Apprentissage de la programmation

avec Python 3

Christian VINCENT

LETTRES SORBONNE UNIVERSITÉ

L1 - Semestre 1 - année 2023-2024

Fonctions, conteneurs de *Python*

- Chapitre III -

Définition

Une fonction regroupe plusieurs instructions dans un même bloc. La fonction est appelée ensuite par son nom, nom choisi en dehors de la liste des mots réservés (comme type, print, for, while, ...)

Définition

Une fonction regroupe plusieurs instructions dans un même bloc. La fonction est appelée ensuite par son nom, nom choisi en dehors de la liste des mots réservés (comme type, print, for, while, ...)

- Les fonctions sont très pratiques lors de l'exécution de tâches qui seront nécessaires plusieurs fois dans un programme.

Définition

Une fonction regroupe plusieurs instructions dans un même bloc. La fonction est appelée ensuite par son nom, nom choisi en dehors de la liste des mots réservés (comme type, print, for, while, ...)

- Les fonctions sont très pratiques lors de l'exécution de tâches qui seront nécessaires plusieurs fois dans un programme.
- *print()*, *input()*, *len()*, ... sont des fonctions déjà intégrées dans *Python*.

Définition

Une fonction regroupe plusieurs instructions dans un même bloc. La fonction est appelée ensuite par son nom, nom choisi en dehors de la liste des mots réservés (comme type, print, for, while, ...)

- Les fonctions sont très pratiques lors de l'exécution de tâches qui seront nécessaires plusieurs fois dans un programme.
- *print()*, *input()*, *len()*, ... sont des fonctions déjà intégrées dans *Python*. En créer d'autres permet d'étendre les capacités de *Python*.

Définition

Une fonction regroupe plusieurs instructions dans un même bloc. La fonction est appelée ensuite par son nom, nom choisi en dehors de la liste des mots réservés (comme type, print, for, while, ...)

- Les fonctions sont très pratiques lors de l'exécution de tâches qui seront nécessaires plusieurs fois dans un programme.
- *print()*, *input()*, *len()*, ... sont des fonctions déjà intégrées dans *Python*. En créer d'autres permet d'étendre les capacités de *Python*.
- On définit une fonction avec le mot clé **def**. C'est une instruction composée (donc suivie de 2 points (:)

Syntaxe de la définition d'une fonction :

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
```

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    # début du bloc
```

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    ♯ début du bloc
    instruction2
```

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    ♯ début du bloc
    instruction2
    ...
```

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    # début du bloc
    instruction2
    ...
    instructionN                    # fin du bloc
```

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    ♯ début du bloc
    instruction2
    ...
    instructionN                    ♯ fin du bloc
```

Remarque : les paramètres ne sont pas toujours nécessaires, mais les parenthèses le sont !

Syntaxe de la définition d'une fonction :

Définir une fonction

```
def nom_de_la_fonction(paramètre1, paramètre2, ...) :
    instruction1                    ♯ début du bloc
    instruction2
    ...
    instructionN                    ♯ fin du bloc
```

Remarque : les paramètres ne sont pas toujours nécessaires, mais les parenthèses le sont !

au minimum...

```
def mafonction() :
    xxxxxxxxxxxx
    xxxxxxxxxxxx
```

Exemple de définition de fonction (sans paramètre) :

Exemple de définition de fonction (sans paramètre) :

Date du jour

```
def donneLaDate() :
    from datetime import date
    print(date.today())
```

Exemple de définition de fonction (sans paramètre) :

Date du jour

```
def donneLaDate() :
    from datetime import date
    print(date.today())
```

À chaque fois que dans le code d'un programme *Python*, il y aura l'instruction donneLaDate(), la date du jour sera affichée à l'écran !

Exemple de définition de fonction (sans paramètre) :

Date du jour

```
def donneLaDate() :
    from datetime import date
    print(date.today())
```

À chaque fois que dans le code d'un programme *Python*, il y aura l'instruction donneLaDate(), la date du jour sera affichée à l'écran !

Dans un programme

```
donneLaDate()
```

renvoie...

Exemple de définition de fonction (sans paramètre) :

Date du jour

```
def donneLaDate() :
    from datetime import date
    print(date.today())
```

À chaque fois que dans le code d'un programme *Python*, il y aura l'instruction donneLaDate(), la date du jour sera affichée à l'écran !

Dans un programme

```
donneLaDate()
```

renvoie...

Date du jour

```
2019-10-11
```

- La fonction précédente (**donneLaDate()**) n'avait besoin d'aucune donnée extérieure pour fonctionner.

- La fonction précédente (**donneLaDate()**) n'avait besoin d'aucune donnée extérieure pour fonctionner.
- Une fonction peut opérer sur une ou plusieurs données qu'on lui soumet.

- La fonction précédente (**donneLaDate()**) n'avait besoin d'aucune donnée extérieure pour fonctionner.
- Une fonction peut opérer sur une ou plusieurs données qu'on lui soumet.
- Ces données qu'on lui soumet sont appelées **paramètres**.

- La fonction précédente (**donneLaDate()**) n'avait besoin d'aucune donnée extérieure pour fonctionner.
- Une fonction peut opérer sur une ou plusieurs données qu'on lui soumet.
- Ces données qu'on lui soumet sont appelées **paramètres**.

Voici une fonction qui calcule l'aire d'un rectangle !

- La fonction précédente (**donneLaDate()**) n'avait besoin d'aucune donnée extérieure pour fonctionner.
- Une fonction peut opérer sur une ou plusieurs données qu'on lui soumet.
- Ces données qu'on lui soumet sont appelées **paramètres**.

Voici une fonction qui calcule l'aire d'un rectangle !

Définition de la fonction aireRect()

```
def aireRect(Long, larg) :
    aire = Long * larg
    print("L'aire du rectangle est :", aire)
```

Usage : dans un programme...

Usage : dans un programme...

programme : la fonction *aireRect()* a été définie auparavant

```
L = int(input("Donne une longueur : "))
l = int(input("Donne une largeur : "))
aireRect(L, l)
```

Usage : dans un programme...

programme : la fonction *aireRect()* a été définie auparavant

```
L = int(input("Donne une longueur : "))
l = int(input("Donne une largeur : "))
aireRect(L, l)
```

Remarques :

- *L* et *l* doivent être données dans le bon ordre !

Usage : dans un programme...

programme : la fonction *aireRect()* a été définie auparavant

```
L = int(input("Donne une longueur : "))
l = int(input("Donne une largeur : "))
aireRect(L, l)
```

Remarques :

- *L* et *l* doivent être données dans le bon ordre !
- Dans la fonction *aireRect()*, L sera Long et l sera larg.

Autre exemple :

Autre exemple : une fonction avec son appel dans un programme.

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))
```

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))

phrase = input("Entre une phrase :")
```

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))

phrase = input("Entre une phrase :")
print("Analyse de la phrase...")
```

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))

phrase = input("Entre une phrase :")
print("Analyse de la phrase...")
longchaine(phrase)
```

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))

phrase = input("Entre une phrase :")
print("Analyse de la phrase...")
longchaine(phrase)
```

Remarques :

- on saute une ligne entre la définition d'une fonction et le programme

Autre exemple : une fonction avec son appel dans un programme.

longueur d'une chaîne :

```
def longchaine(p) :
    print("La chaîne a pour longueur : ", len(p))

phrase = input("Entre une phrase :")
print("Analyse de la phrase...")
longchaine(phrase)
```

Remarques :

- on saute une ligne entre la définition d'une fonction et le programme
- on place toujours les définitions de fonctions en tête de programme.

- Vous avez vu des fonctions propres à *Python*, certaines ne renvoient aucune valeur comme la fonction **print()**

- Vous avez vu des fonctions propres à *Python*, certaines ne renvoient aucune valeur comme la fonction **print()**
- D'autres comme **input()** permettent de récupérer une donnée. On dit alors que la fonction **input()** renvoie une donnée.

- Vous avez vu des fonctions propres à *Python*, certaines ne renvoient aucune valeur comme la fonction **print()**
- D'autres comme **input()** permettent de récupérer une donnée. On dit alors que la fonction **input()** renvoie une donnée.
- On réalise une récupération de donnée en *Python* avec l'instruction **return** dont voici un exemple d'usage.

- Vous avez vu des fonctions propres à *Python*, certaines ne renvoient aucune valeur comme la fonction **print()**
- D'autres comme **input()** permettent de récupérer une donnée. On dit alors que la fonction **input()** renvoie une donnée.
- On réalise une récupération de donnée en *Python* avec l'instruction **return** dont voici un exemple d'usage.

usage de return : fonction qui renvoie la dernière lettre d'une chaîne

```
def derLettre(mot) :
    last = mot[-1]
    return last
```

- Vous avez vu des fonctions propres à *Python*, certaines ne renvoient aucune valeur comme la fonction **print()**
- D'autres comme **input()** permettent de récupérer une donnée. On dit alors que la fonction **input()** renvoie une donnée.
- On réalise une récupération de donnée en *Python* avec l'instruction **return** dont voici un exemple d'usage.

usage de return : fonction qui renvoie la dernière lettre d'une chaîne

```
def derLettre(mot) :
    last = mot[-1]
    return last
```

usage

```
a = "lapin"
print("Le mot", a, "se termine par", derLettre(a))
```

Une fonction peut renvoyer plusieurs données : dans une liste, un tuple, un dictionnaire, ...

Une fonction peut renvoyer plusieurs données : dans une liste, un tuple, un dictionnaire, ...

Autre exemple

```
def bilanMot(mot) :
    last = mot[-1]
    prem = mot[0]
    lg = len(mot)
    return [prem, last, lg]
```

Une fonction peut renvoyer plusieurs données : dans une liste, un tuple, un dictionnaire, ...

Autre exemple

```
def bilanMot(mot) :
    last = mot[-1]
    prem = mot[0]
    lg = len(mot)
    return [prem, last, lg]
```

usage

```
a = "lapin"
retour = bilanMot(a)
print("Le mot", a, "a pour longueur", retour[2])
print("Sa 1e lettre est", retour[0], "et sa dernière", retour[1])
```

Les fonctions peuvent s'imbriquer...

Les fonctions peuvent s'imbriquer...

fonctions double et triple

```
def double(nb) :
    return 2*nb
```

Les fonctions peuvent s'imbriquer...

fonctions double et triple

```
def double(nb) :
    return 2*nb

def triple(nb) :
    return 3*nb
```

Les fonctions peuvent s'imbriquer...

fonctions double et triple

```
def double(nb) :
    return 2*nb

def triple(nb) :
    return 3*nb
```

usage

```
nombre = int(input("Donne un nombre : "))
print("Double de", nombre, double(nombre))
print("Triple de", nombre, triple(nombre))
print("Sextuple de", nombre, triple(double(nombre)))
```

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**
- La **docstring** est composée d'une zone de texte placée immédiatement après la 1^e ligne de définition de la fonction.

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**
- La **docstring** est composée d'une zone de texte placée immédiatement après la 1^e ligne de définition de la fonction.
- Ce texte explique :
  - ce qu'attend la fonction

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**
- La **docstring** est composée d'une zone de texte placée immédiatement après la 1^e ligne de définition de la fonction.
- Ce texte explique :
  - ce qu'attend la fonction
  - ce que renvoie la fonction

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**
- La **docstring** est composée d'une zone de texte placée immédiatement après la 1^e ligne de définition de la fonction.
- Ce texte explique :
  - ce qu'attend la fonction
  - ce que renvoie la fonction
  - ce que fait la fonction

- Les fonctions doivent être documentées : on utilise pour cela la **docstring**
- La **docstring** est composée d'une zone de texte placée immédiatement après la 1^e ligne de définition de la fonction.
- Ce texte explique :
  - ce qu'attend la fonction
  - ce que renvoie la fonction
  - ce que fait la fonction
- La visualisation de ces explications se fait avec la fonction **help()**

Exemple de docstring

```
def double(nb) :
    """Prend un nombre en entrée. Renvoie un nombre qui est
    le double du nombre entré."""
    return nb*2
```

Exemple de docstring

```
def double(nb) :
    """Prend un nombre en entrée. Renvoie un nombre qui est
    le double du nombre entré."""
    return nb*2
```

Appel de la docstring avec : help(double), puis en réponse

Exemple de docstring

```
def double(nb) :
    """Prend un nombre en entrée. Renvoie un nombre qui est
    le double du nombre entré."""
    return nb*2
```

Appel de la docstring avec : help(double), puis en réponse

help on function double

```
double(nb)
    Prend un nombre en entrée. Renvoie un nombre qui est le
double du nombre entré.
(END)
```

Exemple de docstring

```
def double(nb) :
    """Prend un nombre en entrée. Renvoie un nombre qui est
    le double du nombre entré."""
    return nb*2
```

Appel de la docstring avec : help(double), puis en réponse

help on function double

```
double(nb)
    Prend un nombre en entrée. Renvoie un nombre qui est le
double du nombre entré.
(END)
```

un appui sur la touche **q** ferme l'aide et retourne à l'éditeur.

- Tout programmeur sérieux écrit une docstring pour ses fonctions.

- Tout programmeur sérieux écrit une docstring pour ses fonctions.
- Cela est très utile quand vous ou une autre personne accède au code, ce que fait une fonction est immédiatement visualisable.

- Tout programmeur sérieux écrit une docstring pour ses fonctions.
- Cela est très utile quand vous ou une autre personne accède au code, ce que fait une fonction est immédiatement visualisable.
- Cela n'empêche pas par ailleurs de commenter son code.

- Tout programmeur sérieux écrit une docstring pour ses fonctions.
- Cela est très utile quand vous ou une autre personne accède au code, ce que fait une fonction est immédiatement visualisable.
- Cela n'empêche pas par ailleurs de commenter son code.

Exemple :

- Tout programmeur sérieux écrit une docstring pour ses fonctions.
- Cela est très utile quand vous ou une autre personne accède au code, ce que fait une fonction est immédiatement visualisable.
- Cela n'empêche pas par ailleurs de commenter son code.

Exemple :

Bonne habitude

```
def prem(mot) :
    """Prend une chaîne en entrée et en renvoie la 1e lettre"""
    premL = mot[0]      # extrait la 1e lettre
    return premL        # renvoie cette lettre
```

Organisation d'un programme :

Organisation d'un programme :

- On réalise en premier les **importations de bibliothèques** nécessaires (si besoin)

Organisation d'un programme :

- On réalise en premier les **importations de bibliothèques** nécessaires (si besoin)
- On définit ensuite les **variables "globales"** (accessibles à tout le programme)

Organisation d'un programme :

- On réalise en premier les **importations de bibliothèques** nécessaires (si besoin)
- On définit ensuite les **variables "globales"** (accessibles à tout le programme)
- On définit ensuite les **fonctions** utiles au programme.

Organisation d'un programme :

- On réalise en premier les **importations de bibliothèques** nécessaires (si besoin)
- On définit ensuite les **variables "globales"** (accessibles à tout le programme)
- On définit ensuite les **fonctions** utiles au programme.
- On place en dernier le **programme principal**

- Une fonction peut s'appeler elle-même, dans ce cas on la dit **récursive**.

- Une fonction peut s'appeler elle-même, dans ce cas on la dit **récursive**.
- Toute fonction qui n'est pas récursive, est **itérative**.

- Une fonction peut s'appeler elle-même, dans ce cas on la dit **récursive**.
- Toute fonction qui n'est pas récursive, est **itérative**.

On définit une fonction qui calcule le produit $1 \times 2 \times ... \times n$

- Une fonction peut s'appeler elle-même, dans ce cas on la dit **récursive**.
- Toute fonction qui n'est pas récursive, est **itérative**.

On définit une fonction qui calcule le produit $1 \times 2 \times ... \times n$

Fonction récursive

```
def prod1(n) :
    if n == 1 :
        return 1
    else :
        return n×prod1(n − 1)
```

Explications :

Explications :

Imaginons que l'on calcule **prod1(3)**.

Explications :

Imaginons que l'on calcule **prod1(3)**.

prod(3) = 3 x prod(2)

Explications :

Imaginons que l'on calcule **prod1(3)**.

prod(3) = 3 x prod(2)

= 3 x 2 x prod(1)

Explications :

Imaginons que l'on calcule **prod1(3)**.

prod(3) = 3 x prod(2)
= 3 x 2 x prod(1)
= 3 x 2 x 1

Explications :

Imaginons que l'on calcule **prod1(3)**.

prod(3) = 3 x prod(2)
= 3 x 2 x prod(1)
= 3 x 2 x 1
= 6

Explications :

Imaginons que l'on calcule **prod1(3)**.

prod(3) = 3 x prod(2)
= 3 x 2 x prod(1)
= 3 x 2 x 1
= 6

Python entretient une **pile** pour stocker les résultats intermédiaires.

Une fonction récursive possède toujours un "cas de base". Ici on a :

Une fonction récursive possède toujours un "cas de base". Ici on a :

```
if n == 0 :
    return 1
```

Une fonction récursive possède toujours un "cas de base". Ici on a :

```
if n == 0 :
    return 1
```

et un cas où la fonction fait appel à elle-même en opérant souvent une incrémentation ou une décrémentation sur une variable.

Une fonction récursive possède toujours un "cas de base". Ici on a :

```
if n == 0 :
    return 1
```

et un cas où la fonction fait appel à elle-même en opérant souvent une incrémentation ou une décrémentation sur une variable.

```
else :
    return n x prod(n-1)
```

La fonction itérative pour le même besoin peut s'écrire ainsi :

La fonction itérative pour le même besoin peut s'écrire ainsi :

Fonction itérative

```
def prod2(n) :
    result = 1
    for i in range(1, n+1) :
        result *= i
    return result
```

La fonction itérative pour le même besoin peut s'écrire ainsi :

Fonction itérative

```
def prod2(n) :
    result = 1
    for i in range(1, n+1) :
        result *= i
    return result
```

Parfois, définir une fonction récursive est plus simple que de la définir de façon itérative. Cela dépend du problème à traiter.

Les listes constituent le terrain de prédilection de *Python*.

Les listes constituent le terrain de prédilection de *Python*.

Définition

Une liste est un ensemble d'éléments ordonnés, non forcément distincts, modifiables et hétérogènes.

Les listes constituent le terrain de prédilection de *Python*.

Définition

Une liste est un ensemble d'éléments ordonnés, non forcément distincts, modifiables et hétérogènes.

- Une liste s'affiche entre crochets [et]

Les listes constituent le terrain de prédilection de *Python*.

Définition

Une liste est un ensemble d'éléments ordonnés, non forcément distincts, modifiables et hétérogènes.

- Une liste s'affiche entre crochets [et]
- elle peut contenir de nombres, des chaînes, des listes et tout objet défini dans un programme.

Création d'une liste :

Création d'une liste : différente façon de créer une liste.

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
```

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
```

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
```

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
```

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
l5 = list("lapin")      # renvoie ["l", "a", "p", "i", "n"]
```

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
l5 = list("lapin")      # renvoie ["l", "a", "p", "i", "n"]
```

Remarques :

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
l5 = list("lapin")      # renvoie ["l", "a", "p", "i", "n"]
```

Remarques :

- L'ordre des éléments est important.

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
l5 = list("lapin")      # renvoie ["l", "a", "p", "i", "n"]
```

Remarques :

- L'ordre des éléments est important.
- Les listes ainsi définies sont modifiables.

Création d'une liste : différente façon de créer une liste.

création

```
l1 = []      # liste vide que l'on remplira ensuite...
l2 = list()      # autre façon...
l3 = [4, "lapin", "e", 7, [2, 5, "a"], 3]  # on écrit les éléments
l4 = list(range(5))      # renvoie [0, 1, 2, 3, 4]
l5 = list("lapin")      # renvoie ["l", "a", "p", "i", "n"]
```

Remarques :

- L'ordre des éléments est important.
- Les listes ainsi définies sont modifiables.
- Les éléments sont séparés par une virgule.

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

slicing

maliste = [5, "z", "r", 7, 1]

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

slicing

```
maliste = [5, "z", "r", 7, 1]
maliste[1]      # fait référence à "z"
```

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

slicing

```
maliste = [5, "z", "r", 7, 1]
maliste[1]      # fait référence à "z"
maliste[1 :3]      # fait référence à ["z","r"]
```

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

slicing

```
maliste = [5, "z", "r", 7, 1]
maliste[1]      # fait référence à "z"
maliste[1 :3]     # fait référence à ["z","r"]
maliste[ :2]     # fait référence à [5, "z"]
```

- Chaque élément d'une liste est repéré par son indice, de 0 à la longueur de la liste moins 1.

liste =	[5,	"w",	"x",	2]
rangs	0	1	2	3

- Comme pour les chaînes, on peut faire référence à un élément ou une plage d'éléments par le **slicing** (crochets et 2 points)

slicing

```
maliste = [5, "z", "r", 7, 1]
maliste[1]      # fait référence à "z"
maliste[1 :3]     # fait référence à ["z","r"]
maliste[ :2]    # fait référence à [5, "z"]
maliste[2 :]    # fait référence à ["r", 7, 1]
```

Rappels sur le slicing :

Rappels sur le slicing :

- La syntaxe **liste[debut :fin :pas]** signifie :

Rappels sur le slicing :

- La syntaxe **liste[debut :fin :pas]** signifie :
  - de l'élément d'indice **debut**

Rappels sur le slicing :

- La syntaxe **liste[debut :fin :pas]** signifie :
  - de l'élément d'indice **debut**
  - jusqu'à l'élément qui précède celui d'indice **fin**

Rappels sur le slicing :

- La syntaxe **liste[debut :fin :pas]** signifie :
  - de l'élément d'indice **debut**
  - jusqu'à l'élément qui précède celui d'indice **fin**
  - par pas de **pas**
- Quand il n'y a rien devant les 2 points (:), cela signifie que l'on accède aux éléments depuis le début de la liste. Exemple : liste[:5]

Rappels sur le slicing :

- La syntaxe **liste[debut :fin :pas]** signifie :
  - de l'élément d'indice **debut**
  - jusqu'à l'élément qui précède celui d'indice **fin**
  - par pas de **pas**
- Quand il n'y a rien devant les 2 points (:), cela signifie que l'on accède aux éléments depuis le début de la liste. Exemple : liste[:5]
- Quand il n'y a rien après les 2 points (:), cela signifie que l'on accède aux éléments jusqu'à la fin de la liste. Exemple : liste[5 :]

On peut modifier un ou plusieurs éléments d'une liste en y accédant par leur indice.

On peut modifier un ou plusieurs éléments d'une liste en y accédant par leur indice.

adressage d'éléments

maliste = [2, 4, 6, 8, 10]

On peut modifier un ou plusieurs éléments d'une liste en y accédant par leur indice.

adressage d'éléments

maliste = [2, 4, 6, 8, 10]
maliste[0] = "Ha !"

On peut modifier un ou plusieurs éléments d'une liste en y accédant par leur indice.

adressage d'éléments

```
maliste = [2, 4, 6, 8, 10]
maliste[0] = "Ha !"
maliste      # renvoie ["Ha !", 4, 6, 8, 10]
```

On peut modifier un ou plusieurs éléments d'une liste en y accédant par leur indice.

adressage d'éléments

```
maliste = [2, 4, 6, 8, 10]
maliste[0] = "Ha !"
maliste      # renvoie ["Ha !", 4, 6, 8, 10]
```

Quand on essaie d'accéder à une liste par un indice non défini, on a le message d'erreur : list index out of range

On peut :

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

ajouter un itérable à la fin d'une liste

```
maliste.extend("Oh !")      # ajoute ["O","h"," !"] à la fin
```

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

ajouter un itérable à la fin d'une liste

```
maliste.extend("Oh !")      # ajoute ["O","h"," !"] à la fin
maliste.extend(autreliste)      # ajoute autreliste à la suite
```

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

ajouter un itérable à la fin d'une liste

```
maliste.extend("Oh !")      # ajoute ["O","h"," !"] à la fin
maliste.extend(autreliste)      # ajoute autreliste à la suite
```

(une chaîne, une liste, ... sont des itérables)

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

ajouter un itérable à la fin d'une liste

```
maliste.extend("Oh !")      # ajoute ["O","h"," !"] à la fin
maliste.extend(autreliste)      # ajoute autreliste à la suite
```

(une chaîne, une liste, ... sont des itérables)

insérer un élément à la position indiquée

```
maliste.insert(2,"Hi")      # insére "Hi" à la position d'indice 2
```

On peut :

ajouter un élément à la fin d'une liste

```
maliste.append(6)      # ajoute 6 à la fin
```

ajouter un itérable à la fin d'une liste

```
maliste.extend("Oh !")      # ajoute ["O","h"," !"] à la fin
maliste.extend(autreliste)      # ajoute autreliste à la suite
```

(une chaîne, une liste, ... sont des itérables)

insérer un élément à la position indiquée

```
maliste.insert(2,"Hi")      # insére "Hi" à la position d'indice 2
```

Les autres éléments (d'indice 2 à ...) sont repoussés à droite.

supprimer la 1^e occurrence d'un élément

```
maliste.remove("Hi")      # supprime "Hi" de la liste
```

supprimer la 1^e occurrence d'un élément

```
maliste.remove("Hi")      # supprime "Hi" de la liste
```

supprimer un élément à la position indiquée : renvoie l'élément

```
maliste.pop(2)      # supprime l'élément d'indice 2
```

supprimer la 1^e occurrence d'un élément

```
maliste.remove("Hi")      # supprime "Hi" de la liste
```

supprimer un élément à la position indiquée : renvoie l'élément

```
maliste.pop(2)      # supprime l'élément d'indice 2
```

Par défaut, si aucune position n'est indiquée, **supprime et renvoie le dernier élément**.

supprimer la 1^e occurrence d'un élément

```
maliste.remove("Hi")      # supprime "Hi" de la liste
```

supprimer un élément à la position indiquée : renvoie l'élément

```
maliste.pop(2)      # supprime l'élément d'indice 2
```

Par défaut, si aucune position n'est indiquée, **supprime et renvoie le dernier élément**.

Exemple

```
maliste = [1, 7, 5]
ret = maliste.pop()      # ret = 5 et maliste = [1, 7]
```

supprimer la 1^e occurrence d'un élément

```
maliste.remove("Hi")      # supprime "Hi" de la liste
```

supprimer un élément à la position indiquée : renvoie l'élément

```
maliste.pop(2)      # supprime l'élément d'indice 2
```

Par défaut, si aucune position n'est indiquée, **supprime et renvoie le dernier élément**.

Exemple

```
maliste = [1, 7, 5]
ret = maliste.pop()      # ret = 5 et maliste = [1, 7]
```

supprimer tous les éléments

```
maliste.clear()      # La liste est vide maintenant
```

trouver l'index d'un élément

```
maliste.index(6)      # renvoie l'index de 6.
```

trouver l'index d'un élément

```
maliste.index(6)      # renvoie l'index de 6.
```

On peut spécifier la plage de recherche par **index(6,debut,fin)**

trouver l'index d'un élément

```
maliste.index(6)      # renvoie l'index de 6.
```

On peut spécifier la plage de recherche par **index(6,debut,fin)**

trier une liste : la liste est modifiée

```
maliste.sort()      # les éléments doivent être de même nature
```

trouver l'index d'un élément

```
maliste.index(6)      # renvoie l'index de 6.
```

On peut spécifier la plage de recherche par **index(6,debut,fin)**

trier une liste : la liste est modifiée

```
maliste.sort()      # les éléments doivent être de même nature
```

compter le nombre d'occurrence d'un élément

```
maliste.count(6)# donne le nombre de 6 contenu dans maliste
```

trouver l'index d'un élément

```
maliste.index(6)      # renvoie l'index de 6.
```

On peut spécifier la plage de recherche par **index(6,debut,fin)**

trier une liste : la liste est modifiée

```
maliste.sort()      # les éléments doivent être de même nature
```

compter le nombre d'occurrence d'un élément

```
maliste.count(6)# donne le nombre de 6 contenu dans maliste
```

renverser une liste : modifie la liste

```
maliste.reverse()      # l'ordre des éléments est renversé
```

Exercice : deviner ce que l'on obtient !

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
list1.sort().reverse()
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
list1.sort().reverse()      # list1 = [6, 5, 4, 3, 2]
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
list1.sort().reverse()      # list1 = [6, 5, 4, 3, 2]
a, b = list1.index(4), list1.count(4)
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
list1.sort().reverse()      # list1 = [6, 5, 4, 3, 2]
a, b = list1.index(4), list1.count(4)      # a = 2 et b = 1
```

Exercice : deviner ce que l'on obtient !

```
list1, list2 = ["z", 6], [3, 2, "y"]
list1.append(4)      # list1 = ["z", 6, 4]
list1.reverse()      # list1 = [4, 6, "z"]
a = list1.pop()      # list1 = [4, 6] et a = "z"
list1.extend(list2)      # list1 = [4, 6, 3, 2, "y"]
list1.insert(1, 5)      # list1 = [4, 5, 6, 3, 2, "y"]
list1.remove("y")      # list1 = [4, 5, 6, 3, 2]
list1.sort().reverse()      # list1 = [6, 5, 4, 3, 2]
a, b = list1.index(4), list1.count(4)      # a = 2 et b = 1
```

Il est très important de savoir correctement manipuler les listes avec *Python* !

récupérer le tri d'une liste

uneliste = [2, 7, 1]

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer la longueur d'une liste

```
long = len(uneliste)
```

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer la longueur d'une liste

```
long = len(uneliste)      # long = 3
```

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer la longueur d'une liste

```
long = len(uneliste)      # long = 3
```

passer d'un string à une liste et réciproquement

```
p1 = "il va en cours"
```

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer la longueur d'une liste

```
long = len(uneliste)      # long = 3
```

passer d'un string à une liste et réciproquement

```
p1 = "il va en cours"
liste = p1.split()      # liste = ["il","va","en","cours"],
```

découpe selon les espaces par défaut

récupérer le tri d'une liste

```
uneliste = [2, 7, 1]
L1 = sorted(uneliste)      # L1 = [1, 2, 7] uneliste = [2, 7, 1]
```

récupérer la longueur d'une liste

```
long = len(uneliste)      # long = 3
```

passer d'un string à une liste et réciproquement

```
p1 = "il va en cours"
liste = p1.split()      # liste = ["il","va","en","cours"],
```

découpe selon les espaces par défaut

```
p2 = "*".join(liste)      # p2 = "il*va*en*cours"
```

boucler sur une liste

uneliste = [2, 7, 1]

boucler sur une liste

```
uneliste = [2, 7, 1]
for elt in uneliste :
    print(elt) # Affiche en colonne 2, 7 puis 1
```

boucler sur une liste

```
uneliste = [2, 7, 1]
for elt in uneliste :
    print(elt) # Affiche en colonne 2, 7 puis 1
```

boucler sur une liste en récupérant l'indice de boucle

```
uneliste = ["a", "b", "c"]
```

boucler sur une liste

```
uneliste = [2, 7, 1]
for elt in uneliste :
    print(elt) # Affiche en colonne 2, 7 puis 1
```

boucler sur une liste en récupérant l'indice de boucle

```
uneliste = ["a", "b", "c"]
for ind, elt in enumerate(uneliste) :
    print(ind, elt) # Affiche en colonne
0 "a"
1 "b"
2 "c"
```

combiner 2 listes

liste1, liste2 = [2, 7, 1], ["s", "n", "m"]

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
```

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
for e in listcomb :
    print(e) # Affiche en colonne (2, 's'), (7, 'n') puis (1, 'm')
```

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
for e in listcomb :
    print(e) # Affiche en colonne (2, 's'), (7, 'n') puis (1, 'm')
```

On obtient ici des tuples (voir après...)

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
for e in listcomb :
    print(e) # Affiche en colonne (2, 's'), (7, 'n') puis (1, 'm')
```

On obtient ici des tuples (voir après...)

décombiner 2 listes

```
listc = [['a', 2], ['b', 5], ['c', 1]]
```

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
for e in listcomb :
    print(e) # Affiche en colonne (2, 's'), (7, 'n') puis (1, 'm')
```

On obtient ici des tuples (voir après...)

décombiner 2 listes

```
listc = [['a', 2], ['b', 5], ['c', 1]]
for e in zip(*listc) :
    print(e) # Affiche ('a','b','c') puis (2,5,1)
```

combiner 2 listes

```
liste1, liste2 = [2, 7, 1], ["s", "n", "m"]
listcomb = zip(liste1, liste2)
for e in listcomb :
    print(e) # Affiche en colonne (2, 's'), (7, 'n') puis (1, 'm')
```

On obtient ici des tuples (voir après...)

décombiner 2 listes

```
listc = [['a', 2], ['b', 5], ['c', 1]]
for e in zip(*listc) :
    print(e) # Affiche ('a','b','c') puis (2,5,1)
```

On obtient encore des tuples.

détruire 1 ou plusieurs éléments par leurs indices

listA = ['a', 'b', 'c']

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]      # elt1 = 7
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]      # elt1 = 7
elt2 = listA[1][0]
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]      # elt1 = 7
elt2 = listA[1][0]      # elt2 = 'c'
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]      # elt1 = 7
elt2 = listA[1][0]      # elt2 = 'c'
listA[1][1] = 0
```

détruire 1 ou plusieurs éléments par leurs indices

```
listA = ['a', 'b', 'c']
del listA[1]      # listA=['a','c'], on peut utiliser la notation ( :)
```

liste de listes : comment y accéder ?

```
listA = [['a', 3, 7], ['c', 's', 4, 1]]
elt1 = listA[0][2]      # elt1 = 7
elt2 = listA[1][0]      # elt2 = 'c'
listA[1][1] = 0      # listA = [['a', 3, 7], ['c', 0, 4, 1]]
```

Définition

Un tuple est un ensemble d'éléments ordonnés, non forcément distincts, non modifiables et qui peuvent être hétérogènes.

Définition

Un tuple est un ensemble d'éléments ordonnés, non forcément distincts, non modifiables et qui peuvent être hétérogènes.

- Les tuples sont non modifiables au niveau de la structure, mais si un élément est une liste, les éléments de cette liste sont modifiables à condition que l'élément reste une liste.

Définition

Un tuple est un ensemble d'éléments ordonnés, non forcément distincts, non modifiables et qui peuvent être hétérogènes.

- Les tuples sont non modifiables au niveau de la structure, mais si un élément est une liste, les éléments de cette liste sont modifiables à condition que l'élément reste une liste.
- Les tuples sont créés à l'aide de parenthèses, mais pas nécessairement.

Définition

Un tuple est un ensemble d'éléments ordonnés, non forcément distincts, non modifiables et qui peuvent être hétérogènes.

- Les tuples sont non modifiables au niveau de la structure, mais si un élément est une liste, les éléments de cette liste sont modifiables à condition que l'élément reste une liste.
- Les tuples sont créés à l'aide de parenthèses, mais pas nécessairement.
- Les éléments sont séparés par une virgule.

Définir un tuple

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
```

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
```

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
```

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
t3 = 5,               # tuple à un élément, ne pas oublier la virgule
```

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
t3 = 5,               # tuple à un élément, ne pas oublier la virgule
t3 = (5,)             # préférable à ce qui précède
```

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
t3 = 5,               # tuple à un élément, ne pas oublier la virgule
t3 = (5,)             # préférable à ce qui précède
```

Avantages des tuples :

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
t3 = 5,               # tuple à un élément, ne pas oublier la virgule
t3 = (5,)             # préférable à ce qui précède
```

Avantages des tuples :

- les tuples sont plus rapidement itérables que les listes.

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'        # possible, mais à éviter
t2 = ()               # tuple vide, pas très utile...
t3 = 5,               # tuple à un élément, ne pas oublier la virgule
t3 = (5,)             # préférable à ce qui précède
```

Avantages des tuples :

- les tuples sont plus rapidement itérables que les listes.
- Ils permettent les affectations multiples (ou parallèles)

Définir un tuple

```
t1 = (2, 3, 'a')      # préférable à ce qui suit
t1 = 2, 3, 'a'      # possible, mais à éviter
t2 = ()      # tuple vide, pas très utile...
t3 = 5,      # tuple à un élément, ne pas oublier la virgule
t3 = (5,)      # préférable à ce qui précède
```

Avantages des tuples :

- les tuples sont plus rapidement itérables que les listes.
- Ils permettent les affectations multiples (ou parallèles)
- Le tuple permet de renvoyer plusieurs valeurs lors du **return** d'une fonction.

On peut quand même opérer sur un tuple, voici quelques exemples :

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t      # renvoie True
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t      # renvoie True
t[2][0] = 22
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t      # renvoie True
t[2][0] = 22      # t = (5, 'la', [22, 'z'])
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t      # renvoie True
t[2][0] = 22      # t = (5, 'la', [22, 'z'])
t[0] = 1
```

On peut quand même opérer sur un tuple, voici quelques exemples :

Usage d'un tuple

```
t = (5, 'la', [2, 'z'])
long = len(t)      # long = 3
5 in t      # renvoie True
t[2][0] = 22      # t = (5, 'la', [22, 'z'])
t[0] = 1      # renvoie un message d'erreur !
```

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.
- Les couples < clé : valeur > sont séparés par une virgule

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.
- Les couples < clé : valeur > sont séparés par une virgule.
- Les clés, comme les valeurs, peuvent être de tout type et sont séparés par 2 points (:).

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.
- Les couples < clé : valeur > sont séparés par une virgule.
- Les clés, comme les valeurs, peuvent être de tout type et sont séparés par 2 points (:).

Exemples de dictionnaire

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.
- Les couples < clé : valeur > sont séparés par une virgule.
- Les clés, comme les valeurs, peuvent être de tout type et sont séparés par 2 points (:).

Exemples de dictionnaire

dico1 = {'coq' : 6, 'lapin' : 4, 'cochon' : 2}

Définition

Un dictionnaire est un tableau associatif modifiable, c'est-à-dire qu'il se comporte un peu comme une liste non ordonnée d'éléments < clé : valeur >. Les clés sont forcément uniques.

- Un dictionnaire se note entre accolades.
- Les couples < clé : valeur > sont séparés par une virgule.
- Les clés, comme les valeurs, peuvent être de tout type et sont séparés par 2 points (:).

Exemples de dictionnaire

dico1 = {'coq' : 6, 'lapin' : 4, 'cochon' : 2}
dico2 = {1 : 'chien', [3, 5] : 't', 'pm' : 7}

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
```

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
```

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
dic['e'] = 8      # 2e clé :valeur
```

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
dic['e'] = 8      # 2e clé :valeur
print(dic)     # affiche dic = {'a' : 5, 'e' : 8}
```

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
dic['e'] = 8      # 2e clé :valeur
print(dic)     # affiche dic = {'a' : 5, 'e' : 8}
```

On peut aussi créer un dictionnaire à partir d'une liste de tuples, ou d'un tuple de listes ou liste de listes...

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
dic['e'] = 8      # 2e clé :valeur
print(dic)      # affiche dic = {'a' : 5, 'e' : 8}
```

On peut aussi créer un dictionnaire à partir d'une liste de tuples, ou d'un tuple de listes ou liste de listes...

liste, tuple –> dictionnaire

```
liste = [['a',2], ['e',7]]      # on utilise le mot-clé dict()
```

On peut créer un dictionnaire vide puis alimenter son contenu avec des éléments **clé : valeur** ensuite.

Exemple :

```
dic = {}      # dictionnaire vide
dic['a'] = 5      # la clé est entre crochets
dic['e'] = 8      # 2e clé :valeur
print(dic)      # affiche dic = {'a' : 5, 'e' : 8}
```

On peut aussi créer un dictionnaire à partir d'une liste de tuples, ou d'un tuple de listes ou liste de listes...

liste, tuple –> dictionnaire

```
liste = [['a',2], ['e',7]]      # on utilise le mot-clé dict()
dico = dict(liste)      # donne dico = {'a' : 2, 'e' : 7}
```

Dictionnaire de départ

dic = {'a' :4, 'e' :8, 'i' :2}

Dictionnaire de départ

dic = {'a' :4, 'e' :8, 'i' :2}

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
dic.values()
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
dic.values()    # génère un itérable sur les valeurs
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
dic.values()    # génère un itérable sur les valeurs
dic.items()
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
dic.values()    # génère un itérable sur les valeurs
dic.items()     # génère un double itérable sur les couples
```

Dictionnaire de départ

```
dic = {'a' :4, 'e' :8, 'i' :2}
```

On peut obtenir la liste des clés, ou des valeurs (qui sont alors des itérables) ou des couples (clé- valeur).

clés, valeurs, et les deux

```
dic.keys()      # génère un itérable sur les clés
dic.values()    # génère un itérable sur les valeurs
dic.items()     # génère un double itérable sur les couples
```

Utilisation

dic = {'a' :4, 'e' :8, 'i' :2}

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)
```

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)      # renvoie
```

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)      # renvoie
```

a
e
i

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)      # renvoie
a
e
i
for cle, val in dic.items() :
    print(cle, val)
```

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)      # renvoie
```

a
e
i

```
for cle, val in dic.items() :
    print(cle, val)      # renvoie
```

Utilisation

```
dic = {'a' :4, 'e' :8, 'i' :2}
for cle in dic.keys() :
    print(cle)      # renvoie
```

```
a
e
i
```

```
for cle, val in dic.items() :
    print(cle, val)      # renvoie
```

```
a 4
e 8
i 2
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

tester l'existence d'une clé

```
'e' in dic
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

tester l'existence d'une clé

```
'e' in dic      # renvoie True
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

tester l'existence d'une clé

```
'e' in dic      # renvoie True
```

supprimer une clé et récupérer sa valeur

```
ret = dic.pop('e')
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

tester l'existence d'une clé

```
'e' in dic      # renvoie True
```

supprimer une clé et récupérer sa valeur

```
ret = dic.pop('e')      # ret = 8
```

Avec : **dic = {'a' :4, 'e' :8, 'i' :2}**

changer une valeur

```
dic['a'] = 5      # dic = {'a' :5, 'e' :8, 'i' :2}
```

détruire une clé

```
del dic['a']      # dic = {'e' :8, 'i' :2}
```

tester l'existence d'une clé

```
'e' in dic      # renvoie True
```

supprimer une clé et récupérer sa valeur

```
ret = dic.pop('e')      # ret = 8
```

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
for elt in ['a', 'i', 't', 'e'] :
    print("valeur de", elt, dic.get(elt, "consonne !"))
```

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
    for elt in ['a', 'i', 't', 'e'] :
        print("valeur de", elt, dic.get(elt, "consonne !"))
```

a 4

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
    for elt in ['a', 'i', 't', 'e'] :
        print("valeur de", elt, dic.get(elt, "consonne !"))
```

```
a 4
i 2
```

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
    for elt in ['a', 'i', 't', 'e'] :
        print("valeur de", elt, dic.get(elt, "consonne !"))
```

```
a 4
i 2
t consonne !
```

Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
    for elt in ['a', 'i', 't', 'e'] :
        print("valeur de", elt, dic.get(elt, "consonne !"))
```

```
a 4
i 2
t consonne !
e 8
```

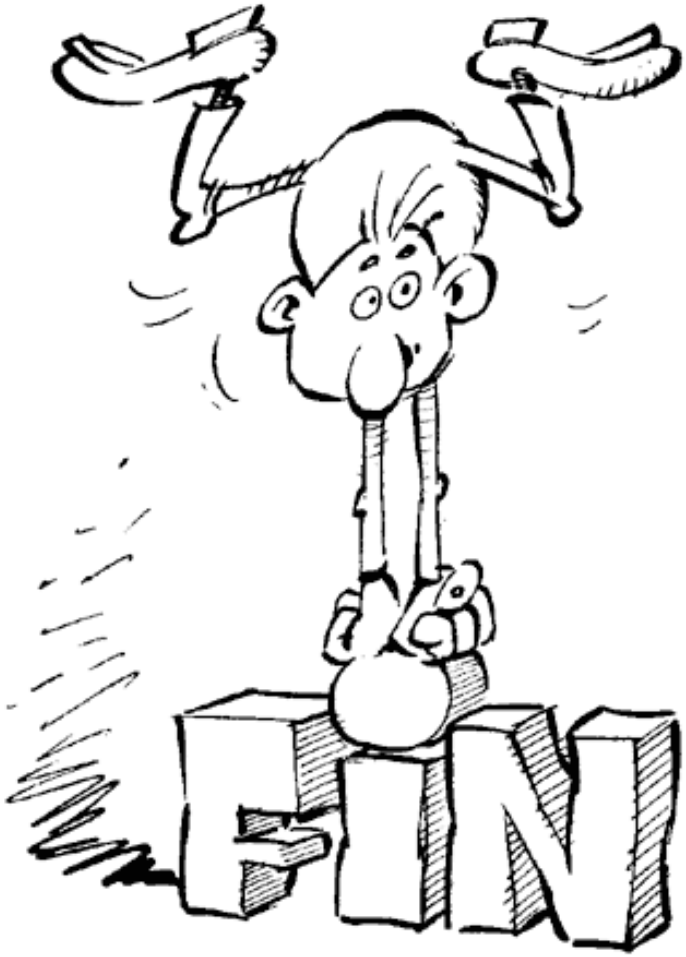
Une dernière méthode bien utile est **get()** qui attend 2 paramètres : une clé et une valeur de retour par défaut.

méthode get() avec dic = {'a' :4, 'e' :8, 'i' :2}

```
    for elt in ['a', 'i', 't', 'e'] :
        print("valeur de", elt, dic.get(elt, "consonne !"))
```

```
a 4
i 2
t consonne !
e 8
```

Les dictionnaires sont vite indispensables pour mettre en œuvre des programmes plus performants.


FIN