un graphe construit avec cette modélisation constitue-t-il une solution au problème de la reconstruction de séquence d'ADN ?

Dans la suite de l'exercice, nous nous restreignons au circuit eulérien qui est plus simple à déterminer qu'un chemin eulérien.
Soit $(u, v) \in E$. On dit que l'arc (u, v) est un arc entrant du sommet v et un arc sortant du sommet u. On définit le degré entrant d'un sommet v, noté de $d_e(v)$, comme le nombre d'arcs entrants du sommet v. On définit le degré sortant d'un sommet u, noté $d_s(u)$, comme le nombre d'arcs sortants du sommet u.

Question 2.6. Montrer qu'un graphe orienté contient un circuit eulérien si et seulement si $\forall v \in V, d_e(v) = d_s(v)$.

Dans la suite de l'exercice, un graphe sera représenté en Python par un dictionnaire dont les clefs sont des chaînes de caractères (`str`) représentant les sommets du graphe et les valeurs sont des listes de chaînes de caractères (`list`) contenant les sommets voisins de la clé.

Question 2.7.

(a) Écrire une fonction `presence_circuit_eulerien` en Python qui prend en arguments un graphe orienté G et qui détermine si G a un circuit eulérien.

Nous supposons que les graphes considérés dans les questions **(b)** à **(e)** comprennent un circuit eulérien.

(b) Écrire une fonction `construction_circuit` en Python qui prend en arguments un graphe orienté G et un sommet v quelconque de G et qui renvoie un circuit ayant pour origine le sommet v.

(c) Écrire une fonction `enleve_circuit` en Python qui prend en arguments un graphe orienté G et un circuit C de G et qui supprime de G les arcs appartenant au circuit C.

(d) Écrire une fonction `sommet_commun` en Python qui prend en arguments un graphe orienté G et un circuit C (qui n'est pas inclus dans G) et qui renvoie un sommet commun de degré non nul appartenant à la fois à G et à C. *Nous supposerons que G et C ont au moins un tel sommet en commun.*

(e) Écrire une fonction `fusion_circuits` en Python qui prend en arguments deux circuits $C1$ et $C2$ et v un sommet commun à $C1$ et $C2$ et qui renvoie un circuit composé des arcs de $C1$ et des arcs de $C2$.

Question 2.8.

(a) À partir des fonctions Python que vous avez proposées à la question 2.7, écrire une fonction `circuit_eulerien` en Python qui prend un entrée un graphe orienté G et qui renvoie un circuit eulérien de G.

On suppose toujours que G contient un circuit eulérien.

(b) Quelle est la séquence d'ADN déterminée par la fonction proposée à la question **(a)** sur le graphe obtenu à la question 2.5**(a)** avec le spectre $\text{SP} = \{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et $\ell = 3$?

(c) Proposer une autre séquence d'ADN compatible avec le spectre $\text{SP} = \{TAC, ACC, ACG, CAC, CCG, CGT, CGC, GCA, GTA\}$ et différente de la séquence déterminée à la question **(b)**.

Aide-mémoire Python

Listes

```
>>> maListe = [1, 2, 3]    # Création
>>> maListe[2]              # Elément d'indice 2
3
>>> maListe[0] = 0          # Modification
>>> len(maListe)            # Longueur
3
>>> maListe + [4, 5]        # Concaténation
[0, 2, 3, 4, 5]
>>> maListe.append(4)       # Ajout en fin de liste
>>> maListe.insert(2, 17) # Insertion dans la liste
>>> maListe.pop(1)          # Suppression de l'élément d'indice 1
2
>>> maListe.remove(3)       # Suppression du premier 3 trouvé
>>> maListe
[0, 17, 4]
>>> for x in maListe: ... # Parcours
```

Uplets

```
>>> monUplet = (1,2,3)       # Création
>>> monUplet[1]              # Deuxième élément
'2'
>>> monUplet[0] = 0          # Affectation interdite !
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
>>> len(monUplet)            # Longueur
3
>>> monUplet + (4,5)         # Concaténation
(1, 2, 3, 4, 5)
>>> for c in monUplet : ... # Parcours
```

Chaînes de caractère

```
>>> maChaine = 'abcde'       # Création
>>> maChaine[3]              # Quatrième élément
'd'
>>> maChaine[4] = x          # Affectation interdite !
Traceback (most recent call last):
...
TypeError: 'str' object does not support item assignment
>>> len(maChaine)            # Longueur
5
>>> maChaine + 'fg'          # Concaténation
'abcdefg'
>>> for c in maChaine : ... # Parcours
```

Dictionnaires

```
>>> monDico = {1:'a', 2:'b', 3:'c'}      # Création
>>> monDico[3]                           # Valeur associée à la clef 3
'c'
>>> monDico[4] = 'd'                     # Ajout d'une valeur
>>> monDico[1] = 'z'                     # Modification d'une valeur
>>> for k in monDico:                    # Parcours des clefs
...     monDico[k] += ' !'
>>> monDico
{1: 'z !', 2: 'but !', 3: 'c !', 4: 'd !'}
```

Lecture et écriture de fichiers

```
>>> with open("mon_fichier.txt", "w") as f: # Écriture dans un fichier
...     a = f.write("Bon")                  # a = nb de caractères écrits
...     b = f.write("jour\n")               #   -> 0 si erreur d'écriture
...
>>> a, b                                   # écriture réussie
(3, 5)
>>>
>>> with open("mon_fichier.txt", "r") as f: # Lecture ligne à ligne
...     for ligne in f:                     # chaque ligne contient \n
...         print(len(ligne), ligne[:-1])   #    comme dernier caractère
...
8 Bonjour
12 Au revoir !
```