

Structures de données

Piles - listes - tableaux - chaînes de caractères - fichiers

Germain Gondor

LYCÉE CARNOT (DIJON), 2016 - 2017

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Introduction

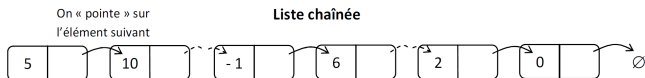
Les listes (pas au sens python) sont des données composites. Elles rassemblent dans une seule structure un ensemble d'entités plus simples qui peuvent être de types différents. En général, on traite plutôt de listes d'éléments de même type, par exemple des listes de nombres.

La liste est un objet de la catégorie des séquences, lesquelles sont des collections ordonnées d'éléments. Cela signifie simplement que les éléments d'une liste sont toujours disposés dans un certain ordre. Contrairement aux chaînes de caractères, les listes sont des séquences modifiables.

On peut implémenter une liste de deux manières :

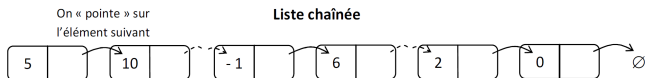
On peut implémenter une liste de deux manières :

- **par une liste chaînée** : Chaque maillon est un objet dans lequel on stocke l'élément et une information pour trouver le suivant (voire aussi le précédent). C'est une structure de données récursive. La liste chaînée n'a pas de taille fixée à l'avance. Seul l'objet n maillon z a une certaine dimension.



On peut implémenter une liste de deux manières :

- par une liste chaînée** : Chaque maillon est un objet dans lequel on stocke l'élément et une information pour trouver le suivant (voire aussi le précédent). C'est une structure de données récursive. La liste chaînée n'a pas de taille fixée à l'avance. Seul l'objet n maillon z a une certaine dimension.



- par un tableau** : Chaque élément de la liste peut être désigné par sa place dans la séquence, à l'aide d'un index. Pour accéder à un élément déterminé, on utilise le nom de la variable qui contient la liste et on lui accole entre deux crochets, l'index numérique qui correspond à sa position dans la chaîne. Le tableau a une dimension fixe. Certains langages (comme Python) permettent de modifier sa taille au cours de son utilisation.

Indices	1	2	...	i	$i + 1$...	$n - 1$	n
Tableau	5	10	...	-1	6	...	2	0

Sommaire

1 Introduction

2 Les piles et les files

- Définitions
- Principales fonctions associées
- Piles et files définies par listes chaînées
- Pile définie par un tableau P de n éléments
- File définie par un tableau F de n éléments

3 Les listes Python

4 Les ensembles

5 Les dictionnaires

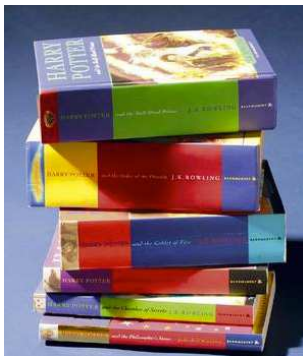
6 Les tableaux Python

Définitions

Les piles et les files sont des listes particulières : on accède aux éléments par les extrémités, c'est-à-dire au début ou à la fin.

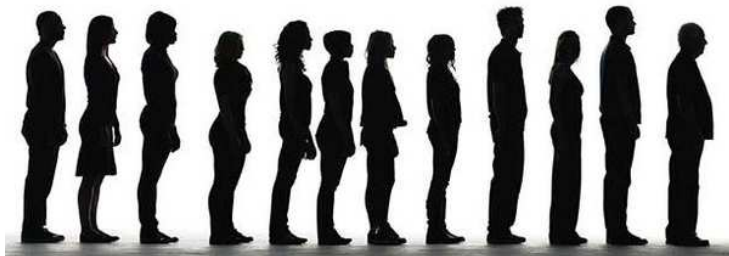
les piles (*stacks*)

Le premier empilé est le dernier à être dépilé
LIFO (Last In first Out)



les files

le premier entré est le premier à sortir
FIFO (First In First Out)



Principales fonctions associées

Pour les piles, les principales fonctions associées aux piles sont :

Principales fonctions associées

Pour les piles, les principales fonctions associées aux piles sont :

- ajouter un sommer un élément x : **empiler** `Pile.append(x)`

Principales fonctions associées

Pour les piles, les principales fonctions associées aux piles sont :

- ajouter un sommer un élément x : **empiler** `Pile.append(x)`
- supprimer du sommet le dernier élément de la pile : **dépiler**
`Pile.pop()`.

REMARQUE: on peut récupérer cet élément en l'affectant à une variable : `x=Pile.pop()`

Principales fonctions associées

Pour les piles, les principales fonctions associées aux piles sont :

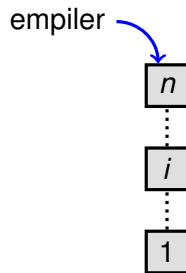
- ajouter un sommer un élément x : **empiler** `Pile.append(x)`
- supprimer du sommet le dernier élément de la pile : **dépiler**
`Pile.pop()`.
REMARQUE: on peut récupérer cet élément en l'affectant à une variable : `x=Pile.pop()`
- tester si la pile est vide : **est_vide** `Pile==[]`

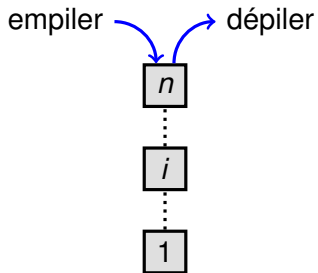
Principales fonctions associées

Pour les piles, les principales fonctions associées aux piles sont :

- ajouter un sommer un élément x : **empiler** `Pile.append(x)`
- supprimer du sommet le dernier élément de la pile : **dépiler** `Pile.pop()`.
REMARQUE: on peut récupérer cet élément en l'affectant à une variable : `x=Pile.pop()`
- tester si la pile est vide : **est_vide** `Pile==[]`
- tester si la pile est pleine : **est_pleine** avertissement par `overflow` ou `stackoverflow`







Pour les files, les principales fonctions associées aux piles sont :

Pour les files, les principales fonctions associées aux piles sont :

- ajouter en queue : **enfiler**

Pour les files, les principales fonctions associées aux piles sont :

- ajouter en queue : **enfiler**
- supprimer en tête : **défiler**

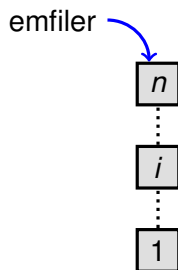
Pour les files, les principales fonctions associées aux piles sont :

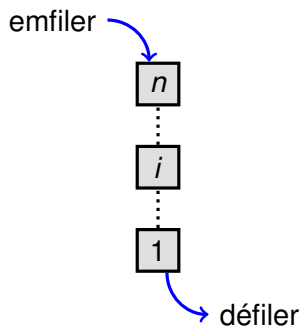
- ajouter en queue : **enfiler**
- supprimer en tête : **défiler**
- tester si la pile est vide : **est_vider** `File==[]`

Pour les files, les principales fonctions associées aux piles sont :

- ajouter en queue : **enfiler**
- supprimer en tête : **défiler**
- tester si la pile est vide : **est_vide** `File==[]`
- tester si la pile est pleine : **est_pleine** avertissement par `overflow` **OU** `stackoverflow`







Piles et files définies par listes chaînées

Voir Tp *ALGO-PROG-TP3 : structures de données.*

Pile définie par un tableau P de n éléments

Pile définie par un tableau P de n éléments

- l'ordre est déterminé par les indices des éléments du tableau $P[1:n]$

Pile définie par un tableau P de n éléments

- l'ordre est déterminé par les indices des éléments du tableau $P[1:n]$
- le tableau possède un attribut `sommet[P]` qui indexe l'élément le plus récemment inséré

Pile définie par un tableau P de n éléments

- l'ordre est déterminé par les indices des éléments du tableau $P[1:n]$
- le tableau possède un attribut `sommet[P]` qui indexe l'élément le plus récemment inséré
- la pile se constitue des éléments $P[1:sommet[P]]$ où $P[1]$ est l'élément de base de la pile (le plus ancien) et $P[sommet[P]]$ est l'élément situé au sommet (le plus récent)

Pile définie par un tableau P de n éléments

- l'ordre est déterminé par les indices des éléments du tableau $P[1:n]$
- le tableau possède un attribut `sommet[P]` qui indexe l'élément le plus récemment inséré
- la pile se constitue des éléments $P[1:sommet[P]]$ où $P[1]$ est l'élément de base de la pile (le plus ancien) et $P[sommet[P]]$ est l'élément situé au sommet (le plus récent)
- quand `sommet[P] = 0`, la pile est vide

Pile définie par un tableau P de n éléments

- l'ordre est déterminé par les indices des éléments du tableau $P[1:n]$
- le tableau possède un attribut `sommet[P]` qui indexe l'élément le plus récemment inséré
- la pile se constitue des éléments $P[1:sommet[P]]$ où $P[1]$ est l'élément de base de la pile (le plus ancien) et $P[sommet[P]]$ est l'élément situé au sommet (le plus récent)
- quand `sommet[P] = 0`, la pile est vide
- Si `sommet[P]` est supérieur à n , on dit que la pile déborde
REMARQUE: Dépiler une pile vide provoque une erreur ; on parle de débordement négatif.

Fonction Pile_vider

PileVide (P)

- 1: **si** sommet[P] = 0 **alors**
- 2: **renvoi:** Vrai
- 3: **sinon**
- 4: **renvoi:** Faux
- 5: **fin si**

Fonction empiler

Empiler (P,x)

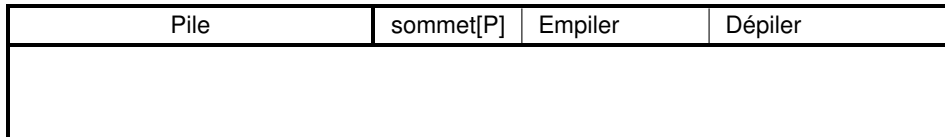
1: $S[P] \leftarrow \text{sommet}[P]+1$

2: $P[\text{Sommet}[P]] \leftarrow x$

Fonction dépiler

Dépiler (P)

- 1: **si** PileVide(P) **alors**
- 2: "débordement négatif" (Erreur)
- 3: **sinon**
- 4: sommet[P] \leftarrow sommet[P]-1
- 5: **renvoi:** P[sommet[P]+1]
- 6: **fin si**



Pile							sommet[P]	Empiler	Dépiler	
	1	2	3	4	5	6	7			
P	15	6	2	9				4	on place en 5	on récupère 9

Pile		sommet[P]	Empiler	Dépiler						
1 2 3 4 5 6 7 <i>P</i> <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>15</td><td>6</td><td>2</td><td>9</td><td></td><td></td><td></td> </tr> </table>	15	6	2	9				4	on place en 5	on récupère 9
15	6	2	9							
1 2 3 4 5 6 7 <i>P</i> <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>15</td><td>6</td><td>2</td><td>9</td><td>17</td><td>3</td><td>12</td> </tr> </table>	15	6	2	9	17	3	12	7	débordement !	on récupère 12
15	6	2	9	17	3	12				
1 2 3 4 5 6 7 <i>P</i> <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>								0	on place en 1	débordement négatif

File définie par un tableau F de n éléments

File définie par un tableau F de n éléments

- la file a un attribut $tête [F]$ qui repère sa tête

File définie par un tableau F de n éléments

- la file a un attribut `tête` [F] qui repère sa tête
- la file a un attribut `queue` [F] qui indexe le prochain emplacement pouvant conserver un nouvel élément.

File définie par un tableau F de n éléments

- la file a un attribut $tête[F]$ qui repère sa tête
- la file a un attribut $queue[F]$ qui indexe le prochain emplacement pouvant conserver un nouvel élément.
- les éléments de la file sont repérés par $tête[F]$, $tête[F+1]$, \dots , $queue[F]-1$

File définie par un tableau F de n éléments

- la file a un attribut $tête[F]$ qui repère sa tête
- la file a un attribut $queue[F]$ qui indexe le prochain emplacement pouvant conserver un nouvel élément.
- les éléments de la file sont repérés par $tête[F]$, $tête[F+1]$, ..., $queue[F]-1$
- l'emplacement 1 suit l'emplacement n dans un ordre circulaire

File définie par un tableau F de n éléments

- la file a un attribut $tête[F]$ qui repère sa tête
- la file a un attribut $queue[F]$ qui indexe le prochain emplacement pouvant conserver un nouvel élément.
- les éléments de la file sont repérés par $tête[F]$, $tête[F+1], \dots, queue[F]-1$
- l'emplacement 1 suit l'emplacement n dans un ordre circulaire
- quand $tête[F] = queue[F]$, la file est vide
REMARQUE: au départ, on a $tête[F] = queue[F] = 1$

File définie par un tableau F de n éléments

- la file a un attribut $tête[F]$ qui repère sa tête
- la file a un attribut $queue[F]$ qui indexe le prochain emplacement pouvant conserver un nouvel élément.
- les éléments de la file sont repérés par $tête[F]$, $tête[F+1], \dots, queue[F]-1$
- l'emplacement 1 suit l'emplacement n dans un ordre circulaire
- quand $tête[F] = queue[F]$, la file est vide
REMARQUE: au départ, on a $tête[F] = queue[F] = 1$
- quand $tête[F] = queue[F] + 1$, la file est pleine.

Fonction Enfiler

Enfiler(F,x)

- 1: $F[\text{queue}[F]] \leftarrow x$
- 2: **si** $\text{queue}[F] = \text{longueur}[F]$ **alors** $\text{queue}[F] \leftarrow 1$
- 3: **sinon** $\text{queue}[F] \leftarrow \text{queue}[F] + 1$
- 4: **fin si**

Fonction Défiler

Défiler(F)

- 1: $x \leftarrow F[\text{tête}[F]]$
- 2: **si** $\text{tête}[F] = \text{longueur}[F]$ **alors** $\text{tête}[F] \leftarrow 1$
- 3: **sinon** $\text{tête}[F] \leftarrow \text{tête}[F] + 1$
- 4: **fin sirenvoy:** x

File	t	q	action dente	précé-
------	---	---	-----------------	--------

t	q	action dente	précé-
---	---	-----------------	--------

File												t	q	action précédente	
	1	2	3	4	5	6	7	8	9	10	11	12	7	12	Les éléments 15, 6, 2, 8 et 4 ont été enfilés
P							15	6	2	8	4				

File		t	q	action précédente																								
P	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>15</td><td>6</td><td>2</td><td>8</td><td>4</td><td></td> </tr> </table>	1	2	3	4	5	6	7	8	9	10	11	12							15	6	2	8	4		7	12	Les éléments 15, 6, 2, 8 et 4 ont été enfilés
1	2	3	4	5	6	7	8	9	10	11	12																	
						15	6	2	8	4																		
P	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td> </tr> <tr> <td>3</td><td>5</td><td></td><td></td><td></td><td></td><td>15</td><td>6</td><td>2</td><td>8</td><td>4</td><td>17</td> </tr> </table>	1	2	3	4	5	6	7	8	9	10	11	12	3	5					15	6	2	8	4	17	7	3	Les éléments 17, 3 et 5 ont été enfilés
1	2	3	4	5	6	7	8	9	10	11	12																	
3	5					15	6	2	8	4	17																	

File												t	q	action précédente		
	1	2	3	4	5	6	7	8	9	10	11	12				
P							15	6	2	8	4					Les éléments 15, 6, 2, 8 et 4 ont été enfilés
	1	2	3	4	5	6	7	8	9	10	11	12				
P	3	5					15	6	2	8	4	17				Les éléments 17, 3 et 5 ont été enfilés
	1	2	3	4	5	6	7	8	9	10	11	12				
P	3	5					15	6	2	8	4	17				l'élément 15 à été défilé

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python**
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Les listes Python

Beaucoup plus pratique que les listes chaînées qui sont briques de bases de la programmation, les listes en python sont des mutables permettant de faire :

- ajouter un élément en fin de liste : `L.append(5)`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`
- assembler des listes bout à bout (concaténation) : `L+[1, 2, 3]`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`
- assembler des listes bout à bout (concaténation) : `L+[1, 2, 3]`
- obtenir une liste composée de n fois la liste `L` mise bout à bout :
`L*3`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`
- assembler des listes bout à bout (concaténation) : `L+[1, 2, 3]`
- obtenir une liste composée de n fois la liste `L` mise bout à bout :
`L*3`
- des listes de ce qu'on veut :

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`
- assembler des listes bout à bout (concaténation) : `L+[1, 2, 3]`
- obtenir une liste composée de n fois la liste `L` mise bout à bout :
`L*3`
- des listes de ce qu'on veut :
 - liste de chaînes de caractère :
`L=['Je ne ', 'Peppa Pig', 'pas', 'suis']`

- ajouter un élément en fin de liste : `L.append(5)`
- récupérer le dernier élément mais en l'enlevant : `x=L.pop()`
- supprimer l'élément i d'une liste `L` : `del L[i]`
- accéder directement à l'élément n d'une liste : `L[n]`
- changer directement (mutabilité) un élément n d'une liste :
`L[n]=3`
- accéder à plusieurs éléments consécutifs (slicing) : `L[1:3]`
- connaître la longueur de la liste: `len(L)`
- assembler des listes bout à bout (concaténation) : `L+[1, 2, 3]`
- obtenir une liste composée de n fois la liste `L` mise bout à bout :
`L*3`
- des listes de ce qu'on veut :
 - liste de chaînes de caractère :
`L=['Je ne ', 'Peppa Pig', 'pas', 'suis']`
 - listes de listes : `[1, 2, [3, 4, [5, 8], [2, 3, 4], 3], 1]`

			4			8	7	
	4	7		9	2		5	
2			6				3	
9	7		5			2		3
5		8		2	4	7		6
6		4			7		8	5
	9		3		8			7
		3	2	4		1	6	
	1	2					9	

```
import numpy as np
mat=np.loadtxt('sudoku-0.txt',unpack='true').transpose()

Mat=[]
for i in range(9):
    Mat.append([[], [], [], [], [], [], [], [], []])

Tai=np.zeros((9,9),int)
for i in range(9):
    for j in range(9):
        Tai[i][j]=9
        if mat[i,j]>0:
            Mat[i][j]=[int(mat[i,j])]
            Tai[i][j]=1
```

REMARQUE: Attention à la copie :

REMARQUE: Attention à la copie :

- écrire `L1=L2` conduit à changer les éléments de `L1` lorsqu'on change les éléments de `L2`

REMARQUE: Attention à la copie :

- écrire `L1=L2` conduit à changer les éléments de `L1` lorsqu'on change les éléments de `L2`
- pour éviter ce problème, on peut écrire :

REMARQUE: Attention à la copie :

- écrire `L1=L2` conduit à changer les éléments de `L1` lorsqu'on change les éléments de `L2`
- pour éviter ce problème, on peut écrire :
 - `L1=L.copy()`

REMARQUE: Attention à la copie :

- écrire `L1=L2` conduit à changer les éléments de `L1` lorsqu'on change les éléments de `L2`
- pour éviter ce problème, on peut écrire :
 - `L1=L.copy()`
 - `L1=list(L)`

REMARQUE: Attention à la copie :

- écrire `L1=L2` conduit à changer les éléments de `L1` lorsqu'on change les éléments de `L2`
- pour éviter ce problème, on peut écrire :
 - `L1=L.copy()`
 - `L1=list(L)`
- attention la copie est superficielle.

```
>>> L=[[1],[2],[3]]
>>> L1=L.copy()
>>> L[0][0]=0
>>> L
[[0], [2], [3]]
>>> L1
[[0], [2], [3]]
>>> L[0]=[7]
>>> L1
[[0], [2], [3]]
>>> L
[[7], [2], [3]]
```

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles**
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Les ensembles

On dispose en python d'un type ensemble : set. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set ()`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.
- **Inclusion** : `s1.issubset(s2)` ou `s1 <= s2`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.
- **Inclusion** : `s1.issubset(s2)` ou `s1 <= s2`
- **Union** : `s1.union(s2)` ou `s1 | s2`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.
- **Inclusion** : `s1.issubset(s2)` ou `s1 <= s2`
- **Union** : `s1.union(s2)` ou `s1 | s2`
- **Intersection** : `s1.intersection(s2)` ou `s1 & s2`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.
- **Inclusion** : `s1.issubset(s2)` ou `s1 <= s2`
- **Union** : `s1.union(s2)` ou `s1 | s2`
- **Intersection** : `s1.intersection(s2)` ou `s1 & s2`
- **Différence** : `s1.difference(s2)` ou `s1 - s2`

Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- **Ensemble vide** : `{}` ou `set()`
- **Affectation** : `s={1, 2, 3}` affecte l'ensemble `{1, 2, 3}`.
- **Cardinal** : `len(s)`
- **Appartenance** : `x in s, x not in s`.
- **Inclusion** : `s1.issubset(s2)` ou `s1 <= s2`
- **Union** : `s1.union(s2)` ou `s1 | s2`
- **Intersection** : `s1.intersection(s2)` ou `s1 & s2`
- **Différence** : `s1.difference(s2)` ou `s1 - s2`
- **et aussi** : `s.add(x), s.remove(x)...`

Pour la copie, les mêmes règles que pour les listes s'appliquent !

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires**
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Les dictionnaires

Ça existe en python mais c'est hors programme.

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python**
- 7 Chaînes de caractères
- 8 Manipulation de fichiers

Les tableaux Python

Les tableaux python s'obtiennent avec **numpy** à partir de la commande `array`. Nous les étudierons quand vous aurez quelques notions d'algèbre linéaire ...

En attendant, on utilisera des listes de listes à la place de tableau à plusieurs dimension. Ainsi, pour accéder à l'élément de la ligne i du tableau T et de la colonne j , on utilisera la liste L telle que :

$$T[i, j] = L[i][j]$$

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères**
- 8 Manipulation de fichiers

Chaînes de caractères

Chaîne de caractères (*str* de string en anglais) sont des suites finies de caractères. Elles sont délimitées par des apostrophes `'`, des guillemets `"` ou des triples guillemets `"""` si on veut aller à la ligne à l'intérieur.

Les renvois à la ligne sont obtenus avec `\n`, les sauts de ligne avec `\r` et `\t` affiche une tabulation. De même, `\'` et `\"` permettent de faire apparaître les caractères.

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne s en plaçant l'indice entre crochets $s[i]$.

REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne s en plaçant l'indice entre crochets $s[i]$.

REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...

- extraire des caractères d'une chaîne pour en faire une sous-chaîne $s[i:j]$

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne s en plaçant l'indice entre crochets $s[i]$.

REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...

- extraire des caractères d'une chaîne pour en faire une sous-chaîne $s[i:j]$
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur +

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne `s` en plaçant l'indice entre crochets `s[i]`.

REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...

- extraire des caractères d'une chaîne pour en faire une sous-chaîne `s[i:j]`
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur `+`
- calculer la longueur d'une chaîne grâce à l'opérateur `len()`

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne `s` en plaçant l'indice entre crochets `s[i]`.
REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...
- extraire des caractères d'une chaîne pour en faire une sous-chaîne `s[i:j]`
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur `+`
- calculer la longueur d'une chaîne grâce à l'opérateur `len()`
- convertir des objets en chaîne de caractère grâce à l'instruction `str()`

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne `s` en plaçant l'indice entre crochets `s[i]`.
REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...
- extraire des caractères d'une chaîne pour en faire une sous-chaîne `s[i:j]`
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur `+`
- calculer la longueur d'une chaîne grâce à l'opérateur `len()`
- convertir des objets en chaîne de caractère grâce à l'instruction `str()`
- évaluer des chaînes de caractères grâce à l'instruction `eval()`

Les éléments ne sont pas mutables mais beaucoup de propriétés des listes sont partagées avec les chaînes de caractères :

- accéder au $i^{\text{ème}}$ caractère d'une chaîne `s` en plaçant l'indice entre crochets `s[i]`.
REMARQUE: la numérotation commence à 0 et elle accepte des indices négatifs : -1 donne le dernier caractère, -2 l'avant dernier,...
- extraire des caractères d'une chaîne pour en faire une sous-chaîne `s[i:j]`
- concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur `+`
- calculer la longueur d'une chaîne grâce à l'opérateur `len()`
- convertir des objets en chaîne de caractère grâce à l'instruction `str()`
- évaluer des chaînes de caractères grâce à l'instruction `eval()`
- faire des test d'inclusion avec l'opérateur `in`

Enfin, pour une chaîne de caractère `chaîne` :

Enfin, pour une chaîne de caractère `chaine` :

- supprimer les retours à la ligne : `chaine.rstrip()`

Enfin, pour une chaîne de caractère `chaine` :

- supprimer les retours à la ligne : `chaine.rstrip()`
- mettre en majuscules : `chaine.upper()`

Enfin, pour une chaîne de caractère `chaine` :

- supprimer les retours à la ligne : `chaine.rstrip()`
- mettre en majuscules : `chaine.upper()`
- mettre en minuscules : `chaine.lower()`

Sommaire

- 1 Introduction
- 2 Les piles et les files
- 3 Les listes Python
- 4 Les ensembles
- 5 Les dictionnaires
- 6 Les tableaux Python
- 7 Chaînes de caractères
- 8 Manipulation de fichiers**

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`
- fermeture : `monfichier.close()`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`
- fermeture : `monfichier.close()`
- lire la ligne suivante : `monfichier.readline()`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`
- fermeture : `monfichier.close()`
- lire la ligne suivante : `monfichier.readline()`
- lire toutes les lignes : `for ligne in monfichier.readlines() :`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`
- fermeture : `monfichier.close()`
- lire la ligne suivante : `monfichier.readline()`
- lire toutes les lignes : `for ligne in monfichier.readlines() :`
- écrire dans le fichier : `monfichier.write('blahblah.. blah')`

Manipulation de fichiers

Pour lire ou sauvegarder du contenu, il est possible d'avoir recours à des fichiers :

- ouvrir en lecture : `monfichier = open('lenom.txt', 'r')`
- ouvrir en écriture : `monfichier = open('lenom.txt', 'w')`
- fermeture : `monfichier.close()`
- lire la ligne suivante : `monfichier.readline()`
- lire toutes les lignes : `for ligne in monfichier.readlines() :`
- écrire dans le fichier : `monfichier.write('blahblah.. blah')`
- écrire dans le fichier et passer à la ligne suivante : `monfichier.write('blahblah.. blah'+'\n')`