Algorithmique et Programmation

Types de données

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes
http://www.math-info.univ-paris5.fr/∽bonzon/

Types de données

1. Les chaînes de caractères
2. Les listes
3. Les ensembles
4. Pour conclure

Les chaînes de caractères

Chaîne de caractères

Chaîne de caractères

Une chaîne de caractères est une séquence de caractères. Son type est `str`.

Chaîne de caractères

Chaîne de caractères

Une chaîne de caractères est une séquence de caractères. Son type est `str`.

- La chaîne vide ne contient aucun caractère. Elle est notée `""` ou `''`
  - Attention à ne pas confondre la chaîne vide avec la chaîne contenant un espace ! (`" "` ou `' '`)
- Python ne fait pas de différence entre un caractère et une chaîne de caractères
- Attention de ne pas confondre l'entier `123` et la chaîne de caractères `"123"`
- Attention à la casse : `'Avion'` et `'avion'` sont deux chaînes de caractères différentes

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \rightarrow \texttt{int}$$

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

```
>>> len("vendredi")
8
```

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

```
>>> len("vendredi")
8
>>> chaine = "Heureux qui, comme Ulysse, a fait un beau voyage"
>>> len(chaine)
48
```

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

```
>>> len("vendredi")
8
>>> chaine = "Heureux qui, comme Ulysse, a fait un beau voyage"
>>> len(chaine)
48
>>> len("48")
2
```

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

```
>>> len("vendredi")
8
>>> chaine = "Heureux qui, comme Ulysse, a fait un beau voyage"
>>> len(chaine)
48
>>> len("48")
2
>>> len("")
0
```

Longueur

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

```
>>> len("vendredi")
8
>>> chaine = "Heureux qui, comme Ulysse, a fait un beau voyage"
>>> len(chaine)
48
>>> len("48")
2
>>> len("")
0
>>> len(" ")
1
```

Concaténation

Concaténation de chaînes de caractères

L'opérateur + permet de concaténer les chaînes de caractères.

Signature :

$$+ :: \texttt{str} \times \texttt{str} \to \texttt{str}$$

Concaténation

Concaténation de chaînes de caractères

L'opérateur + permet de concaténer les chaînes de caractères.

Signature :

$$+ :: \texttt{str} \times \texttt{str} \rightarrow \texttt{str}$$

```
>>> "ven" + "dre" + "di"
'vendredi'
```

Concaténation

Concaténation de chaînes de caractères

L'opérateur + permet de concaténer les chaînes de caractères.

Signature :

$$+ :: \texttt{str} \times \texttt{str} \rightarrow \texttt{str}$$

```
>>> "ven" + "dre" + "di"
'vendredi'
#Différence entre concaténation et addition
>>> "1" + "2"
'12'
>>> 1 + 2
3
```

Concaténation

Concaténation de chaînes de caractères

L'opérateur + permet de concaténer les chaînes de caractères.

Signature :

$$+ :: \texttt{str} \times \texttt{str} \rightarrow \texttt{str}$$

```
>>> "ven" + "dre" + "di"
'vendredi'
#Différence entre concaténation et addition
>>> "1" + "2"
'12'
>>> 1 + 2
3
#La chaîne vide est l'élement neutre
>>> "bonne" + "" + "journée"
'bonnejournée'
>>> "bonne" + " " + "journée"
'bonne journée'
```

Exemple : construction par répétition

Problème : construire une chaîne de caractères formée de n répétitions d'une chaîne donnée

Exemple : construction par répétition

Problème : construire une chaîne de caractères formée de n répétitions d'une chaîne donnée

```
chaine = input("Chaîne de caractères à répéter : ")
n = int(input("Combien de fois voulez-vous la répéter? "))

res = "" #chaine résultat

for var in range(1, n + 1) :
    res = res + chaine
print(res)
```

Exemple : construction par répétition

Problème : construire une chaîne de caractères formée de n répétitions d'une chaîne donnée

```
chaine = input("Chaîne de caractères à répéter : ")
n = int(input("Combien de fois voulez-vous la répéter? "))

res = "" #chaine résultat

for var in range(1, n + 1) :
    res = res + chaine
print(res)
```

```
Chaîne de caractères à répéter : zut!
Combien de fois voulez-vous la répéter? 3
zut! zut! zut!
```

Simulation de boucle : construction par répétition

```
chaine = input("Chaîne de caractères à répéter : ")
n = int(input("Combien de fois voulez-vous la répéter? "))

res = "" #chaine résultat

for var in range(1, n + 1) :
    res = res + chaine
print(res)
```

Pour chaine = "zut ! ", $n = 3$:

tour de boucle	variable var	variable res
entrée	-	""
tour 1	1	"zut! "
tour 2	2	"zut! zut! "
tour 3	3	"zut! zut! zut! "

Duplication

La répétition de chaînes de caractères peut être effectuée plus simplement grâce à l'opérateur de duplication :

Duplication de chaînes de caractères

L'opérateur `*` permet de dupliquer les chaînes de caractères.

Signature :

$$* :: \texttt{str} \times \texttt{int} \to \texttt{str}$$

Duplication

La répétition de chaînes de caractères peut être effectuée plus simplement grâce à l'opérateur de duplication :

Duplication de chaînes de caractères

L'opérateur * permet de dupliquer les chaînes de caractères.

Signature :

$$* :: \texttt{str} \times \texttt{int} \to \texttt{str}$$

```
>>> "zut! " * 3
'zut! zut! zut! '
>>> "Boutros " * 2 + "Ghali"
'Boutros Boutros Ghali'
```

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> chaine = "Rayons X"
>>> 'a' in chaine
True
```

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> chaine = "Rayons X"
>>> 'a' in chaine
True
>>> 'ray' in chaine
False
```

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> chaine = "Rayons X"
>>> 'a' in chaine
True
>>> 'ray' in chaine
False
>>> 'Ray' in chaine
True
```

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> chaine = "Rayons X"
>>> 'a' in chaine
True
>>> 'ray' in chaine
False
>>> 'Ray' in chaine
True
>>> chaine in 'Ray'
False
```

Appartenance

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> chaine = "Rayons X"
>>> 'a' in chaine
True
>>> 'ray' in chaine
False
>>> 'Ray' in chaine
True
>>> chaine in 'Ray'
False
>>> 'ryn' in chaine
False
```

Indexation des chaînes de caractères

Indexation simple

L'indice d'un caractère dans une chaîne est sa position dans la chaîne.

Les indices sont numérotés à partir de 0.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

```
>>> chaine = "Rayons X"
```

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

```
>>> chaine = "Rayons X"
>>> chaine[0]
'R'
```

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

```
>>> chaine = "Rayons X"
>>> chaine[0]
'R'
>>> chaine[6]
' '
```

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

```
>>> chaine = "Rayons X"
>>> chaine[0]
'R'
>>> chaine[6]
' '
>>> chaine[3]
'o'
```

Indexation des chaînes de caractères

Indice d'un caractère dans une chaîne

`chaine[i]` est le caractère en position *i* dans la chaine `chaine`.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

```
>>> chaine = "Rayons X"
>>> chaine[0]
'R'
>>> chaine[6]
' '
>>> chaine[3]
'o'
>>> chaine[9]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine = "Rayons X"
```

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine = "Rayons X"
>>> chaine[-3]
's'
```

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine = "Rayons X"
>>> chaine[-3]
's'
>>> chaine[-8]
'R'
```

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine = "Rayons X"
>>> chaine[-3]
's'
>>> chaine[-8]
'R'
>>> chaine[-12]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices i (inclus) et j (exclus)

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices *i* (inclus) et *j* (exclus)

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices *i* (inclus) et *j* (exclus)

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine[0 : 4]
'Rayo'
```

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices *i* (inclus) et *j* (exclus)

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine[0 : 4]
'Rayo'
>>> chaine[4 : 7]
'ns '
```

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices *i* (inclus) et *j* (exclus)

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine[0 : 4]
'Rayo'
>>> chaine[4 : 7]
'ns '
>>> chaine[5 : 6]
's'
```

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices *i* (inclus) et *j* (exclus)

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> chaine[0 : 4]
'Rayo'
>>> chaine[4 : 7]
'ns '
>>> chaine[5 : 6]
's'
>>> chaine[-5 : -2]
'ons'
```

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> 'vendre' + 'di' == 'vendredi'
True
```

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> 'vendre' + 'di' == 'vendredi'
True
>>> 'vendredi' == 'Vendredi'
False
```

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> 'vendre' + 'di' == 'vendredi'
True
>>> 'vendredi' == 'Vendredi'
False
>>> 'vendred i' == 'vendredi'
False
```

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> 'vendre' + 'di' == 'vendredi'
True
>>> 'vendredi' == 'Vendredi'
False
>>> 'vendred i' == 'vendredi'
False
>>> 'vendredi' != 'Vendredi'
True
```

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$== :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$!= :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

```
>>> 'vendre' + 'di' == 'vendredi'
True
>>> 'vendredi' == 'Vendredi'
False
>>> 'vendred i' == 'vendredi'
False
>>> 'vendredi' != 'Vendredi'
True
>>> 'vendredi' == "vendredi"
True
```

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule
- `s.title()` : retourne la chaine `s` où la première lettre de chaque mot est en majuscule, les autres en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule
- `s.title()` : retourne la chaine `s` où la première lettre de chaque mot est en majuscule, les autres en minuscule
- `s.swapcase()` : retourne la chaine `s` où les lettres majuscules et minuscules sont inversées.

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
```

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
>>> s = s.upper()
>>> s
'PYTHON EST UN LANGAGE SEMI-INTERPRÉTÉ'
```

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
>>> s = s.upper()
>>> s
'PYTHON EST UN LANGAGE SEMI-INTERPRÉTÉ'
>>> s = s.lower()
>>> s
'python est un langage semi-interprété'
```

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
>>> s = s.upper()
>>> s
'PYTHON EST UN LANGAGE SEMI-INTERPRÉTÉ'
>>> s = s.lower()
>>> s
'python est un langage semi-interprété'
>>> s = s.capitalize()
>>> s
'Python est un langage semi-interprété'
```

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
>>> s = s.upper()
>>> s
'PYTHON EST UN LANGAGE SEMI-INTERPRÉTÉ'
>>> s = s.lower()
>>> s
'python est un langage semi-interprété'
>>> s = s.capitalize()
>>> s
'Python est un langage semi-interprété'
>>> s = s.title()
>>> s
'Python Est Un Langage Semi-Interprété'
```

Quelques autres fonctions utiles : exemples

```
>>> s = "Python est un langage semi-interprété"
>>> s = s.upper()
>>> s
'PYTHON EST UN LANGAGE SEMI-INTERPRÉTÉ'
>>> s = s.lower()
>>> s
'python est un langage semi-interprété'
>>> s = s.capitalize()
>>> s
'Python est un langage semi-interprété'
>>> s = s.title()
>>> s
'Python Est Un Langage Semi-Interprété'
>>> s = s.swapcase()
>>> s
'pYTHON eST uN lANGAGE sEMI-iNTERPRÉTÉ'
```

Les listes

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
```

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> print(couleurs)
['trèfle', 'pique', 'carreaux', 'coeur']
```

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> print(couleurs)
['trèfle', 'pique', 'carreaux', 'coeur']
>>> type(couleurs)
<class 'list'>
```

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> print(couleurs)
['trèfle', 'pique', 'carreaux', 'coeur']
>>> type(couleurs)
<class 'list'>
>>> liste1 = ['a', 4] #les éléments peuvent être hétérogènes
>>> liste2 = ['b', 5]
```

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> print(couleurs)
['trèfle', 'pique', 'carreaux', 'coeur']
>>> type(couleurs)
<class 'list'>
>>> liste1 = ['a', 4] #les éléments peuvent être hétérogènes
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2] #liste3 est une liste de listes
```

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> print(couleurs)
['trèfle', 'pique', 'carreaux', 'coeur']
>>> type(couleurs)
<class 'list'>
>>> liste1 = ['a', 4] #les éléments peuvent être hétérogènes
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2] #liste3 est une liste de listes
>>> print(liste3)
[['a', 4], ['b', 5]]
```

Evaluation d'une liste

Quelques exemples d'évaluation :

```
>>> [3+5, 5-2, 6%3, 8-5*2]
[8, 3, 0, -2]
```

Evaluation d'une liste

Quelques exemples d'évaluation :

```
>>> [3+5, 5-2, 6%3, 8-5*2]
[8, 3, 0, -2]
>>> ["am" + "stram" + "gram", 1 + 2 + 3]
['amstramgram', 6]
```

Evaluation d'une liste

Quelques exemples d'évaluation :

```
>>> [3+5, 5-2, 6%3, 8-5*2]
[8, 3, 0, -2]
>>> ["am" + "stram" + "gram", 1 + 2 + 3]
['amstramgram', 6]
>>> []
[]
```

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` $\rightarrow$ `int`

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` → `int`

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
```

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` → `int`

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> len(couleurs)
4
```

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` → `int`

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> len(couleurs)
4
>>> len(liste1)
2
```

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` → `int`

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> len(couleurs)
4
>>> len(liste1)
2
>>> len(liste3)
2
```

Longueur

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` → `int`

```
>>> couleurs = ["trèfle", "pique", "carreaux", "coeur"]
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> len(couleurs)
4
>>> len(liste1)
2
>>> len(liste3)
2
>>> len([])
0
```

Comparaison de listes

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$

Comparaison de listes

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$

```
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5]
True
```

Comparaison de listes

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$

```
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5]
True
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5, 6]
False
```

Comparaison de listes

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$

```
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5]
True
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5, 6]
False
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 5, 'b']
False
```

Comparaison de listes

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \to \texttt{bool}$$

```
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5]
True
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 'b', 5, 6]
False
>>> ['a', 2, 6, 'b', 5] == ['a', 2, 6, 5, 'b']
False
>>> [3+5, 5-2, 6%3, 8-5*2] == [8, 3, 0, -2]
True
```

Appartenance

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

Appartenance

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

```
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> 4 in liste1
True
```

Appartenance

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

```
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> 4 in liste1
True
>>> liste1 in liste3
True
```

Appartenance

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

```
>>> liste1 = ['a', 4]
>>> liste2 = ['b', 5]
>>> liste3 = [liste1, liste2]
>>> 4 in liste1
True
>>> liste1 in liste3
True
>>> 4 in liste3
False
```

Concaténation

Concaténation de listes

L'opérateur + permet de concaténer les listes.

Signature :

$$+ :: \texttt{list} \times \texttt{list} \rightarrow \texttt{list}$$

Concaténation

Concaténation de listes

L'opérateur + permet de concaténer les listes.

Signature :

$$+ :: \texttt{list} \times \texttt{list} \to \texttt{list}$$

```
>>> ['a', 5] + ['b', 6]
['a', 5, 'b', 6]
```

Concaténation

Concaténation de listes

L'opérateur + permet de concaténer les listes.

Signature :

$$+ :: \texttt{list} \times \texttt{list} \rightarrow \texttt{list}$$

```
>>> ['a', 5] + ['b', 6]
['a', 5, 'b', 6]
>>> ['b', 6] + ['a', 5]
['b', 6, 'a', 5]
```

Concaténation

Concaténation de listes

L'opérateur + permet de concaténer les listes.

Signature :

$$+ :: \texttt{list} \times \texttt{list} \rightarrow \texttt{list}$$

```
>>> ['a', 5] + ['b', 6]
['a', 5, 'b', 6]
>>> ['b', 6] + ['a', 5]
['b', 6, 'a', 5]
#La concaténation n'est pas commutative
>>> ['b', 6] + ['a', 5] == ['a', 5] + ['b', 6]
False
```

Concaténation

Concaténation de listes

L'opérateur + permet de concaténer les listes.

Signature :

$$+ :: \texttt{list} \times \texttt{list} \rightarrow \texttt{list}$$

```
>>> ['a', 5] + ['b', 6]
['a', 5, 'b', 6]
>>> ['b', 6] + ['a', 5]
['b', 6, 'a', 5]
#La concaténation n'est pas commutative
>>> ['b', 6] + ['a', 5] == ['a', 5] + ['b', 6]
False
#La liste vide est l'élément neutre
>>> [] + [1, 2, 3, 4]
[1, 2, 3, 4]
>>> [1, 2, 3, 4] + []
[1, 2, 3, 4]
```

Mutabilité

Les types que nous avons traités jusque ici (`int`, `bool`, `str`) sont immutables :

- on ne peut pas les modifier
- on peut remplacer la valeur d'une variable (`a = a + b`)
- Pour supprimer un élément d'une chaîne de caractères, on reconstruit la chaîne sans cet élément

Mutabilité

Un objet mutable peut être modifié : remplacement, suppression ou ajout d'une partie de l'objet.

Les **listes** sont mutables.

Ajout en fin de liste

Méthode `append`

La méthode `append` ajoute un élément à la fin de la liste depuis laquelle elle est appelée.

Ajout en fin de liste

Méthode append

La méthode append ajoute un élément à la fin de la liste depuis laquelle elle est appelée.

Syntaxe :

```
<liste>.append(<element>)
```

- `<liste>.append(<element>)` modifie la liste `<liste>` en lui ajoutant `<element>` à la fin
- La méthode append ne s'applique pas aux autres types de séquence
- Cette méthode ne **retourne rien** et modifie directement la liste

Ajout en fin de liste

```
>>> liste1 = [1, 2, 3, 4, 5]
```

Ajout en fin de liste

```
>>> liste1 = [1, 2, 3, 4, 5]
>>> liste1.append(6)
#Cet appel ne retourne rien, et donc n'affiche rien
```

Ajout en fin de liste

```
>>> liste1 = [1, 2, 3, 4, 5]
>>> liste1.append(6)
#Cet appel ne retourne rien, et donc n'affiche rien
>>> liste1
[1, 2, 3, 4, 5, 6] #liste1 a été modifiée
```

Ajout en fin de liste

```
>>> liste1 = [1, 2, 3, 4, 5]
>>> liste1.append(6)
#Cet appel ne retourne rien, et donc n'affiche rien
>>> liste1
[1, 2, 3, 4, 5, 6] #liste1 a été modifiée
>>> liste2 = ['a', 'b', 'c']
>>> liste2 + ['d'] #Opération : le résultat est affiché
['a', 'b', 'c', 'd']
```

Ajout en fin de liste

```
>>> liste1 = [1, 2, 3, 4, 5]
>>> liste1.append(6)
#Cet appel ne retourne rien, et donc n'affiche rien
>>> liste1
[1, 2, 3, 4, 5, 6] #liste1 a été modifiée
>>> liste2 = ['a', 'b', 'c']
>>> liste2 + ['d'] #Opération : le résultat est affiché
['a', 'b', 'c', 'd']
>>> liste2
['a', 'b', 'c'] #liste2 n'a pas été modifiée
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
>>> liste2
[1, 2, 3, 4, 5, 6, 7] #liste2 a été modifiée
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
>>> liste2
[1, 2, 3, 4, 5, 6, 7] #liste2 a été modifiée
>>> liste2.append(8)
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
>>> liste2
[1, 2, 3, 4, 5, 6, 7] #liste2 a été modifiée
>>> liste2.append(8)
>>> liste2
[1, 2, 3, 4, 5, 6, 7, 8]
>>> liste1
[1, 2, 3, 4, 5, 6, 7, 8] #liste1 a été modifiée
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
>>> liste2
[1, 2, 3, 4, 5, 6, 7] #liste2 a été modifiée
>>> liste2.append(8)
>>> liste2
[1, 2, 3, 4, 5, 6, 7, 8]
>>> liste1
[1, 2, 3, 4, 5, 6, 7, 8] #liste1 a été modifiée
>>> liste2 = [1] #liste2 est associée à un nouvel objet
```

Mutabilité

```
#Les listes sont mutables, l'objet lui même est modifié
>>> liste1 = [1, 2, 3, 4, 5, 6]
>>> liste2 = liste1
>>> liste2
[1, 2, 3, 4, 5, 6]
>>> liste1.append(7)
>>> liste1
[1, 2, 3, 4, 5, 6, 7]
>>> liste2
[1, 2, 3, 4, 5, 6, 7] #liste2 a été modifiée
>>> liste2.append(8)
>>> liste2
[1, 2, 3, 4, 5, 6, 7, 8]
>>> liste1
[1, 2, 3, 4, 5, 6, 7, 8] #liste1 a été modifiée
>>> liste2 = [1] #liste2 est associée à un nouvel objet
>>> liste2
[1]
>>> liste1
[1, 2, 3, 4, 5, 6, 7, 8]
```

Exemple : construction de liste

Problème : construire une liste contenant les n premiers nombres entiers impairs

Exemple : construction de liste

Problème : construire une liste contenant les n premiers nombres entiers impairs

```
n = int(input("Combien de nombres impairs voulez vous? "))

liste = [] #liste résultat

for var in range(0, n):
    liste.append(2 * var + 1)
print(liste)
```

Exemple : construction de liste

Problème : construire une liste contenant les n premiers nombres entiers impairs

```
n = int(input("Combien de nombres impairs voulez vous? "))

liste = [] #liste résultat

for var in range(0, n):
    liste.append(2 * var + 1)
print(liste)
```

```
Combien de nombres impairs voulez vous? 0
[]

Combien de nombres impairs voulez vous? 3
[1, 3, 5]

Combien de nombres impairs voulez vous? 15
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Simulation de boucle : construction de liste

```
n = int(input("Combien de nombres impairs voulez vous? "))

liste = [] #liste résultat

for var in range(0, n):
    liste.append(2 * var + 1)
print(liste)
```

Pour $n = 0$, `range(0, 0)` ne contient aucun élément :

tour de boucle	variable var	variable liste
entrée	-	[]

Simulation de boucle : construction de liste

```
n = int(input("Combien de nombres impairs voulez vous? "))

liste = [] #liste résultat

for var in range(0, n):
    liste.append(2 * var + 1)
print(liste)
```

Pour $n = 3$,

tour de boucle	variable var	variable liste
entrée	-	[]
tour 1	0	[1]
tour 2	1	[1, 3]
tour 3	2	[1, 3, 5]

Indexation des listes

Indexation simple

L'indice d'un élément dans une liste est sa position dans la liste.
Les indices sont numérotés à partir de 0. Il est possible d'utiliser les indices inverses.

Par exemple, la liste `["le", "petit", "chat", "dort", "dans", "son", "couffin"]`

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

Indexation et découpage des listes

Indice d'un élément dans une liste

`liste[i]` est l'élément en position *i* dans la liste `liste`.

On accède à la sous-liste de la liste `liste` qui commence à l'indice `i` **inclus**, et finit à l'indice `j` **exclus** par `liste[i:j]`

Raccourcis :

- `liste[i:]` s'évalue en `liste[i: len(liste)]`
- `liste[:j]` s'évalue en `liste[0:j]`
- `liste[:]` s'évalue en `liste[0: len(liste)]`

Indexation et découpage des listes

Elément	”le”	”petit”	”chat”	”dort”	”dans”	”son”	”couffin”
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
>>> liste[5:6] #Retourne une liste
['son']
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
>>> liste[5:6] #Retourne une liste
['son']
>>> liste[-6:-1]
['petit', 'chat', 'dort', 'dans', 'son']
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
>>> liste[5:6] #Retourne une liste
['son']
>>> liste[-6:-1]
['petit', 'chat', 'dort', 'dans', 'son']
>>> liste[:3]
['le', 'petit', 'chat']
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
>>> liste[5:6] #Retourne une liste
['son']
>>> liste[-6:-1]
['petit', 'chat', 'dort', 'dans', 'son']
>>> liste[:3]
['le', 'petit', 'chat']
>>> liste[3:]
['dort', 'dans', 'son', 'couffin']
```

Indexation et découpage des listes

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[5] #Retourne un élément
'son'
>>> liste[-5]
'chat'
>>> liste[5:6] #Retourne une liste
['son']
>>> liste[-6:-1]
['petit', 'chat', 'dort', 'dans', 'son']
>>> liste[:3]
['le', 'petit', 'chat']
>>> liste[3:]
['dort', 'dans', 'son', 'couffin']
>>> liste[:]
['le', 'petit', 'chat', 'dort', 'dans', 'son', 'couffin']
```

Découpage des listes avec pas

Découpage avec pas positif

`liste[i:j:k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[4:11:2]` renvoie la liste `[liste[4], liste[6], liste[8], liste[10]]`
- `liste[i:j:1]` correspond à `liste[i, j]`

Découpage des listes avec pas

Découpage avec pas positif

`liste[i:j:k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[4:11:2]` renvoie la liste `[liste[4], liste[6], liste[8], liste[10]]`
- `liste[i:j:1]` correspond à `liste[i, j]`

Découpage avec pas négatif

`liste[i:j:-k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[11:4:-2]` renvoie la liste `[liste[11], liste[9], liste[7], liste[5]]`

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
```

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[2:6:2]
['chat', 'dans']
```

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[2:6:2]
['chat', 'dans']
>>> liste[2:7:2]
['chat', 'dans', 'couffin']
```

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[2:6:2]
['chat', 'dans']
>>> liste[2:7:2]
['chat', 'dans', 'couffin']
>>> liste[6:2:-2]
['couffin', 'dans']
```

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[2:6:2]
['chat', 'dans']
>>> liste[2:7:2]
['chat', 'dans', 'couffin']
>>> liste[6:2:-2]
['couffin', 'dans']
>>> liste[1::3]
['petit', 'dans']
```

Découpage des listes avec pas : exemples

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

```
>>> liste =  ["le","petit","chat","dort","dans","son","couffin"]
>>> liste[2:6:2]
['chat', 'dans']
>>> liste[2:7:2]
['chat', 'dans', 'couffin']
>>> liste[6:2:-2]
['couffin', 'dans']
>>> liste[1::3]
['petit', 'dans']
>>> liste[::-1]
['couffin', 'son', 'dans', 'dort', 'chat', 'petit', 'le']
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
>>> liste2 = [liste, [[7, 8], [9, 10]]]
>>> liste2
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10]]]
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
>>> liste2 = [liste, [[7, 8], [9, 10]]]
>>> liste2
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10]]]
>>> liste2[0]
[[1, 2], [3, 4], [5, 6]]
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
>>> liste2 = [liste, [[7, 8], [9, 10]]]
>>> liste2
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10]]]
>>> liste2[0]
[[1, 2], [3, 4], [5, 6]]
>>> liste2[0][1]
[3, 4]
```

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
>>> liste2 = [liste, [[7, 8], [9, 10]]]
>>> liste2
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10]]]
>>> liste2[0]
[[1, 2], [3, 4], [5, 6]]
>>> liste2[0][1]
[3, 4]
>>> liste2[0][1][0]
3
```

Les ensembles

Ensembles

Ensembles

Un ensemble, de type `set` est une collection non ordonnée d'éléments uniques.

Un ensemble est formé d'éléments séparés par des virgules, et entourés d'accolades.

L'ensemble vide, noté `set()`, est un ensemble qui ne contient aucun élément.

Ensembles

Ensembles

Un ensemble, de type `set` est une collection non ordonnée d'éléments uniques.

Un ensemble est formé d'éléments séparés par des virgules, et entourés d'accolades.

L'ensemble vide, noté `set()`, est un ensemble qui ne contient aucun élément.

Un `set` est une transposition informatique de la notion d'ensemble mathématiques.

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> lettres = set('abcdef')
>>> lettres
{'c', 'e', 'd', 'f', 'a', 'b'}
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> lettres = set('abcdef')
>>> lettres
{'c', 'e', 'd', 'f', 'a', 'b'}
>>> doublon = set('aaaabbbbcc') #pas de doublons
>>> doublon
{'c', 'a', 'b'}
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> lettres = set('abcdef')
>>> lettres
{'c', 'e', 'd', 'f', 'a', 'b'}
>>> doublon = set('aaaabbbbcc') #pas de doublons
>>> doublon
{'c', 'a', 'b'}
>>> {1, 2, 3} == {2, 1, 3} #pas ordonné
True
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> lettres = set('abcdef')
>>> lettres
{'c', 'e', 'd', 'f', 'a', 'b'}
>>> doublon = set('aaaabbbbcc') #pas de doublons
>>> doublon
{'c', 'a', 'b'}
>>> {1, 2, 3} == {2, 1, 3} #pas ordonné
True
>>> {1, 2, 3} == {2, 1, 3, 3}
True
```

Ensembles : exemples

```
#expression littérale
>>> couleurs = {'trefle', 'pique', 'carreau', 'coeur'}
>>> couleurs
{'trefle', 'carreau', 'coeur', 'pique'}
#construction à partir des éléments d'un itérable
>>> chiffres = set(range(10))
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> lettres = set('abcdef')
>>> lettres
{'c', 'e', 'd', 'f', 'a', 'b'}
>>> doublon = set('aaaabbbbcc') #pas de doublons
>>> doublon
{'c', 'a', 'b'}
>>> {1, 2, 3} == {2, 1, 3} #pas ordonné
True
>>> {1, 2, 3} == {2, 1, 3, 3}
True
>>> {1, 2, 3} == {2, 1, 3, 4}
False
```

Opérations ensemblistes

Soit E et F deux ensembles, et x un élément quelconque

Notation Python	Notation mathématique
`len(E)`	$\lvert E \rvert$: le cardinal de E
`set()`	$\emptyset$: l'ensemble vide
`x in E`	$x \in E$: l'appartenance
`x not in E`	$x \notin E$: la non-appartenance
`E < F`	$E \subset F$: l'inclusion stricte
`E <= F`	$E \subseteq F$: l'inclusion large
`E & F`	$E \cap F$: l'intersection
`E | F`	$E \cup F$: l'union
`E - F`	$E \setminus F$: la différence

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
>>> ens1 < ens2 #Inclusion stricte
False
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
>>> ens1 < ens2 #Inclusion stricte
False
>>> ens3 = ens1 & ens2 #Intersection
>>> ens3
{4}
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
>>> ens1 < ens2 #Inclusion stricte
False
>>> ens3 = ens1 & ens2 #Intersection
>>> ens3
{4}
>>> ens4 = ens1 | ens2 #Union
>>> ens4
{1, 2, 3, 4, 5, 6}
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
>>> ens1 < ens2 #Inclusion stricte
False
>>> ens3 = ens1 & ens2 #Intersection
>>> ens3
{4}
>>> ens4 = ens1 | ens2 #Union
>>> ens4
{1, 2, 3, 4, 5, 6}
>>> ens1 < ens4
True
```

Opérations ensemblistes : exemple

```
>>> ens1 = {1, 2, 3, 4}
>>> ens2 = {4, 5, 6}
>>> 2 in ens1 #Appartenance
True
>>> 2 not in ens2 #Non-appartenance
True
>>> ens1 < ens2 #Inclusion stricte
False
>>> ens3 = ens1 & ens2 #Intersection
>>> ens3
{4}
>>> ens4 = ens1 | ens2 #Union
>>> ens4
{1, 2, 3, 4, 5, 6}
>>> ens1 < ens4
True
>>> ens5 = ens1 - ens2 #Différence
>>> ens5
{1, 2, 3}
```

Mutabilité

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`

Mutabilité

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`
- La méthode `update` ajoute plusieurs éléments à `s` : `s.update({4, 5, 6})`

Mutabilité

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`
- La méthode `update` ajoute plusieurs éléments à `s` : `s.update({4, 5, 6})`
- Les méthodes `remove` et `discard` suppriment un élément de `s` : `s.remove(4)` ou `s.discard(4)`

Mutabilité

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`
- La méthode `update` ajoute plusieurs éléments à `s` : `s.update({4, 5, 6})`
- Les méthodes `remove` et `discard` suppriment un élément de `s` : `s.remove(4)` ou `s.discard(4)`
  - La différence est que si l'élément que l'on souhaite supprimer n'appartient pas à l'ensemble, `discard` ne modifiera pas l'ensemble, tandis que `remove` retournera une erreur

Mutabilité

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`
- La méthode `update` ajoute plusieurs éléments à `s` : `s.update({4, 5, 6})`
- Les méthodes `remove` et `discard` suppriment un élément de `s` : `s.remove(4)` ou `s.discard(4)`
  - La différence est que si l'élément que l'on souhaite supprimer n'appartient pas à l'ensemble, `discard` ne modifiera pas l'ensemble, tandis que `remove` retournera une erreur
- La méthode `clear` supprime tous les éléments de `s` : `s.clear()`

Ajout en fin de liste

```
>>> s = {1, 3}
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.remove(4)
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.remove(4)
>>> s.discard(1)
>>> s
{2, 3, 5, 6}
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.remove(4)
>>> s.discard(1)
>>> s
{2, 3, 5, 6}
>>> s.discard(4)
>>> s
{2, 3, 5, 6}
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.remove(4)
>>> s.discard(1)
>>> s
{2, 3, 5, 6}
>>> s.discard(4)
>>> s
{2, 3, 5, 6}
>>> s.remove(1)
File "<stdin>", line 1, in <module> KeyError: 1
```

Ajout en fin de liste

```
>>> s = {1, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.update({3, 4, 5, 6})
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.remove(4)
>>> s.discard(1)
>>> s
{2, 3, 5, 6}
>>> s.discard(4)
>>> s
{2, 3, 5, 6}
>>> s.remove(1)
File "<stdin>", line 1, in <module> KeyError: 1
>>> s.clear()
>>> s
set()
```

Pour conclure

Résumé du cours

Aujourd'hui, on a vu

- Les chaînes de caractères, et les opérations que l'on peut utiliser plus en détail
- Les listes
- La notion d'objet mutable ou immutable
- Les ensembles