Bases de l'informatique 1

UNIVERSITÉ CÔTE D'AZUR

Cours 8. Ensembles, dictionnaires et fichiers texte

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre

Ensembles Python

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>>
```

Ensembles Python

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
```

Ensembles Python

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>> type(set())
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~E[i]~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>> set()
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>> set()
set()
```

- La notation ~~`E[i]`~~ n'a pas de sens

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

```
>>>
```
SHELL

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>>
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL

```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL

```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL

```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

SHELL
```
>>>
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

SHELL
```
>>> for x in E:
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

SHELL
```
>>> for x in E:
...     print(x)
```

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL
```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

SHELL
```
>>> for x in E:
...     print(x)
True
Salut
22
31.2
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>>
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>>
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>>
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>>
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- Ou directement avec l'opérateur `<=` (si on veut l'inclusion stricte : `<`) :

SHELL
```
>>>
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- Ou directement avec l'opérateur `<=` (si on veut l'inclusion stricte : `<`) :

SHELL
```
>>> print( A<=B , B<=A , A<=A, A<A , set()<A)
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- Ou directement avec l'opérateur `<=` (si on veut l'inclusion stricte : `<`) :

SHELL
```
>>> print( A<=B , B<=A , A<=A, A<A , set()<A)
True False True False True
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>>
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>>
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>>
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'embête pour rien : fonction `len`

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'~~embête pour~~ rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'~~embête pour~~ rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>>
```

Nombre d'éléments d'un ensemble

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'~~embête pour~~ rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>> len({2,2,1,1,3,3,2,1,2,3})
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'embête pour rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>> len({2,2,1,1,3,3,2,1,2,3})
3
>>>
```

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'embête pour rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>> len({2,2,1,1,3,3,2,1,2,3})
3
>>> len(set())
```

Nombre d'éléments d'un ensemble

- En mathématiques, le cardinal est le nombre d'éléments d'un ensemble.

SCRIPT
```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d'habitude, on s'~~embête pour~~ rien : fonction `len`
  - C'est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>> len({2,2,1,1,3,3,2,1,2,3})
3
>>> len(set())
0
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL

```
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL

```
>>> E = {1 , 2 , 3 , 4}
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL

```
>>> E = {1 , 2 , 3 , 4}
>>> E
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL

```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

```shell
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

```shell
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>>
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>>
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'c', 'b', 'a'}
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'c', 'b', 'a'}
>>> set(['a','b','c'])
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'c', 'b', 'a'}
>>> set(['a','b','c'])
{'c', 'b', 'a'}
>>>
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'c', 'b', 'a'}
>>> set(['a','b','c'])
{'c', 'b', 'a'}
>>> set(('a','b','c'))
```

Construire un ensemble

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'c', 'b', 'a'}
>>> set(['a','b','c'])
{'c', 'b', 'a'}
>>> set(('a','b','c'))
{'c', 'b', 'a'}
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

```shell
>>>
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E              # E contient 3 éléments
{1, 2, 3}
>>>
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E              # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E              # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>>
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E              # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E              # E contient maintenant 4 éléments
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>>
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL

```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL

```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>>
```

- On peut supprimer un élément avec la méthode `remove`

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL

```
>>> E              # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E              # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E              # E contient toujours 4 éléments
```

- On peut supprimer un élément avec la méthode `remove`

Modifier un ensemble

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```
SHELL

- On peut supprimer un élément avec la méthode `remove`

```
>>>
```
SHELL

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

```shell
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

```shell
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
```

Modifier un ensemble

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```
SHELL

- On peut supprimer un élément avec la méthode `remove`

```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>>
```
SHELL

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'HA ha ha'
>>>
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'HA ha ha'
>>> E
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'HA ha ha'
>>> E
{False, 'Ha ha ha', 'Salut', (1, 2, 3)}
>>>
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E                # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E                # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E                # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'HA ha ha'
>>> E
{False, 'Ha ha ha', 'Salut', (1, 2, 3)}
>>> E.remove('Ha ha ha')
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL

```
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL

```
>>> A={1,2,3,4}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
{2, 4}
>>>
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
{2, 4}
>>> A-B
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
{2, 4}
>>> A-B
{1, 3}
```

Opérations sur les ensembles

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{ x : x \in A \text{ ou } x \in B \}$ se note `A | B`
  - L'intersection $A \cap B = \{ x : x \in A \text{ et } x \in B \}$ se note `A & B`
  - La différence $A \setminus B = \{ x : x \in A \text{ et } x \notin B \}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
{2, 4}
>>> A-B
{1, 3}
```

- Exercice : écrire l'intersection et la différence sans utiliser d'opérateurs.

- Bonne question : essayons !

SHELL

```
>>>
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }                # str   : immuable
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>>
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>>
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>>
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.
  - On peut faire des ensembles de tuple (ensembles de points du plan)

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.
  - On peut faire des ensembles de tuple (ensembles de points du plan)
  - On ne peut pas faire des ensembles de listes ou d'ensemble.

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.
  - On peut faire des ensembles de tuple (ensembles de points du plan)
  - On ne peut pas faire des ensembles de listes ou d'ensemble.

- Que signifie *unhashable type* du message d'erreur ?

- Bonne question : essayons !

SHELL

```
>>> { '123' , 'abc' }              # str   : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.
  - On peut faire des ensembles de tuple (ensembles de points du plan)
  - On ne peut pas faire des ensembles de listes ou d'ensemble.

- Que signifie *unhashable type* du message d'erreur ?
  - En interne, les ensembles utilisent des fonctions de hachage.

Sommaire

- Qu'est-ce qu'une empreinte digitale ?

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir
- Existe-il un équivalent numérique ?

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL
```
>>>
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>>
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>>
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>>
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.

- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
'49a5a960c5714c2e29dd1a7e7b950741'
>>>
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
'49a5a960c5714c2e29dd1a7e7b950741'
>>> h(min)
```

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
'49a5a960c5714c2e29dd1a7e7b950741'
>>> h(min)
'0f61485fd84d673c233e28e1d2a5acfe'
```

Pourquoi ?

- À quoi ces fonctions servent-elles ?

Pourquoi ?

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>>
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL

```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c42012567482404030e362f0c3813c15'
>>>
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL

```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c420125674824040 30e362f0c3813c15'
>>> h(B)
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c420125674824040 30e362f0c3813c15'
>>> h(B)
'e5a41e2edb893242fe25aad75dfcca66'
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c420125674824040 30e362f0c3813c15'
>>> h(B)
'e5a41e2edb893242fe25aad75dfcca66'
```

- Soit `s1` et `s2` deux chaînes et `h(s1)` et `h(s2)` leurs empreintes.

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c420125674824040 30e362f0c3813c15'
>>> h(B)
'e5a41e2edb893242fe25aad75dfcca66'
```

- Soit `s1` et `s2` deux chaînes et `h(s1)` et `h(s2)` leurs empreintes.
- On a la garantie suivante :
  - Si `h(s1)≠h(s2)` alors forcément `s1≠s2` (car `h` est une fonction)

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='12345678910111213141516171819202122232425262 7'
>>> B='12345678910111213141516171819202122232425262 7'
>>> h(A)
'c420125674824040 30e362f0c3813c15'
>>> h(B)
'e5a41e2edb893242fe25aad75dfcca66'
```

- Soit `s1` et `s2` deux chaînes et `h(s1)` et `h(s2)` leurs empreintes.
- On a la garantie suivante :
  - Si `h(s1)≠h(s2)` alors forcément `s1≠s2` (car `h` est une fonction)
- Les propriétés suivantes sont extrêmement probables :
  - Si `h(s1)=h(s2)` on peut en pratique considérer que `s1=s2` (on a 1 chance sur 340 282 366 920 938 463 463 374 607 431 768 211 455 de se tromper)
  - Si `s1` et `s2` sont très proches, `h(s1)` et `h(s2)` sont très différents.

Application aux ensembles

- Supposons que l'on souhaite implémenter les ensembles par des listes

Application aux ensembles

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT

```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.
  - S'il y a `n` éléments de taille `T`, il faudra faire `n*T` comparaisons.

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.
  - S'il y a `n` éléments de taille `T`, il faudra faire `n*T` comparaisons.

- Pour gagner en efficacité, on stocke chaque élément avec son empreinte.

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.
  - S'il y a `n` éléments de taille `T`, il faudra faire `n*T` comparaisons.

- Pour gagner en efficacité, on stocke chaque élément avec son empreinte.

SCRIPT
```
def ajouter(L,x):
    hx=h(x) # je calcule l'empreinte de x
    for e in L:
        (hy,y) = e # y est stocké avec son empreinte
        if hx == hy: return # le programme termine
    L.append((hx,x))
    return
```

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return # terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.
  - S'il y a `n` éléments de taille `T`, il faudra faire `n*T` comparaisons.

- Pour gagner en efficacité, on stocke chaque élément avec son empreinte.

SCRIPT
```
def ajouter(L,x):
    hx=h(x) # je calcule l'empreinte de x
    for e in L:
        (hy,y) = e # y est stocké avec son empreinte
        if hx == hy: return # le programme termine
    L.append((hx,x))
    return
```

- Dorénavant je dois faire seulement `N` comparaisons d'empreintes.

- Dans Python

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.
    - On stocke les empreintes

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.
    - On stocke les empreintes
    - On compare avec l'empreinte du mot de passe fourni par l'utilisateur.

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.
    - On stocke les empreintes
    - On compare avec l'empreinte du mot de passe fourni par l'utilisateur.
    - À aucun moment on ne stocke les mots de passe

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.
    - On stocke les empreintes
    - On compare avec l'empreinte du mot de passe fourni par l'utilisateur.
    - À aucun moment on ne stocke les mots de passe
    - Une empreinte ne permet pas de retrouver le mot de passe

- Dans Python
  - Les ensembles sont implémentés de manière bien plus élaborées.
  - Mais ils utilisent des tables de hachage.
  - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - On compare les empreintes (*MD5 check sum*)
    - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  - pour stocker des mots de passe sans les révéler.
    - On stocke les empreintes
    - On compare avec l'empreinte du mot de passe fourni par l'utilisateur.
    - À aucun moment on ne stocke les mots de passe
    - Une empreinte ne permet pas de retrouver le mot de passe
  - pour certifier la liste chaînée d'une *blockchain* (bitcoin) ou de git.

Sommaire

- Un dictionnaire est une collection de couples `clé:valeur`.

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>>
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>>
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL

```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>>
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>>
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire vide se note `{}` ou mieux `dict()`

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire vide se note `{}` ou mieux `dict()`
- Toutes les clés doivent être distinctes

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire vide se note `{}` ou mieux `dict()`

- Toutes les clés doivent être distinctes

SHELL
```
>>>
```

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```
SHELL

- On accède aux valeurs en utilisant les clés comme indices.

```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```
SHELL

- Un dictionnaire vide se note `{}` ou mieux `dict()`

- Toutes les clés doivent être distinctes

```
>>> {'pommes':243 , 'poires':123, 'pommes':23 }
```
SHELL

- Un dictionnaire est une collection de couples `clé:valeur`.
  - `clé` est forcément non mutable ;
  - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire vide se note `{}` ou mieux `dict()`

- Toutes les clés doivent être distinctes

SHELL
```
>>> {'pommes':243 , 'poires':123, 'pommes':23 }
{'pommes': 23, 'poires': 123}
```

- L'accès à une valeur est extrêmement rapide.

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL

```
>>>
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL
```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL
```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>>
```

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL

```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
```

Accès aux éléments

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL

```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>>
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL
```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL
```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
234
>>>
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL
```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
234
>>> dico['Bonjour'] # 'Bonjour' est une valeur !
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL

```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
234
>>> dico['Bonjour'] # 'Bonjour' est une valeur !
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Bonjour'
```

- Si on utilise une clé qui n'existe pas, une exception est levée.

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

```
>>>
```
SHELL

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL

```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>>
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL

```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL

```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>>
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL

```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Ou plus simplement en une ligne avec la méthode `get`

SHELL
```
>>>
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Ou plus simplement en une ligne avec la méthode `get`

SHELL
```
>>> partiel.get("Alice",'ABS')
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Ou plus simplement en une ligne avec la méthode `get`

SHELL
```
>>> partiel.get("Alice",'ABS')
15
>>>
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Ou plus simplement en une ligne avec la méthode `get`

SHELL
```
>>> partiel.get("Alice",'ABS')
15
>>> partiel.get("Gustave",'ABS')
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError:  'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

- On veut affecter à `note` la valeur associée à l'étudiant `étu`
  - Si une telle clé n'existe pas, on pose `note="ABS"`

SCRIPT
```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT
```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

- Ou plus simplement en une ligne avec la méthode `get`

SHELL
```
>>> partiel.get("Alice",'ABS')
15
>>> partiel.get("Gustave",'ABS')
'ABS'
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

```
>>>
```
SHELL

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL

```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL

```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>>
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL

```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>> dico.pop(33)
```

Modifier un dictionnaire

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>> dico.pop(33)
'tri'
>>>
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>> dico.pop(33)
'tri'
>>> dico
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>> dico.pop(33)
'tri'
>>> dico
{11: 'unu', 22: 'du', 44: 'kvar'}
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

```shell
>>> mousquetaires=dict()
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SHELL
```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
>>> mousquetaires["d'Artagnan"]=1615
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SCRIPT

```
def parcours(dico):
    for k in dico:    # k est la clé
        v = dico[k]   # v est la valeur
        print(f"{k} ({v})")
```

SHELL

```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
>>> mousquetaires["d'Artagnan"]=1615
>>>
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SCRIPT

```
def parcours(dico):
    for k in dico:    # k est la clé
        v = dico[k]   # v est la valeur
        print(f"{k} ({v})")
```

SHELL

```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
>>> mousquetaires["d'Artagnan"]=1615
>>> parcours(mousquetaires)
```

- On peut parcourir un dictionnaire en itérant sur les clés.

SCRIPT

```
def parcours(dico):
    for k in dico:    # k est la clé
        v = dico[k]   # v est la valeur
        print(f"{k} ({v})")
```

SHELL

```
>>> mousquetaires=dict()
>>> mousquetaires["Athos"]=1615
>>> mousquetaires["Porthos"]=1617
>>> mousquetaires["Aramis"]=1620
>>> mousquetaires["d'Artagnan"]=1615
>>> parcours(mousquetaires)
Athos (1615)
Porthos (1617)
Aramis (1620)
d'Artagnan (1615)
```

Sommaire

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.
- Principe de l'algorithme

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.
- Principe de l'algorithme
  - Si `n` est une clé du dictionnaire, on renvoie la valeur associée.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.
- Principe de l'algorithme
  - Si `n` est une clé du dictionnaire, on renvoie la valeur associée.
  - Sinon, on calcule `v=n*fact(n-1)` et on ajoute `n:v` dans le dictionnaire.

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.
- Principe de l'algorithme
  - Si `n` est une clé du dictionnaire, on renvoie la valeur associée.
  - Sinon, on calcule `v=n*fact(n-1)` et on ajoute `n:v` dans le dictionnaire.
- Cette méthode s'appelle la mémoïsation.

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>> print(t810/t800)
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>> print(t810/t800)
0.009458297506448839
>>>
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>> print(t810/t800)
0.009458297506448839
>>> print(f'{t810/t800:.1%}')
```

Implémentation de la factorielle

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>> print(t810/t800)
0.009458297506448839
>>> print(f'{t810/t800:.1%}')
0.9%
```

- Le calcul de `fact(810)` est 100 fois plus efficace que celui de `fact(800)`

- On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.

SCRIPT

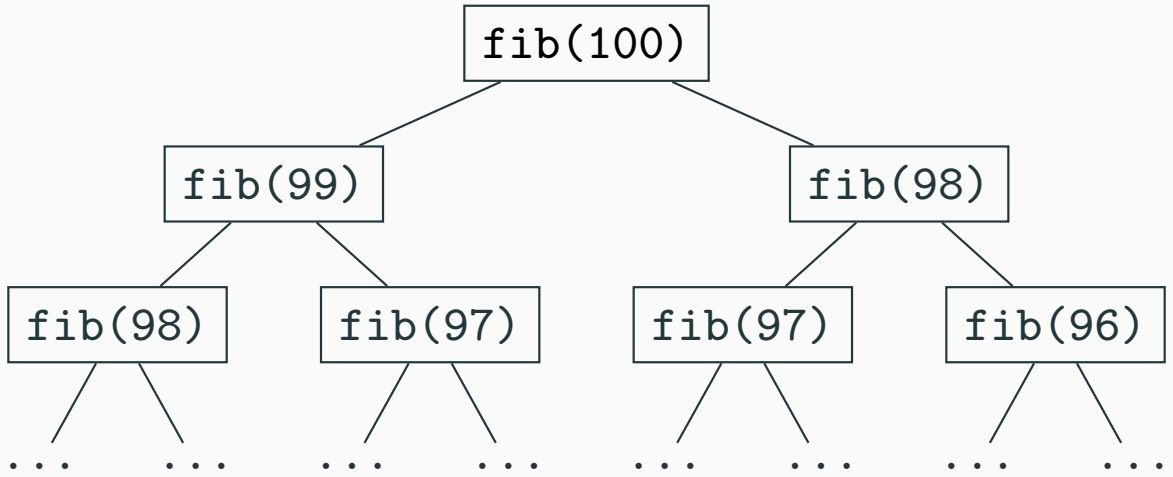
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.
  - Très peu efficaces

SCRIPT
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.
  - Très peu efficaces
  - Les mêmes calculs sont faits de nombreuses fois.

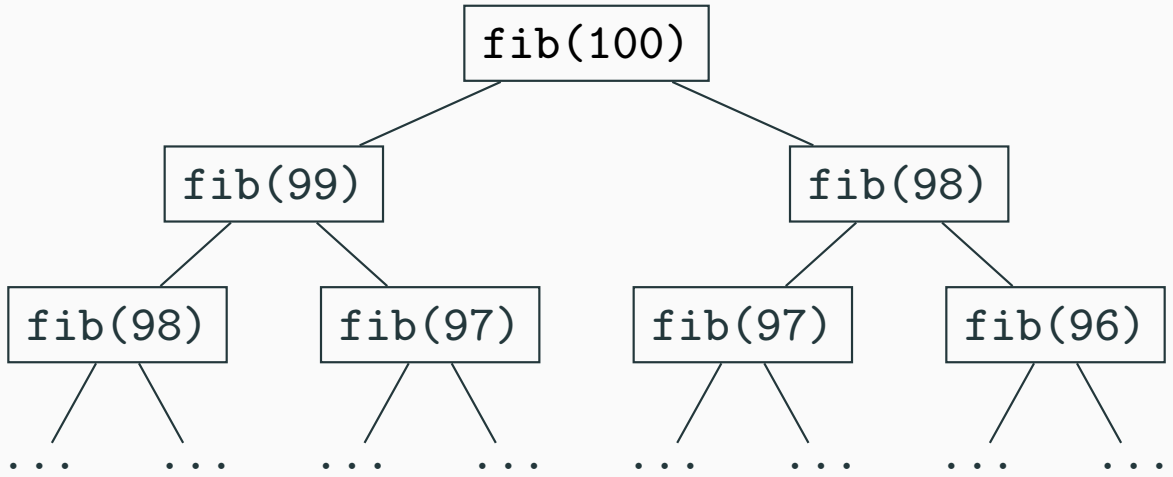
SCRIPT
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



- On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.
  - Très peu efficaces
  - Les mêmes calculs sont faits de nombreuses fois.

SCRIPT
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

On souhaite retenir les résultats intermédiaires (mémoïsation)



- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases}$$

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1 + T_n) = (1 + T_{n-1}) + (1 + T_{n-2}) \\ (1 + T_0) = (1 + T_1) = 2 \end{cases}$$

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1 + T_n) = (1 + T_{n-1}) + (1 + T_{n-2}) \\ (1 + T_0) = (1 + T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2 \cdot$ `fib(100)` $- 1$ appels récursifs

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1 + T_n) = (1 + T_{n-1}) + (1 + T_{n-2}) \\ (1 + T_0) = (1 + T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2 \cdot$ `fib(100)` $- 1$ appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot$ `fib(n)` car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1 + T_n) = (1 + T_{n-1}) + (1 + T_{n-2}) \\ (1 + T_0) = (1 + T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2 \cdot$ `fib(100)` $- 1$ appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$ s ≈ 36 années.

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1+T_n = 2\cdot$`fib(n)` car $1+T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1+T_{n-1}+T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1+T_n) = (1+T_{n-1})+(1+T_{n-2}) \\ (1+T_0) = (1+T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2\cdot$`fib(100)`-1 appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$s ≈ 36 années.

SCRIPT

```
mem = dict()
def fib(n):
    if n==0 or n==1:
        mem[n] = 1
    elif n not in mem:
        mem[n] = fib(n-1)+fib(n-2)
    return mem[n]
```

SHELL

```
>>>
```

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1+T_n = 2\cdot$`fib(n)` car $1+T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1+T_{n-1}+T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1+T_n) = (1+T_{n-1})+(1+T_{n-2}) \\ (1+T_0) = (1+T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2\cdot$`fib(100)`-1 appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$s ≈ 36 années.

SCRIPT

```
mem = dict()
def fib(n):
    if n==0 or n==1:
        mem[n] = 1
    elif n not in mem:
        mem[n] = fib(n-1)+fib(n-2)
    return mem[n]
```

SHELL

```
>>> fib(100) #Quasi instantané
```

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1+T_n = 2\cdot$`fib(n)` car $1+T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1+T_{n-1}+T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1+T_n) = (1+T_{n-1})+(1+T_{n-2}) \\ (1+T_0) = (1+T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2\cdot$`fib(100)`-1 appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$s ≈ 36 années.

SCRIPT

```
mem = dict()
def fib(n):
    if n==0 or n==1:
        mem[n] = 1
    elif n not in mem:
        mem[n] = fib(n-1)+fib(n-2)
    return mem[n]
```

SHELL

```
>>> fib(100) #Quasi instantané
573147844013817084101
```

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1+T_n = 2\cdot$`fib(n)` car $1+T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1+T_{n-1}+T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1+T_n) = (1+T_{n-1})+(1+T_{n-2}) \\ (1+T_0) = (1+T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2\cdot$`fib(100)`-1 appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$s ≈ 36 années.

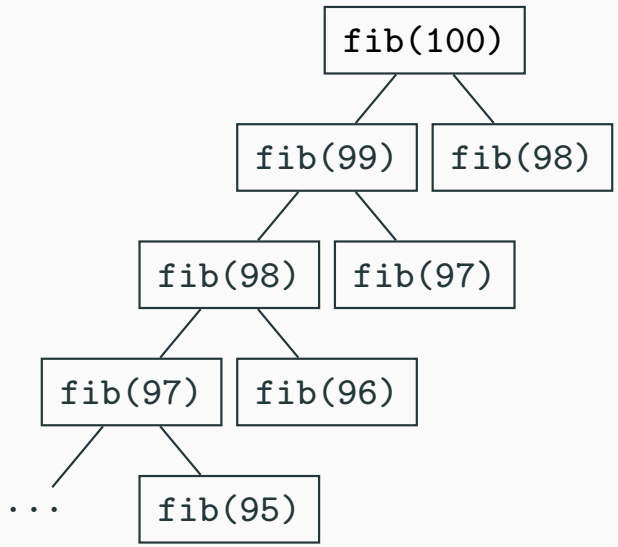
SCRIPT

```
mem = dict()
def fib(n):
    if n==0 or n==1:
        mem[n] = 1
    elif n not in mem:
        mem[n] = fib(n-1)+fib(n-2)
    return mem[n]
```

SHELL

```
>>> fib(100) #Quasi instantané
573147844013817084101
```

Le nouvel arbre d'appels est un « peigne » qui compte seulement 201 appels récursifs.



Sommaire

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

SHELL
```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
```

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

SHELL
```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

SHELL
```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```

- Version plus lisible et plus aisément modifiable sur le long terme.

SHELL
```
>>> f'{n} = {n//d} × {d} + {n%d}'
```

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```
SHELL

- Version plus lisible et plus aisément modifiable sur le long terme.

```
>>> f'{n} = {n//d} × {d} + {n%d}'
'17 = 3 × 5 + 2'
```
SHELL

- On veut afficher le résultat d'une division euclidienne avec `(n,d)=(17,5)`

$$17 = 3 \times 5 + 2$$

- Version lourde : dur à modifier et à lire.

```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```
SHELL

- Version plus lisible et plus aisément modifiable sur le long terme.

```
>>> f'{n} = {n//d} × {d} + {n%d}'
'17 = 3 × 5 + 2'
```
SHELL

L'ajout d'un `f` devant la chaîne permet d'utiliser des variables ou des expressions entre accolades. On parle de `f-string`.

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>>
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>>
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

- On peut même utiliser des pourcentages :

SHELL
```
>>>
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

- On peut même utiliser des pourcentages :

SHELL
```
>>> f'{0.1676:.2%}    {1/2:.0%}    {4/3:.4%}'
```

- On peut choisir le nombre de chiffres après la virgule.

SHELL
```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3    3.14    3.1416'
```

- On peut préférer l'écriture scientifique ($12345 = 1,2345 \times 10^4$) :

SHELL
```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

- On peut même utiliser des pourcentages :

SHELL
```
>>> f'{0.1676:.2%}    {1/2:.0%}    {4/3:.4%}'
'16.76%    50%    133.3333%'
```

- Le formatage permet aussi d'aligner des chaînes de tailles différentes.

SCRIPT

```
texte = "Où suis-je ?"
print('#'*50) # Chaîne de 50 caractères
print(f"{texte:<50} fin") # aligné à gauche sur 50 caractères
print(f"{texte:>50} fin") # aligné à droite sur 50 caractères
print(f"{texte:^50} fin") # centré sur 50 caractères
print("")
print(f"{texte:*<50} fin")
print(f"{texte:*>50} fin") # On peut même préciser
print(f"{texte:*^50} fin") # les caractères à ajouter
```

SHELL

```
##################################################
Où suis-je ?                                    fin
                                     Où suis-je ? fin
                    Où suis-je ?                 fin

Où suis-je ?********************************** fin
**********************************Où suis-je ? fin
********************Où suis-je ?******************** fin
```

- Le formatage permet aussi d'aligner des chaînes de tailles différentes.

SCRIPT

```
texte = "Où suis-je ?"
print('#'*50) # Chaîne de 50 caractères
print(f"{texte:<50} fin") # aligné à gauche sur 50 caractères
print(f"{texte:>50} fin") # aligné à droite sur 50 caractères
print(f"{texte:^50} fin") # centré sur 50 caractères
print("")
print(f"{texte:*<50} fin")
print(f"{texte:*>50} fin") # On peut même préciser
print(f"{texte:*^50} fin") # les caractères à ajouter
```

SHELL

```
##################################################
Où suis-je ?                                    fin
                                     Où suis-je ? fin
                    Où suis-je ?                 fin

Où suis-je ?********************************** fin
**********************************Où suis-je ? fin
********************Où suis-je ?******************** fin
```

- Application : écrire sur 3 chiffres (à faire chez vous) :

SHELL

```
1-11-21-31-41-51-61-71-81-91-101-avant
001-011-021-031-041-051-061-071-081-091-101-après
```

- On peut stocker des informations dans une chaîne de caractères

Découper une ligne de texte

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.
- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

```
>>>
```
SHELL

Découper une ligne de texte

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.
- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL

```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.
- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>>
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.
- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

Découper une ligne de texte

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.
- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>>
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>> '-'.join(['partiel','21','03','2024'])
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>> '-'.join(['partiel','21','03','2024'])
'partiel-21-03-2024'
>>>
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>> '-'.join(['partiel','21','03','2024'])
'partiel-21-03-2024'
>>> ''.join(['partiel','21','03','2024'])
```

- On peut stocker des informations dans une chaîne de caractères
  - avec les divers éléments séparés par une virgule (ou un espace).
  - On parle de format csv : utilisé par les tableurs
  - permet de stocker une feuille de calcul au format texte.

- On peut découper une chaîne pour récupérer la liste des éléments
  - méthode `split`

SHELL
```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15  8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

- Inversement, on peut recoller les éléments d'une liste de chaînes
  - méthode `sep.join(L)` (éléments de `L` séparés par `sep`)

SHELL
```
>>> '-'.join(['partiel','21','03','2024'])
'partiel-21-03-2024'
>>> ''.join(['partiel','21','03','2024'])
'partiel21032024'
```

Sommaire

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

Utilité des fichiers

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

- Le fichier désignait le recueil des fiches (ou le meuble les contenant).

Utilité des fichiers

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

- Le fichier désignait le recueil des fiches (ou le meuble les contenant).
- Un fichier informatique est un ensemble structuré d'information
  - tableau : feuille de calcul
  - suite d'instructions : programme
  - texte (txt, html, odt).
  - données quelconques (image, vidéo, etc.)

Utilité des fichiers

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

- Le fichier désignait le recueil des fiches (ou le meuble les contenant).
- Un fichier informatique est un ensemble structuré d'information
  - tableau : feuille de calcul
  - suite d'instructions : programme
  - texte (txt, html, odt).
  - données quelconques (image, vidéo, etc.)
- Les systèmes d'exploitation permettent de travailler sur des fichiers
  - stockage de grandes quantités d'informations
  - recherche, classement, modification
  - fichiers stockés sur le disque dur (ou clé USB, etc)

Utilité des fichiers

- Le mot fichier provient de fiche :

  feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.

- Le fichier désignait le recueil des fiches (ou le meuble les contenant).
- Un fichier informatique est un ensemble structuré d'information
  - tableau : feuille de calcul
  - suite d'instructions : programme
  - texte (txt, html, odt).
  - données quelconques (image, vidéo, etc.)
- Les systèmes d'exploitation permettent de travailler sur des fichiers
  - stockage de grandes quantités d'informations
  - recherche, classement, modification
  - fichiers stockés sur le disque dur (ou clé USB, etc)
- Système d'exploitation : *Operating System* (OS).
  - exemples : GNU/Linux, Android, Windows, macOS, etc

- Le père de tous les systèmes d'exploitation moderne est UNIX ($\approx$ 1975).

- Le père de tous les systèmes d'exploitation moderne est UNIX ($\approx$ 1975).
- Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.

- Le père de tous les systèmes d'exploitation moderne est UNIX (≈ 1975).
- Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.
  - Pour les serveurs :
    - La famille BSD : FreeBSD, NetBSD, OpenBSD (sécurité), dragonflyBSD
    - La famille propriétaire : AIX (IBM), Solaris (Oracle), HP-UX (HP),...
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...

- Le père de tous les systèmes d'exploitation moderne est UNIX (≈ 1975).
- Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.
  - Pour les serveurs :
    - La famille BSD : FreeBSD, NetBSD, OpenBSD (sécurité), dragonflyBSD
    - La famille propriétaire : AIX (IBM), Solaris (Oracle), HP-UX (HP),...
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
  - Pour les particuliers :
    - macOS (anciennement Mac OS X)
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
    - Les autres Linux (android)

- Le père de tous les systèmes d'exploitation moderne est UNIX (≈ 1975).
- Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.
  - Pour les serveurs :
    - La famille BSD : FreeBSD, NetBSD, OpenBSD (sécurité), dragonflyBSD
    - La famille propriétaire : AIX (IBM), Solaris (Oracle), HP-UX (HP),...
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
  - Pour les particuliers :
    - macOS (anciennement Mac OS X)
    - La famille GNU/Linux : Debian, Ubuntu, Red Hat, Gentoo,...
    - Les autres Linux (android)
- L'autre grande famille est composée des Windows...

- Les fichiers sont regroupés dans une arborescence,
  - avec une ou plusieurs racines
  - les nœuds sont les répertoires (ou dossiers)
  - les feuilles sont les fichiers
  - un répertoire peut contenir des sous-répertoires et des fichiers.

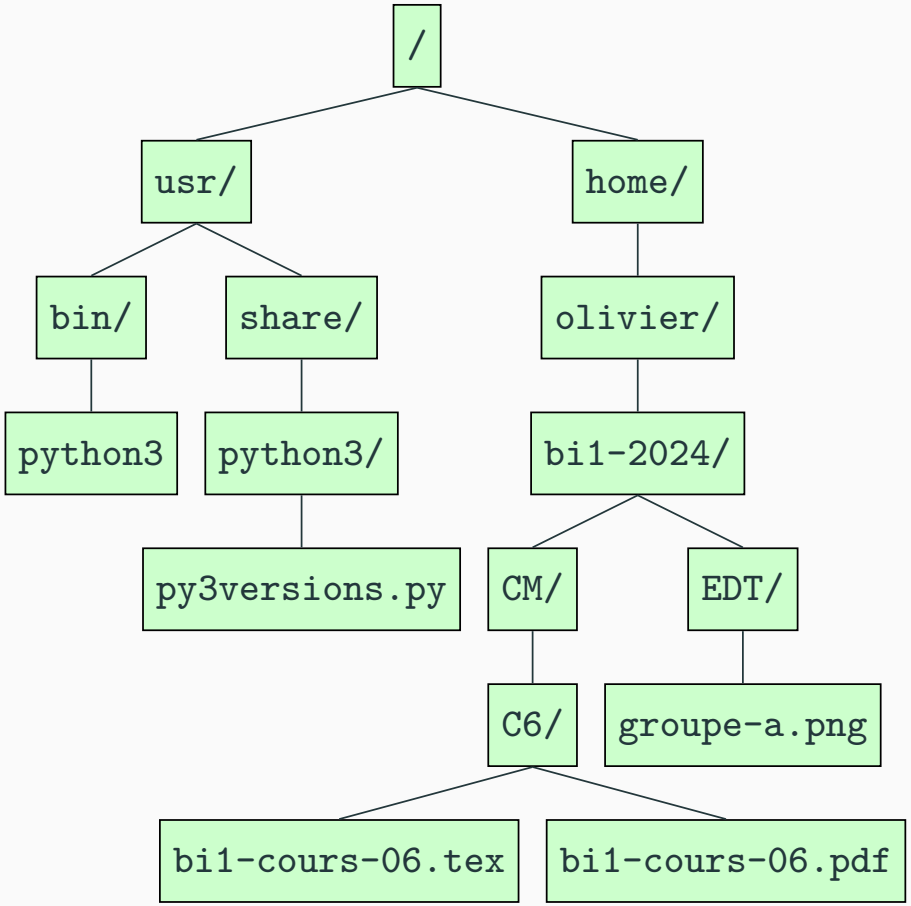
- Les fichiers sont regroupés dans une arborescence,
  - avec une ou plusieurs racines
  - les nœuds sont les répertoires (ou dossiers)
  - les feuilles sont les fichiers
  - un répertoire peut contenir des sous-répertoires et des fichiers.
- Il y a essentiellement deux systèmes d'arborescence :
  - Unix (GNU/Linux, macOS, BSD) avec des petites variantes
  - Windows

- Les fichiers sont regroupés dans une arborescence,
  - avec une ou plusieurs racines
  - les nœuds sont les répertoires (ou dossiers)
  - les feuilles sont les fichiers
  - un répertoire peut contenir des sous-répertoires et des fichiers.
- Il y a essentiellement deux systèmes d'arborescence :
  - Unix (GNU/Linux, macOS, BSD) avec des petites variantes
  - Windows
- Sur Unix, une seule racine (nommée /).

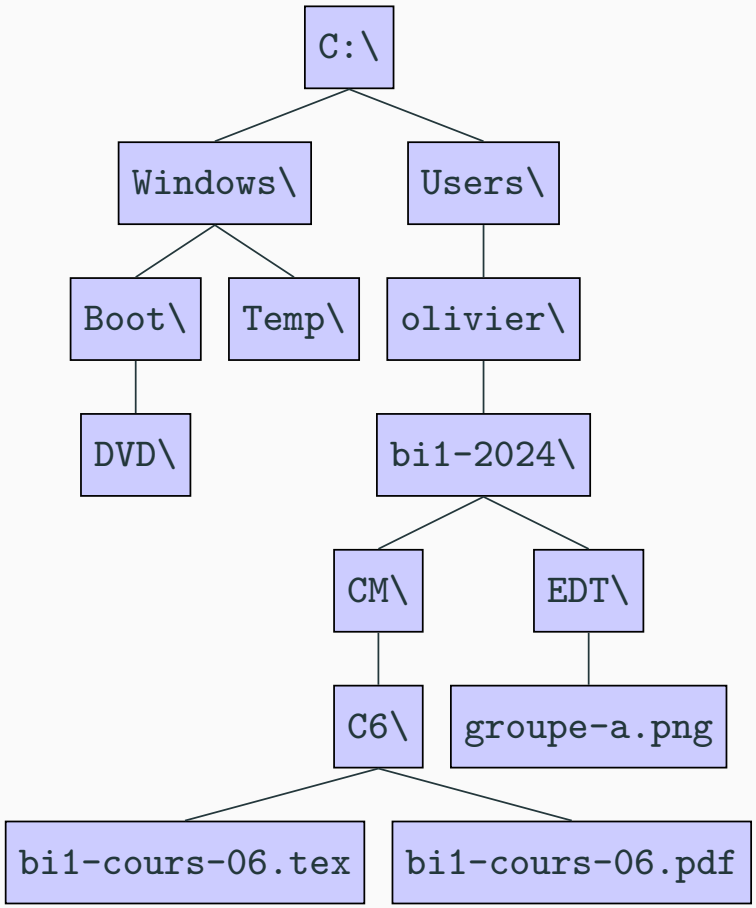
- Les fichiers sont regroupés dans une arborescence,
  - avec une ou plusieurs racines
  - les nœuds sont les répertoires (ou dossiers)
  - les feuilles sont les fichiers
  - un répertoire peut contenir des sous-répertoires et des fichiers.
- Il y a essentiellement deux systèmes d'arborescence :
  - Unix (GNU/Linux, macOS, BSD) avec des petites variantes
  - Windows
- Sur Unix, une seule racine (nommée `/`).
- Sur Windows, plusieurs racines (nommées `C:\`, `D:\`, etc).
  - En général, `C:\` représente le disque principal.

- Un répertoire est relié à la racine par un chemin

Arborescence Unix

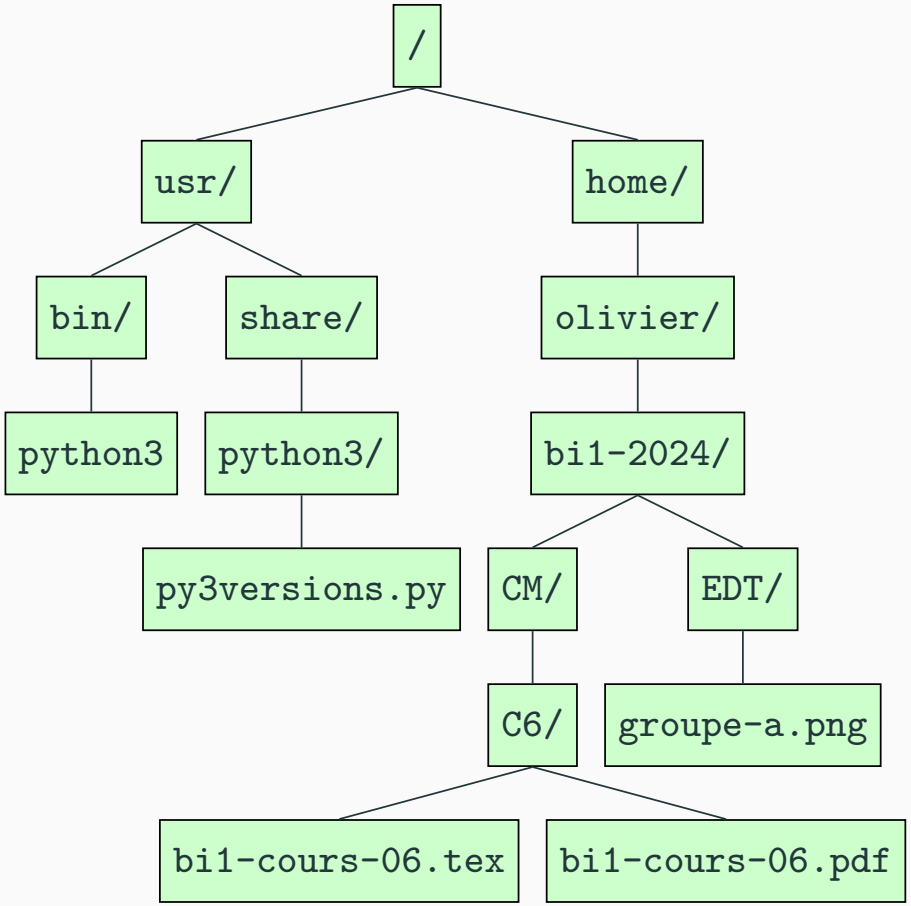


Arborescence Windows

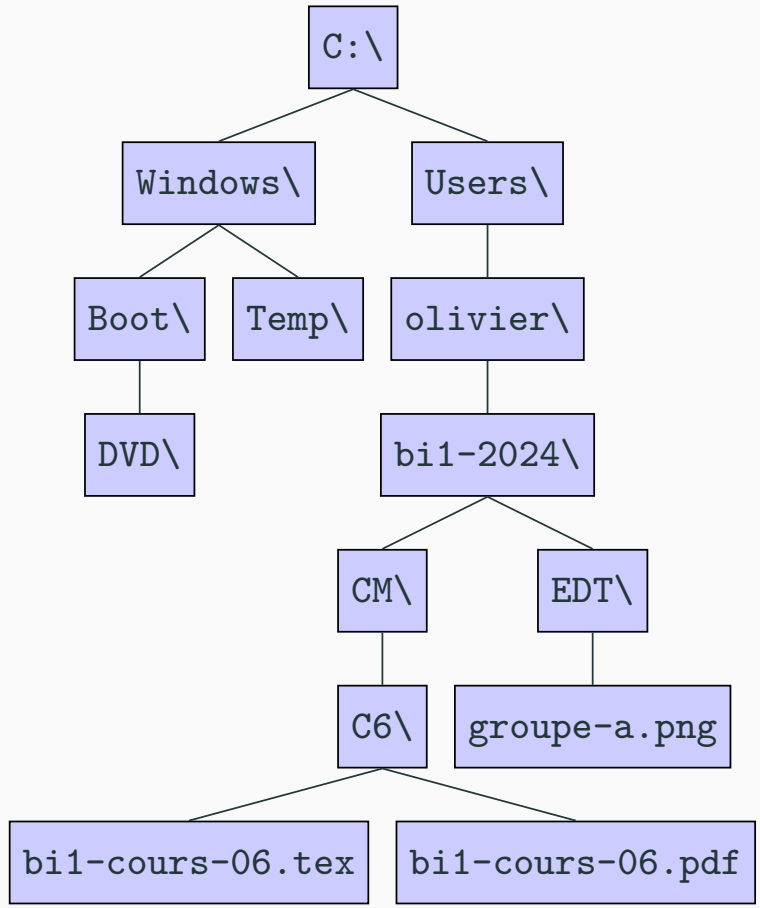


- Un répertoire est relié à la racine par un chemin

Arborescence Unix



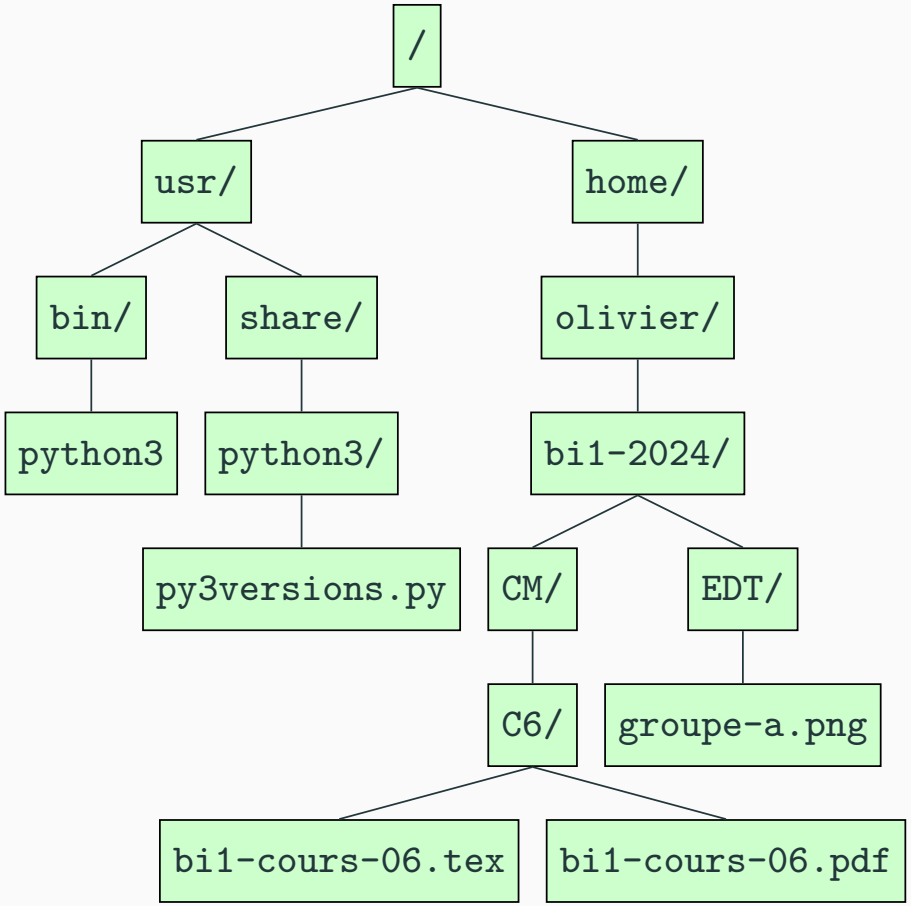
Arborescence Windows



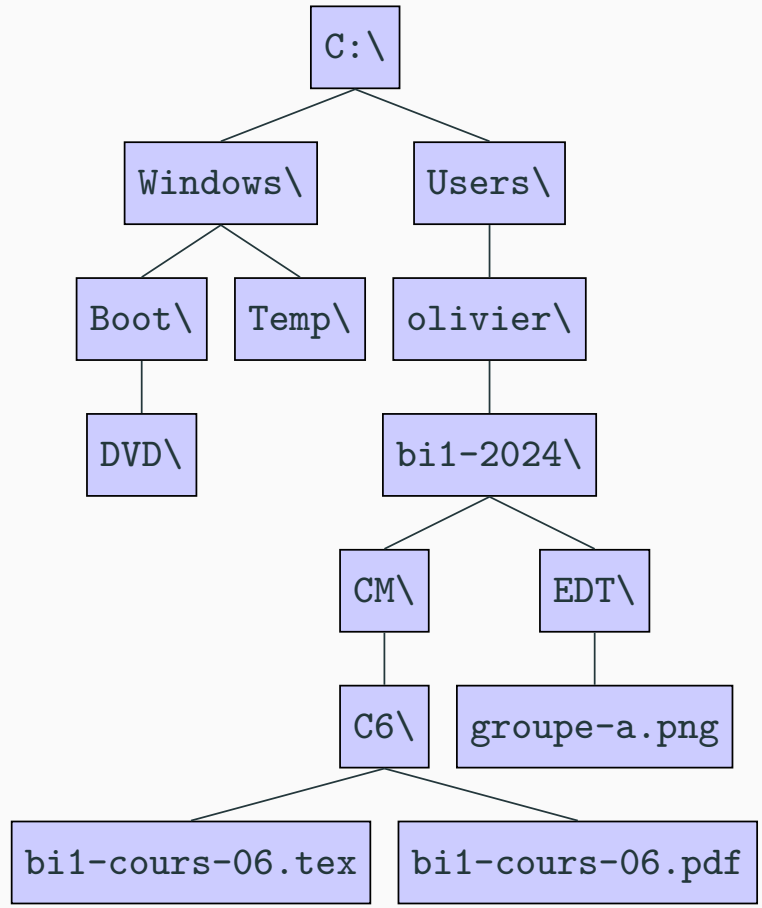
- Chemin :
  - Unix `/home/olivier/bi1-2024/EDT/groupe-a.png`

- Un répertoire est relié à la racine par un chemin

Arborescence Unix



Arborescence Windows



- Chemin :
  - Unix `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - Windows `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs

- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs
- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`
- Chemins relatifs : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - `/home/olivier/bi1-2024/EDT/groupe-a.png`
    - `/home/olivier/bi1-2024/sauvegarde/EDT/groupe-a.png`

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs
- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`
- Chemins relatifs : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - `/home/olivier/bi1-2024/EDT/groupe-a.png`
    - `/home/olivier/bi1-2024/sauvegarde/EDT/groupe-a.png`
  - Cela dépend si je suis dans `sauvegarde` ou `bi1-2024`

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs
- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`
- Chemins relatifs : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - `/home/olivier/bi1-2024/EDT/groupe-a.png`
    - `/home/olivier/bi1-2024/sauvegarde/EDT/groupe-a.png`
  - Cela dépend si je suis dans `sauvegarde` ou `bi1-2024`
- Le répertoire parent (du dessus) se note `..`
  - Si je suis dans `/home/olivier`, `..` est le chemin vers `/home`
  - La racine `/` est le seul répertoire sans parent.

- Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - Les chemins absolus
  - Les chemins relatifs
- Chemins absolus : chemin complet à partir de la racine.
  - `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`
- Chemins relatifs : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - `/home/olivier/bi1-2024/EDT/groupe-a.png`
    - `/home/olivier/bi1-2024/sauvegarde/EDT/groupe-a.png`
  - Cela dépend si je suis dans `sauvegarde` ou `bi1-2024`
- Le répertoire parent (du dessus) se note `..`
  - Si je suis dans `/home/olivier`, `..` est le chemin vers `/home`
  - La racine `/` est le seul répertoire sans parent. :'(

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

```shell
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'
```

  - `listdir()` renvoie les fichiers du répertoire (Unix : `ls`)

SHELL
```
>>>
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'
```

  - `listdir()` renvoie les fichiers du répertoire (Unix : `ls`)

SHELL
```
>>> os.listdir()
```

- Le module `os` permet d'utiliser des équivalents aux commandes Unix
  - `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

SHELL
```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

  - `chdir(...)` permet de changer de répertoire (Unix : `cd`)

SHELL
```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'
```

  - `listdir()` renvoie les fichiers du répertoire (Unix : `ls`)

SHELL
```
>>> os.listdir()
['exo2.py', 'exo1.py', 'sujet.pdf']
```

- Un chemin peut être codé en Python par une chaîne.

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>>
```

SHELL
```
>>>
```

- On peut construire un chemin de manière portable.

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
```

SHELL
```
>>> os.name #Sous Windows
```

- On peut construire un chemin de manière portable.

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
'posix'
```

SHELL
```
>>> os.name #Sous Windows
'nt'
```

- On peut construire un chemin de manière portable.

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
'posix'
```

SHELL
```
>>> os.name #Sous Windows
'nt'
```

- On peut construire un chemin de manière portable.

SHELL
```
>>>
```

SHELL
```
>>>
```

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
'posix'
```

SHELL
```
>>> os.name #Sous Windows
'nt'
```

- On peut construire un chemin de manière portable.

SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Linux
```

SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Windows
```

- Un chemin peut être codé en Python par une chaîne.
- Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - les \ doivent être doublés
  - \ est un caractère d'échappement dans une chaîne (`"\n \' \\ "`)
  - Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- Un bon logiciel doit fonctionner sur tous les OS.
- On peut demander en Python sur quel système on travaille.

SHELL
```
>>> os.name #Sous Linux
'posix'
```

SHELL
```
>>> os.name #Sous Windows
'nt'
```

- On peut construire un chemin de manière portable.

SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Linux
'../QCM/q1.xml'
```

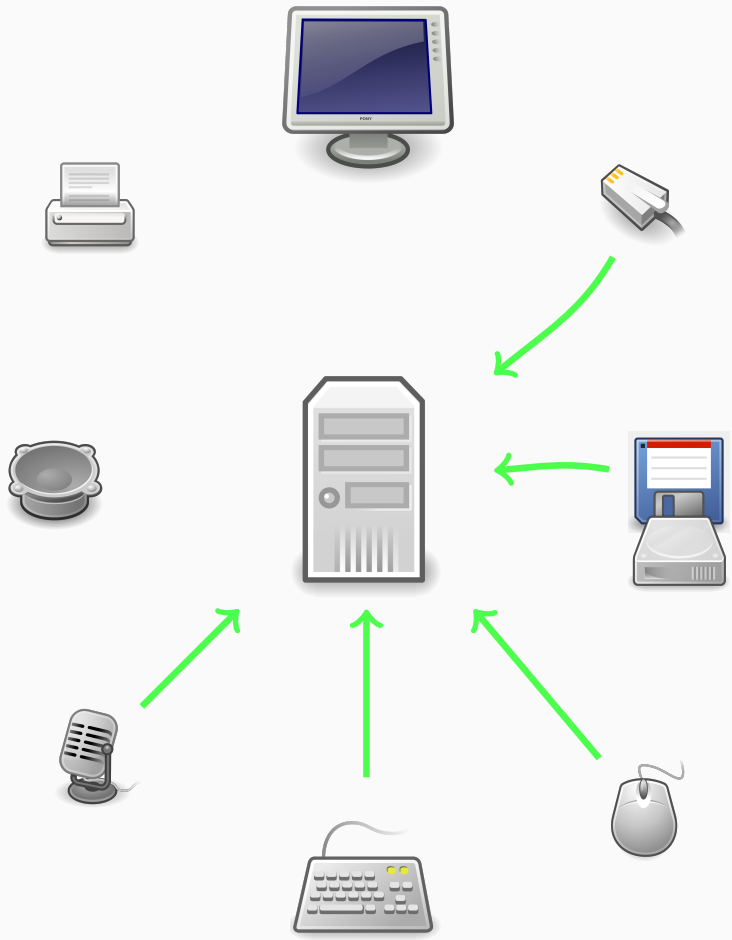
SHELL
```
>>> os.path.join('..','QCM','q1.xml') # Sous Windows
'..\\QCM\\q1.xml'
```

`os.getcwd()`	renvoie le répertoire courant
`os.chdir(path)`	change de répertoire courant
`os.listdir(path='.')`	liste des fichiers et répertoires
`os.path.join(path1,path2,…)`	construction portable d'un chemin
`os.remove(path)`	suppression d'un fichier
`os.path.isfile(path)`	test d'existence d'un fichier
`os.path.isdir(path)`	test d'existence d'un répertoire
`os.path.split(path)`	pour extraire le fichier d'un chemin
`os.path.getsize(path)`	la taille d'un fichier

`https://docs.python.org/fr/3.7/library/os.html`

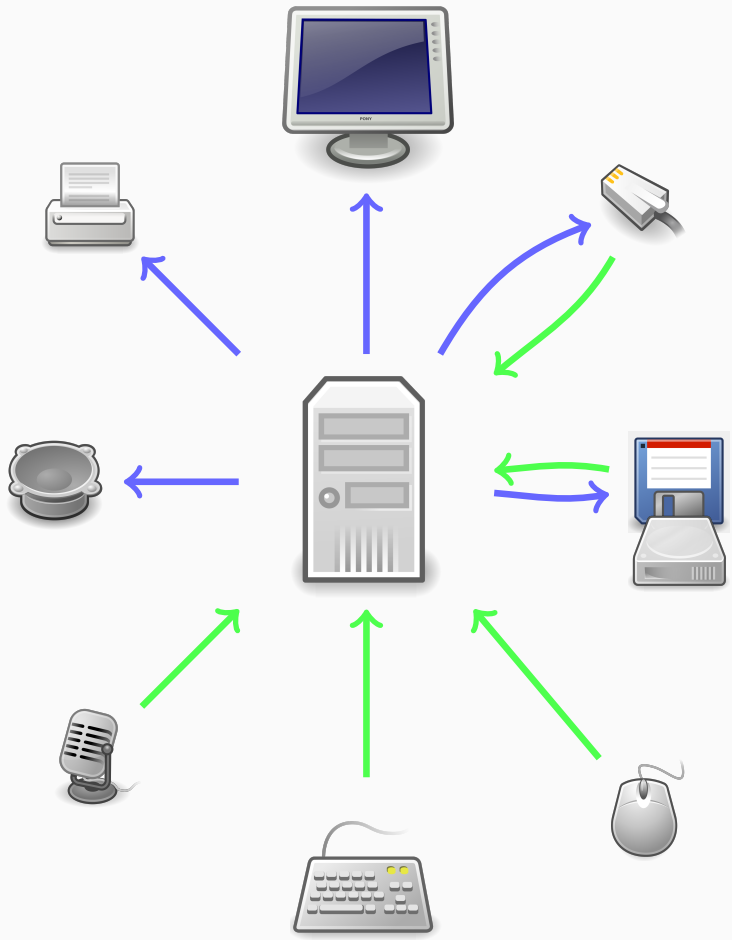
Sommaire

L'ordinateur communique avec le reste du monde



L'ordinateur communique avec le reste du monde

- Entrée :
  - Souris et clavier
  - Micro, caméra
  - Disque dur, clé usb, disquette
  - Câble éthernet (internet)



L'ordinateur communique avec le reste du monde

- Entrée :
  - Souris et clavier
  - Micro, caméra
  - Disque dur, clé usb, disquette
  - Câble éthernet (internet)
- Sortie :
  - Écran
  - Imprimante
  - Disque dur, clé usb, disquette
  - Enceinte

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>>
```

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>> fichier
```

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>> fichier
<_io.TextIOWrapper name='test.txt' mode='w' encoding='utf-8'>
```

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>> fichier
<_io.TextIOWrapper name='test.txt' mode='w' encoding='utf-8'>
```

- La valeur de `fichier`, renvoyé par `open`, est un descripteur de fichier.

Ouverture d'un fichier en écriture

- On souhaite créer un fichier `test.txt` contenant des résultats.
- Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

SCRIPT
```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - Si un ancien fichier de ce nom existe, il sera remplacé.

SHELL
```
>>> fichier
<_io.TextIOWrapper name='test.txt' mode='w' encoding='utf-8'>
```

- La valeur de `fichier`, renvoyé par `open`, est un descripteur de fichier.
- L'encodage par défaut dépend de la machine.

Écriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.

Écriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.

Écriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.
  - s'il existait déjà : on l'écrase et on recommence à zéro.

Écriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.
  - s'il existait déjà : on l'écrase et on recommence à zéro.
- On écrit dans le fichier avec la méthode `write` du descripteur de fichier.

SCRIPT

```
fichier.write('Bonjour tout le monde !\n')
fichier.write('Voici un texte écrit depuis Python !\n')
```

Écriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.
  - s'il existait déjà : on l'écrase et on recommence à zéro.
- On écrit dans le fichier avec la méthode `write` du descripteur de fichier.

SCRIPT

```
fichier.write('Bonjour tout le monde !\n')
fichier.write('Voici un texte écrit depuis Python !\n')
```

- Il faut déposer le caractère de retour à la ligne `'\n'`,
  - sinon, le prochain `write` prendra effet sur la même ligne.

- Une fois le fichier ouvert, il est prêt à recevoir des données.
  - si le fichier n'existait pas, on va le créer.
  - s'il existait déjà : on l'écrase et on recommence à zéro.
- On écrit dans le fichier avec la méthode `write` du descripteur de fichier.

SCRIPT
```
fichier.write('Bonjour tout le monde !\n')
fichier.write('Voici un texte écrit depuis Python !\n')
```

- Il faut déposer le caractère de retour à la ligne `'\n'`,
  - sinon, le prochain `write` prendra effet sur la même ligne.
- Les écritures sont mises en tampon ; elles ne prennent pas forcément effet immédiatement. À la fin du traitement, il faut fermer le fichier (`close`) pour que tout soit bien écrit.

SCRIPT
```
fichier.close()
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

```shell
>>>
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>>
```
SHELL

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>>
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> fichier.write(n)
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> fichier.write(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  write() argument must be str, not int
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> fichier.write(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  write() argument must be str, not int
```

- Pour écrire un nombre, il faut d'abord le convertir en chaîne

SCRIPT
```
fichier.write(str(n))
fichier.write(" est l'année de naissance de Python.\n")
```

- Dans le fichier, on ne peut écrire que des chaînes de caractères.

SHELL
```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> fichier.write(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  write() argument must be str, not int
```

- Pour écrire un nombre, il faut d'abord le convertir en chaîne

SCRIPT
```
fichier.write(str(n))
fichier.write(" est l'année de naissance de Python.\n")
```

- ou encore...

SCRIPT
```
fichier.write(f"{n} est l'année de naissance de Python.\n")
```

Exemple : générer une table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    fichier=open('table'+str(n)+'.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        fichier.write(str(i)+'*'+ str(n)+'='+str(i*n)+'\n')
    # étape 3 : fermeture du fichier
    fichier.close()

créer_table(7)
```

Exemple : générer une table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    fichier=open('table'+str(n)+'.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        fichier.write(str(i)+'*'+ str(n)+'='+str(i*n)+'\n')
    # étape 3 : fermeture du fichier
    fichier.close()

créer_table(7)
```

- Pour lire le fichier `table7.txt`, on utilise un lecteur de fichier texte.

TABLE7.TXT

```
1*7=7
2*7=14
3*7=21
4*7=28
5*7=35
6*7=42
7*7=49
8*7=56
9*7=63
10*7=70
```

Exemple : générer une jolie table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    fichier = open(f'table{n}.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')
    # étape 3 : fermeture du fichier
    fichier.close()

créer_table(7)
```

Exemple : générer une jolie table de multiplication

SCRIPT

```
def créer_table(n):
    """crée un fichier contenant la table
    de multiplication de n"""
    # étape 1 : ouverture du fichier
    fichier = open(f'table{n}.txt','w',encoding='utf-8')
    # étape 2 : écriture dans le fichier
    for i in range(1, 11):
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')
    # étape 3 : fermeture du fichier
    fichier.close()

créer_table(7)
```

- Les nombres sont correctement alignés.

TABLE7.TXT

```
 1 × 7 =  7
 2 × 7 = 14
 3 × 7 = 21
 4 × 7 = 28
 5 × 7 = 35
 6 × 7 = 42
 7 × 7 = 49
 8 × 7 = 56
 9 × 7 = 63
10 × 7 = 70
```

Écrire à la fin d'un fichier

- Il peut être intéressant d'ajouter des lignes à un fichier. Il faut alors l'ouvrir en écriture en mode `'a'` (ajout) et non `'w'`.

SCRIPT

```
fichier = open('test.txt', 'a', encoding = 'utf-8')
```

- Il peut être intéressant d'ajouter des lignes à un fichier. Il faut alors l'ouvrir en écriture en mode `'a'` (ajout) et non `'w'`.

SCRIPT
```
fichier = open('test.txt', 'a', encoding = 'utf-8')
```

- Pour ajouter deux lignes à la fin du fichier `table7.txt`.

SCRIPT
```
def allonger_fichier(n):
    fichier = open(f'table{n}.txt','a',encoding='utf-8')
    for i in range(11, 13):
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')
    fichier.close()
```

- Il peut être intéressant d'ajouter des lignes à un fichier. Il faut alors l'ouvrir en écriture en mode `'a'` (ajout) et non `'w'`.

SCRIPT
```
fichier = open('test.txt', 'a', encoding = 'utf-8')
```

- Pour ajouter deux lignes à la fin du fichier `table7.txt`.

SCRIPT
```
def allonger_fichier(n):
    fichier = open(f'table{n}.txt','a',encoding='utf-8')
    for i in range(11, 13):
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')
    fichier.close()
```

- Remarque : il n'est pas possible de supprimer des lignes dans un fichier directement (mais on peut créer un nouveau fichier et détruire l'ancien).

Sommaire

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

```shell
>>>
```

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

```shell
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

Lecture d'un fichier : `read`

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```
SHELL

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne

```
>>>
```
SHELL

Lecture d'un fichier : `read`

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```
SHELL

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

```
>>> texte = fichier.read()
```
SHELL

Lecture d'un fichier : `read`

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>>
```

Lecture d'un fichier : `read`

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>> fichier.close()
```

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>> fichier.close()
>>>
```

Lecture d'un fichier : `read`

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>> fichier.close()
>>> texte # contient tout le fichier !
```

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```
SHELL

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

```
>>> texte = fichier.read()
>>> fichier.close()
>>> texte # contient tout le fichier !
'5 × 1 =  5\n5 × 2 = 10\n5 × 3 = 15\n5 × 4 = 20\n'
>>>
```
SHELL

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?
- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>> fichier.close()
>>> texte # contient tout le fichier !
'5 × 1 =  5\n5 × 2 = 10\n5 × 3 = 15\n5 × 4 = 20\n'
>>> print(texte)
```

- Problème inverse : comment lire (`read`) le fichier `table5.txt` ?

- On doit connaître son encodage ; on sait qu'il est en `utf-8`.

SHELL
```
>>> fichier = open('table5.txt', 'r', encoding='utf-8')
```

- On peut lire d'un seul coup la totalité du fichier dans une seule chaîne
  - avec la méthode `read()`.

SHELL
```
>>> texte = fichier.read()
>>> fichier.close()
>>> texte # contient tout le fichier !
'5 × 1 =  5\n5 × 2 = 10\n5 × 3 = 15\n5 × 4 = 20\n'
>>> print(texte)
5 × 1 =  5
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
```

- On peut itérer directement sur les lignes avec une boucle `for`

SCRIPT

```
def affiche_et_compte(f):
    i=0 # pour compter le nombre de lignes
    fichier = open(f, 'r', encoding ='utf-8')
    for ligne in fichier:
        print('>' ,ligne, end = '')
        i=i+1
    fichier.close()
    return i # nombre de lignes
```

SHELL

```
>>>
```

- On peut itérer directement sur les lignes avec une boucle `for`

SCRIPT

```
def affiche_et_compte(f):
    i=0 # pour compter le nombre de lignes
    fichier = open(f, 'r', encoding ='utf-8')
    for ligne in fichier:
        print('>' ,ligne, end = '')
        i=i+1
    fichier.close()
    return i # nombre de lignes
```

SHELL

```
>>> affiche_et_compte('table5.txt')
```

- On peut itérer directement sur les lignes avec une boucle `for`

SCRIPT

```
def affiche_et_compte(f):
    i=0 # pour compter le nombre de lignes
    fichier = open(f, 'r', encoding ='utf-8')
    for ligne in fichier:
        print('>' ,ligne, end = '')
        i=i+1
    fichier.close()
    return i # nombre de lignes
```

SHELL

```
>>> affiche_et_compte('table5.txt')
> 5 × 1 =  5
> 5 × 2 = 10
> 5 × 3 = 15
> 5 × 4 = 20
4
```

Lire, écrire, compter : objectifs

- Une méthode classique de travail sur fichier texte
  - on prend un fichier en entrée
  - on produit un autre fichier en sortie

Lire, écrire, compter : objectifs

- Une méthode classique de travail sur fichier texte
  - on prend un fichier en entrée
  - on produit un autre fichier en sortie
- Exemple :
  - lecture d'un fichier : chaque ligne contient des nombres
  - écriture d'un fichier : chaque ligne contient la somme de ces nombres.

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

SOMME.TXT

```
246
208
324
190
269
314
307
301
```

- Pour générer le fichier aléatoire, on doit écrire 8 lignes de 5 colonnes chacune ; on utilise deux boucles imbriquées.

SCRIPT

```
def créer_aléa(nom_fichier):
    fichier = open(nom_fichier, 'w', encoding = 'utf-8')
    for ligne in range(8):   # je produis 8 lignes
        for colonne in range(5): # chaque ligne a 5 colonnes
            fichier.write(f'{randint(10,99)} ')
        fichier.write('\n') # je vais à la ligne
    fichier.close()
```

SHELL

```
>>>
```

- Pour générer le fichier aléatoire, on doit écrire 8 lignes de 5 colonnes chacune ; on utilise deux boucles imbriquées.

SCRIPT

```
def créer_aléa(nom_fichier):
    fichier = open(nom_fichier, 'w', encoding = 'utf-8')
    for ligne in range(8):   # je produis 8 lignes
        for colonne in range(5): # chaque ligne a 5 colonnes
            fichier.write(f'{randint(10,99)} ')
        fichier.write('\n') # je vais à la ligne
    fichier.close()
```

SHELL

```
>>> créer_aléa('aléa.txt')
```

- Pour générer le fichier aléatoire, on doit écrire 8 lignes de 5 colonnes chacune ; on utilise deux boucles imbriquées.

SCRIPT

```
def créer_aléa(nom_fichier):
    fichier = open(nom_fichier, 'w', encoding = 'utf-8')
    for ligne in range(8):    # je produis 8 lignes
        for colonne in range(5): # chaque ligne a 5 colonnes
            fichier.write(f'{randint(10,99)} ')
        fichier.write('\n') # je vais à la ligne
    fichier.close()
```

SHELL

```
>>> créer_aléa('aléa.txt')
```

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

Lire, écrire, compter : générer `somme.txt`

- Pour générer le fichier `somme.txt`,
  - on lit `aléa.txt` ligne par ligne
  - à chaque ligne lue, on écrit la somme de la ligne dans `somme.txt`

Lire, écrire, compter : générer `somme.txt`

- Pour générer le fichier `somme.txt`,
  - on lit `aléa.txt` ligne par ligne
  - à chaque ligne lue, on écrit la somme de la ligne dans `somme.txt`

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # je calcule la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

- Pour générer le fichier `somme.txt`,
  - on lit `aléa.txt` ligne par ligne
  - à chaque ligne lue, on écrit la somme de la ligne dans `somme.txt`

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # je calcule la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

ALÉA.TXT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

SOMME.TXT

```
246
208
324
190
269
314
307
301
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>>
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                    # L1
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                    # L1
['83', '21', '10', '34', '98']
>>>
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                    # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                    # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>>
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                          # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98])                               # S2
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                          # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98])                               # S2
246
>>>
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                          # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98])                               # S2
246
>>> f'{246}\n' # Ce qu'on écrit dans le fichier
```

- Pour ceux qui ronflent au fond de l'amphi.
  - Pour comprendre ce que fait un programme…
  - … on arrête de se tourner les pouces et on sort sa console Python !

SCRIPT

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée  = open('aléa.txt',  'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SHELL

```
>>> '83 21 10 34 98\n'.split()                          # L1
['83', '21', '10', '34', '98']
>>> [ int(x) for x in ['83','21','10','34','98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98])                               # S2
246
>>> f'{246}\n' # Ce qu'on écrit dans le fichier
'246\n'
```

- Si le fichier n'existe pas.

SHELL
```
>>>
```

- Si le fichier n'existe pas.

SHELL
```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
```

- Si le fichier n'existe pas.

SHELL

```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'complot.py'
```

- Si le fichier n'existe pas.

SHELL
```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

SHELL
```
>>>
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
```

- Si le fichier n'existe pas.

```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```
SHELL

- S'il n'y a pas les permissions pour modifier le fichier.

```
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```
SHELL

Les exceptions liées aux fichiers

- Si le fichier n'existe pas.

```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'complot.py'
```
SHELL

- S'il n'y a pas les permissions pour modifier le fichier.

```
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError: [Errno 13] Permission denied: 'lecture_seule.txt'
```
SHELL

- Si le descripteur de fichier a été fermé.

```
>>>
```
SHELL

- Si le fichier n'existe pas.

```shell
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```shell
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```shell
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>>
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
>>>
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
>>> fichier.read()
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
>>> fichier.read()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  I/O operation on closed file.
```

- Si le fichier n'existe pas.

```
SHELL
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError:  [Errno 2] No such file or directory:
'complot.py'
```

- S'il n'y a pas les permissions pour modifier le fichier.

```
SHELL
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError:  [Errno 13] Permission denied: 'lecture_seule.txt'
```

- Si le descripteur de fichier a été fermé.

```
SHELL
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
>>> fichier.read()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  I/O operation on closed file.
```

- etc.

- Ouverture en lecture (`read`) :

SCRIPT

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```

- Ouverture en lecture (`read`) :

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```
SCRIPT

- Ouverture en écriture (`write` ou `add`) :

```
fichier = open('fichier.txt', 'w', encoding='utf-8')
```
SCRIPT

ou

```
fichier = open('fichier.txt', 'a', encoding='utf-8')
```
SCRIPT

- Ouverture en lecture (`read`) :

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```
SCRIPT

- Ouverture en écriture (`write` ou `add`) :

```
fichier = open('fichier.txt', 'w', encoding='utf-8')
```
SCRIPT

ou

```
fichier = open('fichier.txt', 'a', encoding='utf-8')
```
SCRIPT

- Lecture :
  - `fichier.read()`
  - `for ligne in fichier:`

- Ouverture en lecture (`read`) :

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```
SCRIPT

- Ouverture en écriture (`write` ou `add`) :

```
fichier = open('fichier.txt', 'w', encoding='utf-8')
```
SCRIPT

ou

```
fichier = open('fichier.txt', 'a', encoding='utf-8')
```
SCRIPT

- Lecture :
  - `fichier.read()`
  - `for ligne in fichier:`
- Écriture :
  - `fichier.write(chaîne)`

- Ouverture en lecture (`read`) :

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```
SCRIPT

- Ouverture en écriture (`write` ou `add`) :

```
fichier = open('fichier.txt', 'w', encoding='utf-8')
```
SCRIPT

ou

```
fichier = open('fichier.txt', 'a', encoding='utf-8')
```
SCRIPT

- Lecture :
  - `fichier.read()`
  - `for ligne in fichier:`
- Écriture :
  - `fichier.write(chaîne)`
- Fermeture :

```
fichier.close()
```
SCRIPT

Merci pour votre attention
Questions

?

Cours 8 — Ensembles, dictionnaires et fichiers texte

Licence