Autres structures de données

Développeur Python

Sommaire

1. Ensembles.
2. Dictionnaires.
3. Fichiers.

1. Ensembles.

1. Ensembles.

Notion d’ensemble

- Structure de données non séquentielle implémentant la notion d’ensemble mathématique.

- Deux types d’ensembles :

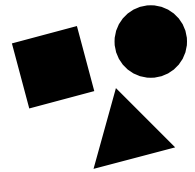
    1. La structure “set“ qui est muable.
    2. La structure “frozenset“ qui est immuable.

1. Ensembles.

Notion d’ensemble : remarques

- Par “ensemble mathématique“ on entend ici “collection d’éléments“ distincts. Lors de la création d’un ensemble ou lors de sa mise à jour, les doublons seront donc supprimés automatiquement.

- Cette structure étant non séquentielle, l’accès aux éléments ne se fera donc pas via leurs positions.



1. Ensembles.

Principales syntaxes pour créer un ensemble

```
mySet = {item1, item2, …, itemN}

mySet = set(otherStructure)

myFrozenset = frozenset(otherStructure)
```

1. Ensembles.

Opérations d'appartenance à un ensemble

Opération	*Résultat*
x in s	Teste si **x** appartient à **s**
x not in s	Teste si **x** n'appartient pas à **s**
len(s)	Nombre d'éléments de **s**
s.isdisjoint(t)	**True** si l'intersection de **s** et **t** est vide
s.issubset(t)	**True** si **s** est inclus dans **t**
s <= t	
s.issuperset(t)	**True** si **t** est inclus dans **s**
s >= t	

1. Ensembles.

Opérations classiques sur les ensembles

Opération	*Résultat*
s.union(t)	Union de **s** et **t**
s | t	
s.intersection(t)	Intersection de **s** et **t**
s & t	
s.difference(t)	Différence de **s** par **t**
s - t	
s.symmetric_difference(t)	Différence symétrique de **s** et **t**
s ^ t	

1. Ensembles.

Itération sur les éléments

- On peut itérer sur les éléments d'un "set" ou d'un "frozenset" :

```
for x in mySet:
    traitement de l'élément x
```

- Leurs éléments n'étant pas mémorisés de façon séquentielle, l'ordre de parcours sera par contre indéfini et imprévisible.

1. Ensembles.

Opérations de mises à jour spécifiques aux “set“

Opération	***Résultat***
s.update(t)	**s** mis à jour vers l’union de **s** et **t**
s |= t	
s.intersection_update(t)	**s** mis à jour vers l’intersection de **s** et **t**
s &= t	
s.difference_update(t)	**s** mis à jour vers la différence de **s** par **t**
s -= t	
s.symmetric_difference_update(t)	**s** mis à jour vers la différence symétrique de **s** et **t**
s ^= t	

1. Ensembles.

Opération d'ajouts et de suppressions d'éléments à un "set"

Opération	***Résultat***
s.add(x)	Ajoute l'élément **x** à **s**
s.remove(x)	Supprime l'élément **x** de **s**, retourne une erreur s'il est absent
s.discard(x)	Supprime l'élément **x** de **s** s'il est présent, ne retourne rien s'il est absent
s.pop()	Supprime et retourne un élément de **s** quelconque
s.clear()	Supprime tous les éléments de **s**

1. Ensembles.

Ensembles définis en compréhension : principe

- Le principe est exactement le même que celui des listes définies en compréhension :

```
{expression(x) for x in mySequence if condition}
```

1. Ensembles.

Ensembles définis en compréhension : exemple

```
mySequence = tuple(range(0,10))
mySet = {x**2 for x in mySequence if x%2==0}

print(mySet)
```

```
{0, 64, 4, 36, 16}
```

1. Ensembles.

2. Dictionnaires.

2. Dictionnaires.

Notion de dictionnaire

- Structure de données non séquentielle et muable permettant de mémoriser des couples clé : valeur.
- L'accès à une valeur se fait donc via sa clé et non pas grâce à sa position comme dans une séquence.
- Les clés doivent nécessairement être d'un type immuable (int, str, t-uple, etc.) et sont toutes différentes.
- Les valeurs sont par contre de n'importe quel type.

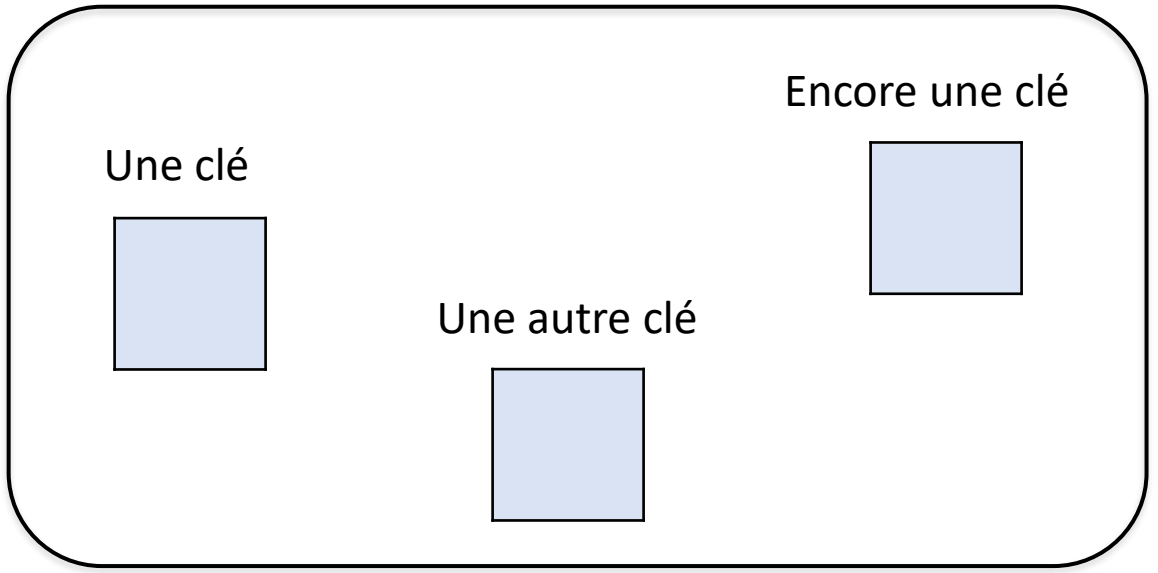
2. Dictionnaires.

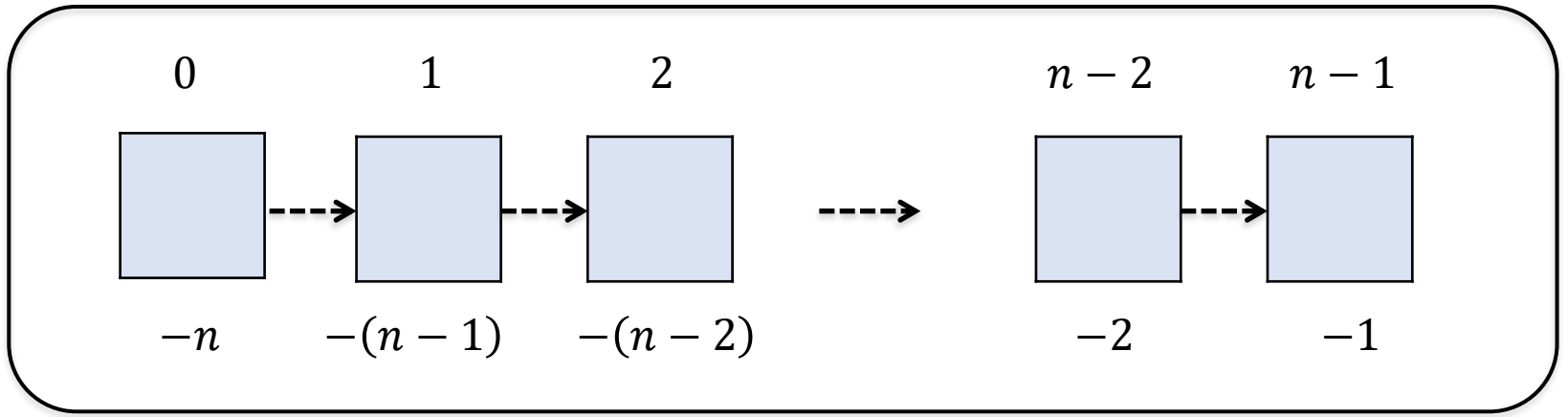
Représentation imagée d'un dictionnaire

Un dictionnaire

versus

une séquence





2. Dictionnaires.

Principales syntaxes pour créer un dictionnaire

```
myDict = {key1:value1, key2:value2, ..., keyN:valueN}

myDict = dict([(key1, value1), ..., (keyN, valueN)])

myDict = dict(((key1, value1), ..., (keyN, valueN)))

myDict = dict(zip([key1, ..., keyN], [value1, ..., valueN]))
```

2. Dictionnaires.

Ajout d'un couple clé : valeur ou modification d'une valeur

- Syntaxe :

```
myDict[myKey] = myValue
```

- Si "myKey" n'est pas déjà une clé du dictionnaire, il y a création du couple "myKey:Myvalue".

- Sinon, il y a remplacement de l'ancienne valeur associée à "myKey" par "myValue".

2. Dictionnaires.

Opérations classiques sur les dictionnaires

Opération	***Résultat***
x in d	Teste si la clé **x** appartient à **d**
x not in d	Teste si la clé **x** n'appartient pas à **d**
len(d)	Nombre d'éléments de **d**
d[x]	Si **x** est une clé, retourne **d[x]** sinon renvoie une erreur
del d[x]	Si **x** est une clé, supprime **d[x]** sinon renvoie une erreur
d.clear()	Supprime tous les éléments de **d**
d.update(e)	Met à jour **d** avec **e**

2. Dictionnaires.

Opérations de traitement des dictionnaires

Opération	*Résultat*
d.get(x)	si **x** est une clé retourne **d[x]** sinon **None**
d.get(x,y)	si **x** est une clé retourne **d[x]** sinon **y**
d.pop(x)	si **x** est une clé retourne **d[x]** et supprime le couple **x : d[x]**, sinon renvoie une erreur
d.pop(x,y)	si **x** est une clé retourne **d[x]** et supprime le couple **x : d[x]**, sinon retourne **y**
d.setdefault(x,y)	si **x** est une clé retourne **d[x]** sinon retourne **y** et insère le couple **x : y**

2. Dictionnaires.

Itération sur les clés d'un dictionnaire

- Deux syntaxes possibles :

```
for key in myDict:
    traitement de l'élément de clé key
```

```
for key in myDict.keys():
    traitement de l'élément de clé key
```

- L'ordre de parcours sera celui d'insertion des éléments dans le dictionnaire.

2. Dictionnaires.

Itération sur les valeurs et itération sur les couples clé : valeur

- Valeurs :

```
for value in myDict.values():
    traitement de la valeur value
```

- Couples clé : valeur

```
for key, value in myDict.items():
    traitement du couple key : value
```

2. Dictionnaires.

Dictionnaires définis en compréhension

- Le principe est exactement le même que celui des listes ou ensembles définis en compréhension :

```
{key(x, y):value(x, y) for x, y in mySequence if condition}
```

2. Dictionnaires.

Dictionnaires définis en compréhension : exemple

```
myDict1 = {chr(i+64):i for i in range(1, 6)}
myDict2 = {v:k for k, v in myDict1.items()}

print(myDict1)
print(myDict2)
```

```
{'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5}
{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E'}
```

2. Dictionnaires.

3. Fichiers.

3. Fichiers.

Objectif de la gestion de fichiers

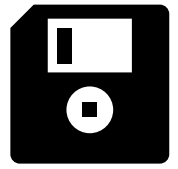
- Pouvoir conserver des données en mémoire de façon durable.
- Pour l'instant nos données ne sont disponibles que pendant l'exécution du programme.
- Dans notre contexte, un fichier sera donc pour nous un support pour conserver une masse de données.

3. Fichiers.

Deux principaux types de fichiers

- Fichier texte : fichier organisé sous la forme d'une suite de lignes, chacune étant composée d'un certain nombre de caractères et terminée par '\n'.

- Fichier binaire : suite d'octets, pouvant représenter toutes sortes de données.



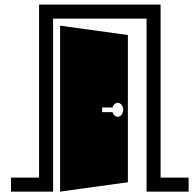
3. Fichiers.

Ouverture d'un fichier

- Syntaxe :

```
myFile = open('fileName', 'mode')
```

- Différents modes d'ouverture :
  - Lecture : 'r'
  - Écriture : 'w'
  - Ajout : 'a'

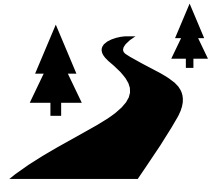


3. Fichiers.

Ouverture d'un fichier : remarque

- Par défaut on considère que le fichier est situé dans le même répertoire que notre script.

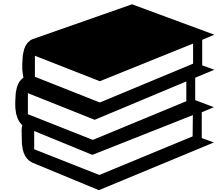
- Dans le cas contraire, on devra préciser son emplacement, en utilisant un chemin absolu (du style C:\ … sous windows) ou un chemin relatif à partir du répertoire courant.



3. Fichiers.

Les trois modes d'ouverture possible

- 'r' : lecture. On peut lire le contenu du fichier mais pas y écrire. Le fichier doit exister auparavant.
- 'w' : écriture. Si le fichier existait, son contenu est effacé, sinon il est créé.
- 'a' : ajout. Si le fichier existait on peut écrire à la fin de celui-ci sans effacer son contenu. Sinon il est créé.



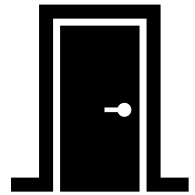
3. Fichiers.

Fermeture d’un fichier

- Syntaxe :

```
myFile.close()
```

- Permet de rendre le fichier disponible à d’autres applications.



3. Fichiers.

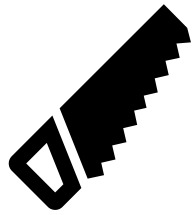
Lecture du contenu d'un fichier

Commande	***Résultat***
myFile.read()	Retourne l'intégralité du fichier sous la forme d'une chaîne de caractères
myFile.readline()	Retourne une ligne du fichier
myFile.readlines()	Retourne une liste constituée de toutes les lignes du fichier

3. Fichiers.

Lecture du contenu d’un fichier : remarque

- On peut passer un entier en paramètre à la fonction “read“ pour spécifier un nombre de caractères à lire.
- Cela permet d’éviter des débordements mémoire en cas de fichier volumineux.
- On sera lors positionné à cet endroit pour une éventuelle lecture ultérieure.



3. Fichiers.

Itération sur les lignes d'un fichier

```
for line in myFile:
    traitement de la ligne line
```

3. Fichiers.

Écriture dans un fichier

- Syntaxe :

```
myFile.write(myText)
```

- Cette commande est la même que l'on soit en mode écriture ou en mode ajout.

3. Fichiers.

Écriture dans un fichier : remarques

- Si on utilise plusieurs fois cette procédure, on écrira les différents textes les uns à la suite des autres.

- Le paramètre “myText“ est nécessairement du type ‘str’, il faut donc éventuellement convertir les variables numériques.

3. Fichiers.