

1 Introduction

Les types de données simples ont été vus précédemment : ce sont les types `int` (nombres entiers), `float` (nombres flottants), `bool` (booléens).

Le type `str` est un peu moins simple. Un objet de type `str` est une chaîne de caractères qui s'écrit entre des guillemets ou des apostrophes. Dans une chaîne, chaque caractère est repéré par un indice qui commence à 0. Avec la variable de type chaîne `mot = "Exemple"`, `mot[0]` est le caractère "E", `mot[1]` est le caractère "x", et ainsi de suite.

Ces types simples ne sont plus suffisants si nous avons besoin de garder en mémoire un grand nombre de valeurs comme dans le cas d'un traitement de données statistiques. Il en est de même si l'on souhaite regrouper des valeurs, par exemple afin d'avoir une variable représentant les coordonnées d'un point.

L'objectif est donc de construire un type de variable capable de contenir plusieurs valeurs. Nous pouvons nous inspirer du type `str` et utiliser des indices pour repérer les éléments. Ceci amène à la construction des p-uplets, type `tuple`, et des listes, type `list`.

2 Les p-uplets (tuples)

2.1 DEFINITION

Un objet de type `tuple`, un p-uplet, est une suite ordonnée d'éléments qui peuvent être chacun de n'importe quel type. On parle indifféremment de p-uplet ou de tuple.

2.2 CREATION D'UN P-UPLET

Pour créer un p-uplet non vide, on écrit n valeurs séparées par des virgules.

`t = "a", "b", "c", 3, 4` pour un tuple à 5 éléments

Les parenthèses ne sont pas obligatoires, mais elles améliorent la lisibilité :

`t = ("a", "b", "c", 3, 4)`

Un objet t de type tuple n'est pas modifiable par une affectation `t[i]=valeur`.

Commande à saisir	Résultat obtenu	Opération réalisée
<code>>>> t = "a", "b", "c", 3, 4</code> <code>>>> t</code> <code>>>> type(t)</code>	('a', 'b', 'c', 3, 4) <class 'tuple'>	Création d'un p-uplet (cinq éléments)
<code>>>> t[0]</code>	'a'	donne la 1 ^e valeur
<code>>>> t[1]</code>	'b'	donne la 2 ^e valeur
<code>>>> t[-1]</code>	4	donne la dernière valeur
<code>>>> t2 = t[1:3]</code> <code>>>> t2</code> <code>>>> type(t2)</code>	('b', 'c') <class 'tuple'>	Création d'un p-uplet à partir d'un autre
<code>>>> t2[0] = "a"</code>	Type error: tuple etc.	un p-uplet n'est pas modifiable
<code>>>> t3 = "abc",</code> <code>>>> t3</code> <code>>>> type(t3)</code>	('abc',) <class 'tuple'>	création d'un p-uplet (un élément)
<code>>>> t4 = ()</code> <code>>>> t4</code> <code>>>> type(t4)</code>	() <class 'tuple'>	création d'un p-uplet à Ø élément

2.3 OPERATIONS SUR LES P-UPLETS

Nous avons deux opérateurs de concaténation qui s'utilisent comme avec les chaînes de caractères, ce sont les opérateurs + et *. De nouveaux p-uplets peuvent être créés à partir de ces opérations.

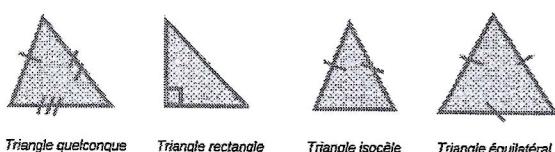
Commande à saisir	Résultat obtenu	Opération réalisée
>>> t5 = "a", "b" >>> t6 = "c", "d" >>> t7 = t5 + t6 >>> t7 >>> type(t7)	('a', 'b', 'c', 'd') <class 'tuple'>	+
>>> 3*t5	('a', 'b', 'a', 'b', 'a', 'b')	*
>>> "a" in t5	True	in
>>> "c" in t5	False	in
>>> len(t7)	4	len()
>>> t8=('a','b','c'),('c','d') >>> t8 >>> type(t8) >>> len(t8)	((('a', 'b', 'c'), ('c', 'd'))) <class 'tuple'> 2	p-uplets emboîtés : un p-uplet contenant des p-uplets
>>> type(t8[0]) >>> len(t8[0])	<class 'tuple'> 3	Le premier élément du p-uplet t8 est un p-uplet de 3 éléments
>>> t8[1][0] >>> type(t8[1][0])	'c' <class 'str'>	Le premier élément du second p-uplet est une chaîne
>>> t8[0][2] == t8[1][0]	True	On a bien la même valeur pour ces deux éléments

On peut utiliser un p-uplet pour effectuer une affectation multiple de variables.

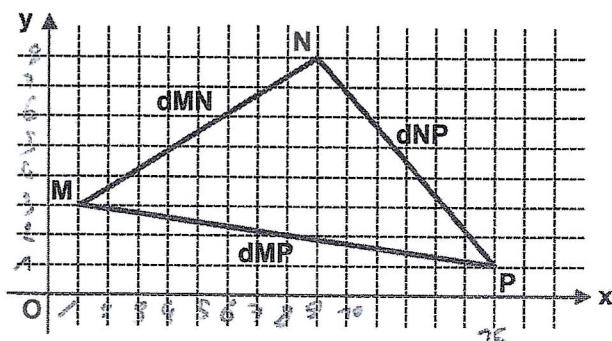
Commande à saisir	Résultat obtenu	Opération réalisée
>>> t = 2, 3.14, 'oui', False >>> t >>> type(t) >>> a, b, c, d = t >>> a >>> type(a) >>> b >>> type(b) >>> c >>> type(c) >>> d >>> type(d)	(2, 3.14, 'oui', False) <class 'tuple'> 2 <class 'int'> 3.14 <class 'float'> 'oui' <class 'str'> False <class 'bool'>	Affectation multiple de variables à l'aide d'un p-uplet

Exercice 1 :

Écrire un programme **LongueurCotesTriangle.py** qui calcule la longueur des trois côtés d'un triangle. Les sommets M, N et P sont donnés par leurs coordonnées (x,y) sous forme de p-uplets.



$$dMN = \sqrt{(x_N - x_M)^2 + (y_N - y_M)^2}$$



Le script du programme *LongueurCotesTriangle.py*, donné ci-dessous est à compléter et à tester.

```
import numpy as math # permet d'avoir accès à la fonction racine carrée du module numpy

def calculLongueurs(A, B, C):
    xA, yA = A
    xB, yB = B
    xC, yC = C

    dAB = math.sqrt((xA - xB) ** 2 + (yA - yB) ** 2)
    dBC = math.sqrt((xB - xC) ** 2 + (yB - yC) ** 2)
    dAC = math.sqrt((xA - xC) ** 2 + (yA - yC) ** 2)

    return dAB, dBC, dAC

def main():
    """ Calcul de la longueur des côtés d'un triangle """
    M = (3.4, 7.8) # coordonnées du premier point de type tuple
    N = (5, 1.6) # coordonnées du second point
    P = (-3.8, 4.3) # coordonnées du troisième point

    dMN, dNP, dMP = calculLongueurs(M, N, P) # appel de la fonction de calcul

    print("La longueur du côté MN est", round(dMN, 3))
    print("La longueur du côté NP est", round(dNP, 3))
    print("La longueur du côté MP est", round(dMP, 3))

    print("La longueur totale des côtés du triangle est", dMN + dNP + dMP)

if __name__ == '__main__':
    main()
```

Avec les valeurs des coordonnées saisies ci-dessus dans le script, vous devriez obtenir :

```
Console > 
>>> %Run LongueurCotesTriangle.py
La longueur du côté MN est 6.403
La longueur du côté NP est 9.205
La longueur du côté MP est 8.006
La longueur totale des côtés du triangle est 23.614
>>>
```

A partir du graphique donné sur la page précédente, déterminez les coordonnées des points M, N et P (les lignes du quadrillage sont espacées d'une unité), modifiez-les dans votre programme et consignez les résultats obtenus ci-dessous :

$$M = 7; 3$$

$$N = 9; 8$$

$$P = 15; 7$$

$$dMN = 9,934$$

$$dNP = 9,22$$

$$dMP = 19,192 \quad \text{Longueur Totale} = 32,796$$

A faire avant de commencer :

Installer le module numpy dans Thonny :

- Outils
- Gérer les paquets
- Rechercher « numpy »
- Installer
- Fermer

Relancer Thonny.exe

3 Les listes

3.1 DEFINITION

Contrairement au type tuple, un objet de type list est modifiable par une affectation, ce qui autorise de nombreuses méthodes applicables à ces objets mais en contrepartie nécessite une grande vigilance sur leur utilisation.

Une liste, ressemble à un p-uplet : un ensemble ordonné d'éléments avec des indices pour les repérer.
Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

3.2 CREATION D'UNE LISTE

Commande à saisir	Résultat obtenu	Opération réalisée
>>> liste1 = ["a", "b", "c"] >>> liste1 >>> type(liste1) >>> len(liste1)	[a, b, c] <class 'list'> 3	Création d'une liste de trois éléments
>>> liste2 = [1] >>> liste2 >>> type(liste2)	[1] <class 'list'>	création d'une liste d'un élément
>>> liste3 = [[1,2],[3,4,5]] >>> liste3 >>> type(liste3) >>> liste3[0] >>> type(liste3[0]) >>> liste3[1] >>> type(liste3[1])	[[1, 2], [3, 4, 5]] <class 'list'> [1, 2] <class 'list'> [3, 4, 5] <class 'list'>	création de deux lists dans une liste
>>> liste4 = [] >>> liste4 >>> len(liste4)	[] 0	création d'une liste vide
>>> liste5 = list(range(5)) >>> liste5 >>> len(liste5)	[0, 1, 2, 3, 4] 5	Utilisation de la fonction range pour construire une liste d'entiers
>>> multip3 = [] >>> multip3 >>> for i in range(11): ... multip3.append(3 * i) >>> multip3	[] [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]	Création d'une liste vide Afficher la liste vide Boucle à parcourir 11 fois (de 0 à 10) append = Ajouter chaque élément calculé à la liste (méthode) Afficher la liste complète (11 éléments)

3.3 CONSTRUCTION PAR COMPREHENSION

L'instruction s'écrit sous la forme [expression(i) for i in objet]. Ce type de construction est spécifique au langage Python.

Exemples :

```
>>> multiples_de_3 = [3 * i for i in range(14)]  
>>> multiples_de_3
```

```
>>> multiples_de_6 = [2 * n for n in multiples_de_3]  
>>> multiples_de_6
```

Si on dispose d'une fonction f et d'une liste d'abscisses :
images = [f(x) for x in abscisses]

Listes emboîtées :

```
>>> liste = [ [ [i, j] for i in range(3) ] for j in range(2) ]  
>>> liste
```

Exercice 2 :

Le script du programme **FonctionX2.py**, donné ci-dessous est à compléter et à tester.
On veut créer une liste d'abscisses de 0 à 5 par pas de 0.25
On veut créer la liste d'ordonnées qui correspond aux valeurs de $f(x)$.
On veut créer la liste des coordonnées des points de la fonction $f(x)$, chaque point est caractérisé par ses coordonnées $[x, f(x)]$.

La fonction souhaitée est $y = f(x) = x^2 + 3x + 2$

```
def calcul(x):
    """ Calcul des valeurs d'une fonction du second degré """
    resultat = (x ** 2) + 3 * x + 2 # calcul de f(x)
    return resultat

def main():
    """ Calcul des valeurs d'une fonction f(x) """

    abscisses = [x * 0.25 for x in range(0, 21)] # création de la liste des abscisses à calculer
    ordonnees = [calcul(x) for x in abscisses] # création de la liste des ordonnées
    points = [(abscisse, ordonnee) for abscisse, ordonnee in zip(abscisses, ordonnees)] # création de la liste des coordonnées des points
    print("Liste des abscisses :", abscisses)
    print("Liste des ordonnées :", ordonnees)
    print("Liste des points : ", points) # affichage de la liste des points

if __name__ == '__main__':
    main()
```

Travail à faire :

1. Dans la fonction `calcul(x)`, compléter la ligne `resultat = ...` en codant la fonction du second degré proposée.
2. Dans le programme principal, compléter la ligne `abscisses = ...`

Tester votre script et si votre programme est fonctionnel :

3. Limiter le format des valeurs des ordonnées à deux décimales (modifier la ligne concernée).
4. Créer une liste de points (liste de listes) qui contient les coordonnées $[x, f(x)]$ de la fonction.
5. Afficher cette liste de points sous les deux autres listes

3.4 METHODES UTILISABLES AVEC LES LISTES

Le type d'une variable définit les valeurs qui peuvent être affectées à cette variable ainsi que les opérateurs et les fonctions utilisables. Les fonctions propres à un type donné sont appelées des méthodes.
 La fonction `len` par exemple, s'applique aux chaînes de caractères, aux p-uplets et aux listes.
 La méthode `append` (voir tableau du 3.2) est par contre propre aux listes.

Attention à la syntaxe : on écrit `len(liste)` pour une fonction, mais `liste.append(...)` pour une méthode.
 Le nom de la variable est suivi d'un point puis du nom de la méthode.

Exemples de méthodes utilisables sur les listes :

Commande à saisir	Résultat obtenu	Opération réalisée
<pre>>>> liste6 = ["a", "c", "d"] >>> liste6 ['a', 'c', 'd'] >>> liste6.insert(1, "b") >>> liste6</pre>	<pre>['a', 'b', 'c', 'd']</pre>	Création d'une liste de trois éléments <code>.insert(i,var)</code> = Insérer à l'indice i (de la liste) l'élément var
<pre>>>> liste6.append("e") >>> liste6</pre>	<pre>['a', 'b', 'c', 'd', 'e']</pre>	<code>.append(var)</code> = insérer un élément à la fin
<pre>>>> liste6.remove("c") >>> liste6</pre>	<pre>['a', 'b', 'd', 'e']</pre>	<code>.remove(var)</code> = supprime l'élément dans l'ordre
<pre>>>> liste6.reverse() >>> liste6</pre>	<pre>['e', 'd', 'b', 'a']</pre>	<code>.reverse()</code> = inverse la liste
<pre>>>> liste6.append("c") >>> liste6 >>> liste6.sort() >>> liste6</pre>	<pre>['a', 'b', 'c', 'c', 'd', 'e']</pre>	<code>.sort()</code> = met dans l'ordre les éléments
<pre>>>> var = liste6.pop() >>> var >>> liste6 >>> var = liste6.pop() >>> var >>> liste6 >>> var = liste6.pop() >>> var >>> liste6</pre>	<pre>liste6: ['a', 'b', 'c', 'd'] var = e liste6: ['a', 'b', 'c'] var = d liste6: ['a', 'b'] var = c liste6: ['a'] var = b liste6: []</pre>	<code>.pop()</code> = prend la dernière valeur de la liste et l'assigne à une variable en la faisant disparaître de la liste

Ces méthodes modifient la liste initiale contrairement aux opérateurs de concaténation + et * avec lesquels une nouvelle liste est créée. Ces deux opérateurs de concaténation + et * s'utilisent comme avec les p-uplets (voir tableau du 2.3).

Il est possible de trier une liste sans la modifier avec la fonction `sorted` qui crée une nouvelle liste.

Commande à saisir	Résultat obtenu	Opération réalisée
<pre>>>> liste7 = [5, 2, 7, 4] >>> liste7 [5, 2, 7, 4] >>> liste8 = sorted(liste7) >>> liste8 [2, 4, 5, 7] >>> liste7</pre>	<pre>[5, 2, 7, 4]</pre>	Création d'une liste de quatre éléments <code>sorted(liste)</code> = tri une liste dans l'ordre et l'assigne dans une autre liste
<pre>>>> liste9 = ["rs", "wd", "pic", "adc"] >>> liste9 ['rs', 'wd', 'pic', 'adc'] >>> liste10 = sorted(liste9, reverse=True) >>> liste10</pre>	<pre>['adc', 'pic', 'rs', 'wd']</pre>	Création d'une liste de quatre éléments

Attention aux copies de listes !!!

Commande à saisir	Résultat obtenu	Opération réalisée
<pre>>>> liste11 = [1, 2, 3, 4] >>> liste12 = liste11 >>> liste11 >>> liste12 >>> liste11[1]=5 >>> liste11 >>> liste12</pre>	<p>[1, 2, 3, 4] [1, 2, 3, 4] [1, 2, 3, 4]</p>	<p>Création d'une liste de quatre éléments « Copie » de la liste</p> <p>Modifier l'élément 1 de la liste initiale</p>
<pre>>>> liste11 = [1, 2, 3, 4] >>> liste12 = list(liste11) >>> liste11 >>> liste12 >>> liste11[1]=5 >>> liste11 >>> liste12</pre>	<p>[1, 2, 3, 4] [1, 2, 3, 4] [1, 2, 3, 4]</p>	<p>Création d'une liste de quatre éléments Création d'une copie de la liste</p>

Une liste est juste une adresse en mémoire qui contient les adresses de ses éléments.

Dans le premier cas, la copie créée est une nouvelle adresse en mémoire qui contient les adresses des éléments. Chacun des éléments garde la même adresse, celle définie dans la liste initiale.

Dans le second cas, on crée une nouvelle liste et on copie aussi tous les éléments de la liste initiale dans d'autres emplacements mémoire. Il s'agit bien d'une liste différente de la liste initiale.

Des fonctions de copie sont disponibles à partir du module `copy`.

En particulier, dans le cas d'une liste de listes, pour effectuer une copie en profondeur, nous disposons de la fonction `deepcopy`.

```
>>> from copy import deepcopy
>>> liste1 = [["a", "b"], ["c", "d"]]
>>> liste2 = deepcopy(liste1)
>>> liste2[1][0] = "e"
>>> liste2
```

```
>>> liste1
```

3.5 IMPORTATION DE MODULES

Pour utiliser certaines fonctions, il faut importer un module : une sorte de "collection" de fonctions.

Par exemple, le module `math` pour les fonctions en mathématiques, le module `csv` pour manipuler des données en CSV, le module `PIL` pour manipuler des photos numériques, le module `folium` pour la géolocalisation, le module `micropython` pour l'informatique embarquée ... etc. On peut importer un module de plusieurs façons :

```
import nomDuModule          # on importe toutes les fonctions
                             # ensuite il faut appeler la fonction comme ceci : nomDuModule.nomFonction
```

```
import nomDuModule as alias # on importe toutes les fonctions
                             # ensuite il faut appeler la fonction comme ceci : alias.nomFonction
```

```
from nomDuModule import *   # on importe toutes les fonctions
                             # ensuite il faut appeler la fonction comme ceci : nomFonction
```

```
from nomDuModule import nomFonction # on importe uniquement la fonction
                                    # ensuite il faut appeler la fonction comme ceci : nomFonction
```

nous
fera
amelle

Dans la console Python, pour obtenir de l'aide sur un module et ses fonctions, il faut d'abord l'importer avec `import nomDuModule` puis taper `help(nomDuModule)`, ou consulter la documentation sur internet sur <http://docs.python.org/3/>.

Exemple :

```
>>> import webbrowser
>>> webbrowser.open('https://www.openstreetmap.org/#map=17/49.41437/2.10737')
```

Exercice 3 :

Le script du programme **CourbeX2.py**, donné ci-dessous est à compléter et à tester (*utiliser une copie du script de FonctionX2.py*).
On veut créer et afficher une liste d'abscisses de -3.0 à +3.0 par pas de 0.1
On veut créer et afficher la liste d'ordonnées qui correspond aux valeurs de $f(x)$.
On veut créer et afficher la liste des coordonnées $[x, f(x)]$ des points de la fonction $f(x)$.
On veut visualiser la courbe de la fonction $f(x)$ sur l'intervalle choisi.

La fonction souhaitée est $y = f(x) = x^2 + x - 4$

```
import matplotlib.pyplot as plt # bibliothèque pour représenter des graphiques en 2D

def calcul(x):
    """ Calcul des valeurs d'une fonction du second degré """
    resultat = ...  $x^2 + x - 4$ 
    return resultat

def main():
    """ Calcul des valeurs d'une fonction f(x) """
    abscisses = [...] [ -3 + i * 0,1 for i in range(61) ] # création de la liste des abscisses à calculer
    ordonnees = [...] [calcul(i) for i in abscisses] # création de la liste des ordonnées
    points = [...] [ [absisses[i], ordonnees[i]] for i in range(61) ] # création de la liste des coordonnées des points de la fonction
    print("Liste des abscisses :", abscisses) # affichage de la liste des abscisses dans la console
    print("Liste des ordonnées :", ordonnees) # affichage de la liste des ordonnées dans la console
    print("Liste des points :", points) # affichage de la liste des points dans la console
    plt.plot(abscisses, ordonnees) # trace des lignes entre les points
    plt.savefig("maCourbe.png") # sauvegarde la courbe dans un fichier image
    plt.show() # montre la courbe obtenue dans une fenêtre
    plt.close() # ferme la fenêtre ouverte avec show (fermée dans Windows)

if __name__ == '__main__':
    main()
```

*Remarque : La fonction show est bloquante.
Tant que la fenêtre n'a pas été fermée avec l'appui sur la croix, le reste du code ne s'exécute pas.*

4 Les tableaux et matrices

Une liste peut être composée d'objets de différents types et en particulier de listes.

Une liste constituée de n listes de longueur p représente un tableau de n lignes et de p colonnes, ou une matrice (n, p).

Attention à la taille qui pourrait excéder la capacité maximale de la mémoire (quelques dizaines de millions).

On peut tester cette capacité maximale avec l'instruction suivante :

```
>>> liste = [0] * n      où l'on met une valeur numérique à la place de n pour déterminer sa valeur maximale
```

Pour créer une liste de listes, on peut commencer par créer une liste vide **matrice** puis avec une boucle **for**, on crée les listes **ligne** une par une en les ajoutant à la liste **matrice**.

Exemple 1 à tester dans un fichier **matrice.py**

```
# création d'une matrice 5 x 4
matrice = []
for n in range(5):
    ligne = [4*n+i for i in range (4)]
    matrice.append(ligne)
print("Matrice de 5 lignes et de 4 colonnes")
print(matrice)
```

Après avoir testé et complété les deux lignes sur votre document, ajouter dans le fichier **matrice.py**, à la suite, ce second exemple :

```
# création d'une matrice 4 x 3
matrice = []
for n in range(4):
    ligne = [3*n+i for i in range (3)]
    matrice.append(ligne)
print("Matrice de 4 lignes et de 3 colonnes")
print(matrice)
```

Notez ci-dessous ce que l'on obtient dans la console :

*Matrice de 5 lignes et de 4 colonnes
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15], [16, 17, 18, 19]]*

Exercice 4 :

Analyser bien les deux exemples précédents ainsi que le résultat obtenu dans la console puis ajouter, toujours à la suite, le script suivant qu'il faudra compléter aux emplacements indiqués en utilisant les deux variables **nombreLignes** et **nombreColonnes**.

```
# création d'une matrice nombreLignes x nombreColonnes
matrice = []
nombreLignes = 9 # indiquer ici la valeur numérique qui correspond au nombre de lignes
nombreColonnes = 5 # indiquer ici la valeur numérique qui correspond au nombre de colonnes
for n in range(nombreLignes):
    ligne = [nombreColonnes*n+i for i in range(nombreColonnes)]
    matrice.append(ligne)
print("Matrice de %s lignes et de %s colonnes" %(nombreLignes, nombreColonnes))
```

Tester votre script en utilisant les valeurs des deux exemples pour valider son fonctionnement.

Le tester ensuite avec d'autres valeurs affectées aux deux variables **nombreLignes** et **nombreColonnes**.

Pour aller plus loin :

Modifier les deux lignes d'affectation des variables de façon à inviter l'utilisateur à saisir les valeurs numériques au clavier.
Les noter ci-dessous :

```
..... nombreLignes = int(input('Entrez un nombre entier'))
..... nombreColonnes = int(input('Entrez un second nombre entier'))
```

Dans la boucle, modifier l'instruction d'affectation de la variable ligne en utilisant **ligne = [[n,i] for i in range ...]**
Notez ci-dessous ce que l'on obtient dans la console avec une matrice 3 x 4 :

5 Les dictionnaires

5.1 DEFINITION

Les éléments d'une liste sont repérés par des indices 0, 1, 2, ... qui sont des entiers.

Une différence essentielle avec un dictionnaire, objet de type `dict`, est que les indices peuvent être du type `str`, `float` ou `tuple` (seulement si les n-uplets ne contiennent que des entiers, des flottants ou des p-uplets).

On appelle ces « indices » des **clés** et à chaque clé correspond une **valeur**. Ces clés ne sont pas ordonnées.

5.2 CREATION D'UN DICTIONNAIRE

Les éléments d'un dictionnaire sont des couples **clé-valeur**.

Un dictionnaire est créé avec des accolades, les différents couples étant séparés par des virgules.

La clé et la valeur correspondante d'un élément sont séparées par « : ».

Exemples :

```
>>> dico1 = {"A": 1, "B": 2, "C": 3, "D": 4}
>>> dico1
```

```
>>> dico2 = {"lundi": 1, "mardi": 2, "mercredi": 3, "jeudi": 4, "vendredi": 5, "samedi": 6, "dimanche": 7}
>>> dico2
```

```
>>> dico3 = {"oui": "yes", "non": "no", "et": "and", "ou": "or"}
>>> dico3
```

Dans la console, que constatez-vous à propos de l'ordre dans lequel python donne les couples clé-valeur ?

Elles sont dans l'ordre dans lequel nous l'avons entrée

La construction par compréhension utilisée pour construire des listes (chapitre 3.3) est permise pour les dictionnaires :

```
>>> dico4 = {str(n): 2**n for n in range(8)}
>>> dico4
```

```
>>> dico5 = {chr(65+i): chr(65+(i+7)%26) for i in range(26)}
>>> dico5
```

Explications sur dico5 :

On décale notre dictionnaire de 7 cases à l'arrière

On peut convertir une liste de listes à deux éléments en dictionnaire avec la fonction `dict` :

```
>>> liste = [['A', 1], ['B', 2], ['C', 3]]
>>> dico6 = dict(liste)
>>> dico6
```

On peut aussi créer un dictionnaire en utilisant la fonction `dict` avec des couples de la forme `clé=valeur`

```
>>> dico7 = dict(un=1, deux=2, trois=3, quatre=4)
>>> dico7
```

Exercice 5 :

Le script du programme **CodageMessage.py**, donné ci-dessous est à saisir, analyser, compléter (deux lignes) et tester.

```
def codeMot(mot):
    """ Codage d'un mot avec décalage de 7 caractères """
    dico1 = {chr(65+i): chr(65+(i+7)%26) for i in range(26)}
    motCode = ""
    for caractère in mot:
        motCode = motCode + dico1[caractère]
    return motCode

def main():
    """ Demande de saisir un mot constitué de lettres majuscules
    Affiche le mot initial et le mot codé correspondant """
    message = input("Entrez un mot composé de lettres majuscules")

    messageCode = codeMot(message)

    print("Le codage du mot %s est le mot %s" % (message, messageCode))

if __name__ == '__main__':
    main()
```

Exemples de tests effectués :

```
>>>
*** Console de processus distant Réinitialisée ***
Le codage du mot BONJOUR est le mot IMUQVBY
>>>
*** Console de processus distant Réinitialisée ***
Le codage du mot PYTHON est le mot WFAOVU
>>>
```

Lorsque le programme fonctionne, en faire une copie et la renommer **CodageDecodageMessage.py**

Modifier cette copie de façon à obtenir le fonctionnement suivant :

- Le programme demande à l'utilisateur de saisir un mot, qui peut être en majuscules ou pas.
- Le programme affiche le mot codé et le mot décodé sur deux lignes.
- Ces deux mots sont uniquement constitués de lettres majuscules.

*Conseil : ajouter une seconde fonction **decodeMot** qui assurera la fonction de décodage du mot saisi.*

Exemple de résultats à obtenir :

```
>>>
*** Console de processus distant Réinitialisée ***
Le codage du mot Bonjour est le mot IMUQVBY
Le décodage du mot Bonjour est le mot BONJOUR
>>>
*** Console de processus distant Réinitialisée ***
Le codage du mot IMUQVBY est le mot PCKXCIF
Le décodage du mot IMUQVBY est le mot BONJOUR
>>>
```

Consigner ci-dessous les lignes importantes du programme modifié :

5.3 UTILISATION DU DICTIONNAIRE

Accès aux éléments d'un dictionnaire

Pour accéder aux clés ou aux valeurs, nous disposons de méthodes : .keys() .values() .get(...)

Commande à saisir	Résultat obtenu	Opération réalisée
>>> dico = {"A": 1, "B": 2, "C": 3} >>> dico >>> dico.keys() >>> type(dico.keys())	{'A': 1, 'B': 2, 'C': 3} 'dict_keys' object <class 'dict_keys'>	Création d'un dictionnaire de trois éléments .keys() = montre les clés
>>> dico.values() >>> type(dico.values())	[1, 2, 3] 'dict_values' object <class 'dict_values'>	.values() = montre les valeurs
>>> dico.items() >>> type(dico.items())	(('A', 1), ('B', 2), ('C', 3)) 'dict_items' object <class 'dict_items'>	.items() = fais une liste avec ce qui est déjà dans le dico
>>> "A" in dico	True	in Il y a ?
>>> 1 in dico	False	demande si il y a 1 dans le dico
>>> 1 in dico.values()	False	Il y a 1 dans le dico valeur
>>> ("C", 3) in dico.items()	True	Il y a ("C", 3) dans la dico items
>>> for cle in dico: ... print(cle, end=' ')	A B C	on fait un affichage des clés
>>> dico["A"]	1	donne la valeur de la clé demandée
>>> dico ["D"] = 5 >>> dico	1 2 3 5	ajoute à son moment un exemple
>>> dico ["D"] = 4 >>> dico	1 2 3 4	modification d'une valeur - clé
>>> len(dico)	4	len(...) = donne le nombre de caract.
>>> val = dico.get("C") >>> print(val)	3	.get(...) = recherche de la valeur d'une clé
>>> val = dico.get("E") >>> print(val)	None	.get(...) = valeur non trouvée
>>> dico >>> del dico["D"] >>> dico		del() = suppression d'un couple
>>> dico.clear() >>> dico	{}	.clear() = vider le dico

Remarque : les éléments d'un dictionnaire peuvent être des listes ou des dictionnaires.

Copie d'un dictionnaire

Les comportements sont similaires à ceux rencontrés avec les listes, en particulier si les valeurs des couples sont des listes. Il est donc conseillé d'utiliser la fonction deepcopy du module copy pour être certain d'obtenir une « vraie » copie du dictionnaire.

Dictionnaire vs Liste

Un dictionnaire pourrait être remplacé par une liste de liste

```
dico = {"A": 1, "B": 2, "C": 3}
liste = [["A", 1], ["B", 2], ["C", 3]]
```

La recherche des éléments dans un dictionnaire est beaucoup plus rapide que la recherche dans une liste, le gain est important lorsque le nombre d'éléments l'est aussi. La recherche dans une liste nécessite parfois de la parcourir en intégralité.