

Course: Python

Ali ZAINOUL <contact@clearcode.fr>

Crystal Clear Code
June 16, 2025



1 Les Fonctions

- Définition et appel d'une fonction
- Nombre variable de paramètres dans une fonction
- Fonctions avec des paramètres clé/valeur
- Les générateurs - Instruction yield
- Notion de scope (Espace de noms) - portée des variables
- Scopes pré-définis (`__builtins__`)
- Fonction `dir()`
- Retourner des valeurs, instruction `return`
- Fonctions génériques (duck typing)
- Valeurs par défaut
- Passage par étiquette
- Nombre d'arguments arbitraire (`*args`, `**kwargs`)
- Fonctions anonymes (`lambda`)
- Fonctions `eval()`, `exec()`, `map()` et `filter()`

2 Exceptions

- Principe de fonctionnement
- Exceptions pré-définies et arbre d'héritage
- Instructions `try ... except ... else ... finally`

- Propagation des exceptions
- Déclenchement d'exceptions
- Définition d'une exception

3 Manipulation des fichiers

- Fonction `open()` et méthode `close()`
- Méthodes `readline()` et `readlines()` Objet itérable
- Instruction `with` avec les fichiers
- Méthodes `read()` et `write()`
- Méthodes `tell()` et `seek()`
- Méthode `writelines()`
- Modules complémentaires : `struct`, `csv`, `json`, `xml`
 - Module `struct`
 - Module `csv`
 - Module `json`
 - Module `xml`
- Sérialisation avec le module `pickle`
- Sérialisation avec le module `shelve`

**** Les Fonctions ****

Fonctions en Python

- On peut vouloir définir une fonction afin de l'utiliser dans un programme, la syntaxe générale pour déclarer et définir une fonction est la suivante :

```
1 def functionName(arguments):  
2     # function implementation
```

- Une fonction sera toujours définie avec le mot-clé `def`.
- Veuillez noter l'importance de l'indentation dans l'implémentation de la fonction.

Fonctions en Python : exemple 1

Définir une fonction sans paramètres et sans retour :

```
1 # Definition of the utility function printLine
2 def printLine():
3     print("-----")
4
5 # Testing our function
6 printLine()
```

Fonctions en Python : exemple 2

Définir une fonction qui prend un nombre en tant que paramètre et renvoie son carré :

```
1 # Definition of our square function
2 def Square(n):
3     return n**2
4
5 # Testing our function
6 Square(4)
```

Notez l'utilisation de l'opérateur `**` pour calculer la puissance d'un nombre.

Notion de récursivité : fonctions récursives

Définitions :

- Un algorithme est appelé **récursif** s'il est défini par lui-même.
- De la même manière, une fonction est appelée **récursive** si elle fait référence à elle-même.

Fonctions en Python : exemple 3

Définir une fonction récursive avec un paramètre et sans retour (le retour renvoie vide) :

```
1 # Definition of our recursive function Count
2 def Count(n):
3     print(n)
4     if n == 0 :
5         print("Terminated.")
6         return
7     Count(n-1)
8
9 # Testing our function
10 Count(10)
```

Nombre variable de paramètres dans une fonction

- Python vous permet de créer des fonctions avec un nombre variable de paramètres. Exemple: fonction `print()`.
- La syntaxe générale de telles fonctions est :

```
1 # Definition of the function nameFunction with variable arguments
2 def nameFunction(*args):
3     # instructions
```

```
1 # Example:
2 def print_foo(*args):
3     for arg in args:
4         print(arg)
5
6 # Test our function
7 print_foo("Hello", 1, 2, 3, "World")
8 ''' # Output:
9 Hello
10 1
11 2
12 3
13 World
14 '''
```

Fonctions avec des paramètres clé/valeur

- Les paramètres peuvent également être gérés sous forme de clés/valeurs pour peupler un dictionnaire :
- La syntaxe générale de telles fonctions est :

```
1 # Definition of the function nameFunction with keyword arguments
2 def nameFunction(**kwargs):
3     # instructions
```

```
1 # Example:
2 def printDict(**kwargs):
3     for (k, v) in kwargs.items():
4         print(k, v)
5
6 # Test our function
7 printDict(Day_1="Monday", Day_2="Tuesday")
8 ''' # Output:
9 Day_1 Monday
10 Day_2 Tuesday
11 '''
```

Générateurs

- Un générateur est implémenté avec le mot-clé `yield`, il permet de créer un flux de données. Une sorte de réservoir de données ; qui peut être appelé morceau par morceau par `next()`. Other doc.

```
1 # Define a generator function
2 def Generator():
3     for i in range(10, 0, -1):
4         yield i
5
6 # Create an instance of the generator
7 gen = Generator()
8
9 # Print the results
10 print(next(gen)) # Output : 10
11 print(next(gen)) # Output : 9
12 # ...
13 print(next(gen)) # Output : 1
14 # print(next(gen)) # Error, because we stop at 1
15 '''
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
19 '''
```

Espace de noms (Scopes)

- En Python, chaque fonction crée son propre espace de noms local lorsqu'elle est appelée. Les variables déclarées à l'intérieur de cette fonction sont accessibles uniquement à l'intérieur de cette fonction, à moins qu'elles ne soient déclarées comme globales.
- L'espace de noms global est accessible depuis n'importe où dans le script Python et peut être modifié à l'aide du mot-clé `global`.

Exemple d'espace de noms

```
1 # Exemple d'espace de noms local et global en Python
2 x = 10 # Variable globale
3
4 def fonction():
5     y = 20 # Variable locale à la fonction
6     print("Variables locales:", locals())
7
8 fonction()
9 print("Variables globales:", globals())
```

Notion de Scope

- Le **scope** (**portée**) d'une variable en Python fait référence à la région du code où la variable peut être utilisée.
- Python utilise une règle de portée lexicale, ce qui signifie que la portée d'une variable est déterminée par son emplacement dans le code source.
- Le concept de portée concerne la visibilité des variables dans le code.

Notion de scope - Exemple complet (1/2)

```
1 # Scope 1: Global Scope
2 global_variable = "I am global"
3
4 def outer_function():
5     # Scope 2: Outer Function Scope
6     outer_variable = "I am in the outer function"
7
8     def inner_function():
9         # Scope 3: Inner Function Scope
10        inner_variable = "I am in the inner function"
11        print("Inner variable:", inner_variable)
12        print("Outer variable inside the inner function:",
13              outer_variable)
14        print("Global variable inside the inner function:",
15              global_variable)
```


Notion de scope - Exemple complet (2/2)

```
28     inner_function()
29     print("Outer variable:", outer_variable)
30     # Uncommenting the line below would result in an error
31     # print("Inner variable outside the inner function:",
        inner_variable)
32
33 outer_function()
34 print("Global variable:", global_variable)
35 # Uncommenting the line below would result in an error
36 # print("Outer variable outside the outer function:",
    outer_variable)
```

Portée de Variables Globales

- Les variables globales sont déclarées en dehors de toute fonction et peuvent être utilisées à travers tout le programme.

```
1 # Example of the notion of global variable
2 global_var = 42
3
4 def print_global():
5     print("Global variable:", global_var)
6
7 print_global() # Displays "Global variable: 42"
8 '''
9 The global_var is accessible inside print_global()'s scope even
10    if print_global() method has no variable named global_var.
11 '''
```

Portée de Variables Locales

- Les variables locales sont déclarées à l'intérieur d'une fonction et sont accessibles uniquement à l'intérieur de cette fonction.

```
1 def example_function():
2     local_variable = "I am local"
3     print(local_variable)
4
5 example_function() # Displays "I am local"
6 # Uncommenting the line below would result in an error
7 # print(local_variable)
```

Différence entre la Portée Locale et Globale

En Python, la portée des variables peut être locale ou globale, ce qui détermine où la variable peut être utilisée ou modifiée.

- **Portée Locale** : Une variable déclarée à l'intérieur d'une fonction a une portée locale. Elle n'est accessible qu'à l'intérieur de cette fonction.
- **Portée Globale** : Une variable déclarée en dehors de toutes les fonctions a une portée globale. Elle peut être utilisée dans l'ensemble du programme.

Il est important de comprendre la différence entre la portée locale et globale pour éviter les erreurs et assurer une utilisation correcte des variables.

Exemple illustrant la différence de portées

```
1 # Declaration of a global variable
2 index = 0
3
4 def modify_index():
5     # Redefining the global variable inside the function
6     modify_index()
7     index = 42
8     print("Value of index inside the function:", index) #
9     Displays: 42
10
11 # Calling the function
12 modify_index()
13
14 # Displaying the value of index outside the function
15 print("Value of index outside the function:", index) # Displays: 0
```

Exemple illustrant la différence de portées (Suite)

- Dans cet exemple, la variable `index` est initialement déclarée en tant que variable globale, et vaut 0.
- Elle est ensuite redéfinie à l'intérieur de la fonction `modify_index`.
- L'appel de la fonction `modify_index` imprime la variable locale `index` qui vaut 42.
- La dernière ligne imprime la valeur de la variable globale `index` qui elle vaut 0.

Mot-clé `global` en Python

- Le mot-clé `global` en Python est utilisé pour déclarer qu'une variable à l'intérieur d'une fonction appartient à l'espace de noms global plutôt qu'à l'espace de noms local. Cela signifie que la variable sera accessible et modifiable à la fois à l'intérieur et à l'extérieur de la fonction.
- L'utilisation du mot-clé `global` permet de modifier une variable définie en dehors de la fonction à l'intérieur de cette fonction, tout en maintenant sa portée globale.

Mot-clé `global` en Python - Exemple pratique

■ Voici un exemple illustrant l'utilisation du mot-clé `global` en Python :

```
1 global_var = 0
2 print("Global Scope *before* calling foo() and bar(): global_var = ", global_var)
3 # Output : Global Scope *before* calling foo(): global_var = 0
4 def foo():
5     global_var = 1
6     print("Inside foo()'s Local Scope: global_var = ", global_var)
7
8 foo()
9 # Output : Inside foo()'s Local Scope: global_var = 1
10 print("Global Scope *after* calling foo(): global_var = ", global_var)
11 # Output : Global Scope *after* calling foo(): global_var = 0
12
13 def bar():
14     global global_var
15     global_var = 2
16     print("Inside bar()'s Local Scope: global_var = ", global_var)
17
18 bar()
19 # Output : Inside bar()'s Local Scope: global_var = 2
20 print("Global Scope *after* calling bar(): global_var = ", global_var)
21 # Output : Global Scope *after* calling bar(): global_var = 2
```


Scopes pré-définis (`__builtins__`)

- En Python, le module `__builtins__` contient les fonctions et les types de données intégrés qui sont disponibles dans tous les espaces de noms.
- Ces fonctions et types de données sont prêts à l'emploi sans avoir besoin d'importer des modules supplémentaires.

Exemple de scopes pré-définis (__builtins__)

```
1 # List of builtin functions __builtins__  
2 print("List of builtin functions __builtins__: ")  
3 print(dir(__builtins__))
```

Fonction `dir()`

- En Python, la fonction `dir()` est utilisée pour obtenir la liste des noms de symboles (variables, fonctions, classes, etc.) définis dans un module ou un objet.
- Elle retourne une liste de chaînes de caractères contenant les noms des symboles.

Exemple de la fonction dir()

```
1 # Usage of dir() function in Python
2 print(" List of symbols and methods of module math: ")
3 import math
4 print(dir(math))
5 list = [1,2,3]
6 print(" List of methods of class list: ")
7 print(dir(list))
```

Fonctions - Retourner des valeurs, instruction `return`

- L'instruction `return` est utilisée dans les fonctions Python pour renvoyer une valeur à l'appelant de la fonction. Lorsque Python rencontre une instruction `return`, il quitte immédiatement la fonction et renvoie l'expression associée à `return` à l'appelant. Cela permet à une fonction de calculer un résultat et de le renvoyer à l'endroit où elle a été appelée.

Exemples - Retourner des valeurs, instruction return

```
1 def add(a, b):  
2     result = a + b  
3     return result  
4  
5 sum = add(3, 5)  
6 print(sum) # Output: 8
```

```
1 def square_and_cube(x):  
2     square = x ** 2  
3     cube = x ** 3  
4     return square, cube  
5  
6 result = square_and_cube(4)  
7 print(result) # Output: (16, 64)
```

```
1 def hello():  
2     print("Hello World!")  
3     return  
4  
5 result = hello()  
6 print(result) # Output: None
```

Le Duck Typing en Python

Définition : Le Duck Typing est un concept en Python qui se concentre sur le comportement d'un objet plutôt que sur son type. Il suit le principe selon lequel "si cela ressemble à un canard, nage comme un canard et fait le cri d'un canard, alors c'est probablement un canard". En Python, cela signifie que le type ou la classe d'un objet est déterminé par ses méthodes et propriétés plutôt que par son héritage explicite ou ses annotations de type.

Spécifications sur le Duck Typing

- Le Duck Typing permet une flexibilité dans les arguments de fonction ou de méthode en fonction du comportement qu'ils présentent plutôt que de leur type explicite.
- Le concept met l'accent sur l'importance du comportement de l'objet plutôt que sur son type formel, favorisant une approche de programmation plus dynamique et ouverte.
- Un exemple de Duck Typing en Python est la fonction intégrée `len()`, qui fonctionne sur n'importe quel objet qui définit une méthode `__len__()`, indépendamment de son type explicite.
- Référence : Real Python - Duck Typing

Examples

```
1 # Example of len() function
2 class CustomList:
3     def __init__(self, items):
4         self.items = items
5
6     def __len__(self):
7         return len(self.items)
8
9 custom_list_instance = CustomList([1, 2, 3, 4, 5])
10 print(len(custom_list_instance)) # Output: 5
```

```
1 class Dog:
2     def sound(self):
3         return "Woof!"
4
5 class Cat:
6     def sound(self):
7         return "Meow!"
8
9 class Duck:
10    def sound(self):
11        return "Quack!"
12
13 def make_sound(animal):
14     return animal.sound()
15
16 # Usage of duck typing
17 dog_instance = Dog()
18 cat_instance = Cat()
19 duck_instance = Duck()
20
21 print(make_sound(dog_instance))    # Output: Woof!
22 print(make_sound(cat_instance))    # Output: Meow!
23 print(make_sound(duck_instance))   # Output: Quack!
```

Fonctions - Valeurs par défaut

- En Python, les fonctions peuvent avoir des paramètres avec des valeurs par défaut. Ces valeurs sont utilisées lorsque l'appelant de la fonction ne fournit pas de valeur pour ces paramètres. Les valeurs par défaut permettent de définir des paramètres optionnels pour les fonctions.

Exemple - Valeurs par défaut

```
1 def greet(name="World"):
2     print("Hello", name)
3
4 greet() # Output: Hello World
5 greet("Alice") # Output: Hello Alice
```

Fonctions - Passage par étiquette

- En Python, les arguments peuvent être passés à une fonction par leur nom, ce qui permet de spécifier explicitement quel argument doit être affecté à quel paramètre. Cela offre une flexibilité dans l'ordre des arguments lors de l'appel d'une fonction.

Exemple - Passage par étiquette

```
1 def greet(first_name, last_name):  
2     print("Hello", first_name, last_name)  
3  
4 greet(last_name="Doe", first_name="John") # Output: Hello John Doe
```

Fonctions - Nombre d'arguments arbitraire (*args, **kwargs)

- En Python, il est possible de définir des fonctions prenant un nombre variable d'arguments. L'utilisation de *args permet de capturer un nombre arbitraire d'arguments positionnels, tandis que **kwargs permet de capturer un nombre arbitraire d'arguments nommés.

Exemple - Nombre d'arguments arbitraire (*args, **kwargs)

```
1 def sum(*args):
2     result = 0
3     for num in args:
4         result += num
5     return result
6
7 print(sum(1, 2, 3, 4)) # Output: 10
```

```
1 def print_info(**kwargs):
2     for key, value in kwargs.items():
3         print(key + ":", value)
4
5 print_info(name="Alice", age=30, city="New York")
6 # Output:
7 # name: Alice
8 # age: 30
9 # city: New York
```


Exemple 1 - Mélange de *args et **kwargs

```
1 def print_info(*args, **kwargs):
2     print("Positional arguments:")
3     for arg in args:
4         print(arg)
5     print("Keyword arguments:")
6     for key, value in kwargs.items():
7         print(key + ":", value)
8
9 print_info("Alice", 30, city="New York", occupation="Engineer")
10 # Output:
11 # Positional arguments:
12 # Alice
13 # 30
14 # Keyword arguments:
15 # city: New York
16 # occupation: Engineer
```

Exemple 2 - Mélange de *args et **kwargs

```
1 def create_person(**kwargs):
2     person = {}
3     for key, value in kwargs.items():
4         person[key] = value
5     return person
6
7 def print_person_info(*args, **kwargs):
8     print("Printing person information:")
9     for person in args:
10        print("Name:", person["name"])
11        print("Age:", person["age"])
12        for key, value in kwargs.items():
13            if value:
14                print(key.capitalize() + ":", person[key])
15        print("\n")
16
17 # Creating persons using **kwargs
18 person1 = create_person(name="Alice", age=30, city="New York", occupation="Engineer")
19 person2 = create_person(name="Bob", age=25, city="Los Angeles", occupation="Artist")
20 person3 = create_person(name="Charlie", age=35, city="Chicago")
21
22 # Printing person information
23 print_person_info(person1, person2, person3)
```

Lambda Functions

- Les fonctions Lambda sont:
 - de petite taille ;
 - anonymes
 - sujettes à une syntaxe plus restrictive ;
 - concises ;
 - peuvent prendre n'importe quel nombre d'arguments.
- On définit une fonction `lambda` avec le mot-clé: **lambda**.
- La syntaxe générale d'une fonction `lambda` est:

```
lambda args : implementation
```

Lambda Functions: examples

```
1 # Example 1: lambda function with 1 parameter
2 f = lambda x : 2 * x
3 print( f(50) )
4 #Output: 100.
```

```
1 # Example 2: lambda function with 2 parameters
2 fun = lambda a, b : a ** b
3 print(fun(7, 2))
4 #Output: 49.
```

Usage of Lambda Functions

- The utility of a Lambda function is when you use it as an anonymous function inside another function.
- A complete example is shown below:

```
1 # Define a multiplier function
2 def multiplier(n):
3     return lambda x: x * n
4
5 # Create instances of the multiplier function
6 Doubler = multiplier(2)
7 Tripler = multiplier(3)
8
9 # Print the results
10 print(Doubler(33))    # Outputs: 66
11 print(Tripler(33))   # Outputs: 99
```

Fonctions `eval()`, `exec()`, `map()` et `filter()`

- Les fonctions `eval()` et `exec()` permettent d'exécuter du code Python dynamiquement à partir d'une chaîne de caractères.
- `eval()` évalue une expression Python et retourne la valeur.
- `exec()` exécute du code Python et ne retourne rien.
- `map()` applique une fonction à chaque élément d'un itérable (comme une liste) et retourne un itérable contenant les résultats.
- `filter()` filtre les éléments d'un itérable en fonction d'une fonction de filtrage et retourne un itérable contenant les éléments filtrés.

Exemples d'utilisation

Exemple avec eval():

```
1 x = 10
2 result = eval('x ** 2 + 5')
3 print(result) # Output: 105
```

Exemple avec exec():

```
1 code = '''for i in range(5): print(i)'''
2 exec(code) # Output: 0 1 2 3 4
```

Exemple avec map():

```
1 numbers = [1, 2, 3, 4, 5]
2 squared = map(lambda x: x ** 2, numbers)
3 print(list(squared)) # Output: [1, 4, 9, 16, 25]
```

Exemple avec filter():

```
1 numbers = [1, 2, 3, 4, 5]
2 even_numbers = filter(lambda x: x % 2 == 0, numbers)
3 print(list(even_numbers)) # Output: [2, 4]
```

**** Exceptions ****

Principe de fonctionnement

- Les exceptions sont des événements qui se produisent lors de l'exécution d'un programme et qui interrompent le flux normal d'exécution.
- En Python, les exceptions sont des objets qui représentent des erreurs ou des conditions anormales.
- Lorsqu'une exception se produit, le programme s'arrête et affiche un message d'erreur à l'utilisateur.

Exemple d'utilisation

```
1 try:
2     # Code that may generate an exception
3     result = 10 / 0
4 except ZeroDivisionError:
5     # Handling of ZeroDivisionError exception
6     print("Division by zero error!")
```

Exceptions pré-définies et arbre d'héritage

- Python fournit un certain nombre d'exceptions pré-définies pour gérer différents types d'erreurs.
- Ces exceptions sont organisées dans une hiérarchie d'héritage, où les exceptions plus spécifiques héritent des exceptions plus générales.

Exemple d'utilisation

```
1 try:
2     # Code that may generate an exception
3     file = open("example.txt", "r")
4     content = file.read()
5     file.close()
6 except FileNotFoundError:
7     # Handling of FileNotFoundError exception
8     print("File not found!")
```

Instructions `try ... except ... else ... finally`

- L'instruction `try ... except ... else ... finally` est utilisée pour gérer les exceptions de manière plus complexe.
- La clause `else` est exécutée si aucune exception n'est levée dans la clause `try`.
- La clause `finally` est exécutée quel que soit le résultat de la clause `try`, que des exceptions aient été levées ou non.

Exemple d'utilisation

```
1 try:
2     # Code that may generate an exception
3     file = open("example.txt", "r")
4 except FileNotFoundError:
5     # Handling of FileNotFoundError exception
6     print("File not found!")
7 else:
8     # Code executed if no exception is raised
9     print("File read successfully!")
10 finally:
11     # Code executed anyways
12     file.close()
```

Propagation des exceptions

- Lorsqu'une exception est levée dans une fonction, elle peut être propagée vers les fonctions appelantes jusqu'à ce qu'elle soit capturée.
- La propagation des exceptions permet de gérer les erreurs à différents niveaux de la hiérarchie d'appels de fonctions.

Exemple d'utilisation

```
1 def read_file(filename):  
2     try:  
3         file = open(filename, "r")  
4         content = file.read()  
5         file.close()  
6         return content  
7     except FileNotFoundError:  
8         print("File not found!")  
9         raise  
10  
11 try:  
12     data = read_file("example.txt")  
13 except FileNotFoundError:  
14     print("Error reading file!")
```


Déclenchement d'exceptions

- Les exceptions peuvent être déclenchées manuellement à l'aide de l'instruction `raise`.
- Cela permet de signaler des erreurs ou des conditions anormales dans le code.

Exemple d'utilisation

```
1 def validate_age(age):  
2     if age < 0:  
3         raise ValueError("Age cannot be negative!")  
4  
5 try:  
6     validate_age(-5)  
7 except ValueError as e:  
8     print("Invalid age:", e)
```

Définition d'une exception

- En Python, il est possible de définir ses propres exceptions en créant de nouvelles classes dérivées de la classe de base `Exception`.
- Cela permet de personnaliser le comportement des exceptions et de les adapter aux besoins spécifiques de l'application.

Exemple d'utilisation

```
1 class CustomError(Exception):  
2     pass  
3  
4 def process_data(data):  
5     if not data:  
6         raise CustomError("Empty data!")  
7  
8 try:  
9     process_data([])  
10 except CustomError as e:  
11     print("Error:", e)
```

**** Manipulation des fichiers ****

Entrées/Sorties de fichiers en Python

Entrées/Sorties de fichiers en Python

- Les opérations d'entrée et de sortie sur les fichiers sont une partie essentielle de nombreux programmes.
- En Python, vous pouvez utiliser la fonction intégrée `open()` et la méthode `close()` pour travailler avec des fichiers.
- Pour lire des données à partir d'un fichier, vous pouvez utiliser la méthode `read()`.
- Pour écrire des données dans un fichier, vous pouvez utiliser la méthode `write()`.

Ouverture et fermeture de fichiers

- La fonction `open()` est utilisée pour ouvrir un fichier en mode lecture, écriture ou ajout.
- La méthode `close()` est utilisée pour fermer le fichier ouvert.

```
1 # Example of using open() and close()
2 file = open("example.txt", "r")
3 content = file.read()
4 file.close()
```

- **Usage:** L'ouverture et la fermeture de fichiers sont utilisées pour lire ou écrire dans des fichiers texte.
- **Documentation:** Documentation officielle de `open()`,
Documentation officielle de la manipulation des fichiers

Lecture de fichiers ligne par ligne

- La méthode `readline()` est utilisée pour lire une ligne à la fois à partir du fichier.
- La méthode `readlines()` est utilisée pour lire toutes les lignes du fichier et les retourner dans une liste.
- Les fichiers eux-mêmes peuvent être utilisés comme objets itérables.

```
1 # Example of using readline() and readlines()
2 with open("example.txt", "r") as file:
3     line = file.readline()
4     lines = file.readlines()
```

- **Usage:** Ces méthodes sont utiles pour lire des fichiers texte ligne par ligne ou pour récupérer toutes les lignes dans une liste.
- **Documentation:**
[Documentation officielle de la manipulation des fichiers](#)

Utilisation de l'instruction `with`

- L'instruction `with` est utilisée avec les fichiers pour garantir la fermeture automatique du fichier après son utilisation.
- Cela simplifie le code et évite les erreurs de gestion des ressources.

```
1 # Example of using the with instruction
2 with open("example.txt", "r") as file:
3     content = file.read()
```

- **Usage:** L'instruction `with` est recommandée pour la manipulation des fichiers car elle garantit la fermeture automatique du fichier après son utilisation.
- **Documentation:**
Documentation officielle des méthodes d'objets de fichier

Travailler avec des fichiers en Python (1/5)

Par défaut, `open()` ouvre le fichier en mode texte, ce qui signifie qu'il lit et écrit des chaînes de caractères. Si vous souhaitez travailler avec des données binaires, vous pouvez utiliser la fonction `open()` avec le modificateur de mode `b` (`wb` ou `rb`). N'oubliez pas de toujours fermer vos fichiers lorsque vous avez terminé de travailler avec eux, car cela libère toutes les ressources système associées au fichier et contribue à éviter la corruption des données.

Travailler avec des fichiers en Python (2/5)

Travailler avec les chemins de fichiers en Python

- Lorsque vous travaillez avec des fichiers en Python, il est important de spécifier le chemin du fichier correct.
- Le chemin du fichier est l'emplacement du fichier sur votre ordinateur.
- Sous Windows, les chemins de fichiers utilisent le caractère barre oblique inverse \ (backslash) pour séparer les répertoires.
- Sous Linux et macOS, les chemins de fichiers utilisent le caractère barre oblique / (slash ou forward slash).

Travailler avec des fichiers en Python (3/5)

```
1 import os
2 '''      Use the os module in Python to work with file paths.
3 The os.path.join() method is a platform-independent way to join
   path components.      Here is an example of how to use
   os.path.join() to build a file path:      '''
4 file_path = os.path.join("mydirectory", "myfile.txt")
5 # Open the file
6 file = open(file_path, "r")
7 # Read the content of the file
8 data = file.read()
9 # Close the file
10 file.close()
11 # Display the data
12 print(data)
```

Travailler avec des fichiers en Python (4/5)

```
1 import os
2 # Join path components to build a file path
3 file_path = os.path.join("mydirectory", "myfile.txt")
4 # Open the file
5 file = open(file_path, "r")
6 # Read the content of the file
7 data = file.read()
8 # Close the file
9 file.close()
10 # Display the data
11 print(data)
```

Travailler avec des fichiers en Python (5/5)

- Dans cet exemple, `os.path.join()` joint le nom du répertoire `mydirectory` et le nom du fichier `myfile.txt` pour construire le chemin du fichier `mydirectory/myfile.txt`.
- Cela est fait de manière indépendante de la plate-forme, de sorte que le code fonctionnera à la fois sur les systèmes Windows et Unix.
- En utilisant le module `os` et la fonction `open()`, vous pouvez travailler avec des fichiers et des chemins de fichiers de manière à la fois facile et fiable.

Ajouter des données à un fichier en Python

- En plus de lire et d'écrire des données dans un fichier, vous pouvez également ajouter des données à un fichier existant en Python. Cela vous permet d'ajouter de nouvelles données à la fin d'un fichier sans écraser son contenu existant.
- Pour ajouter des données à un fichier, vous pouvez utiliser le modificateur de mode `a` (pour `append`) lors de l'ouverture du fichier. Voici un exemple :

```
1 # Open the file in append mode
2 file = open("myfile.txt", "a")
3 # Write data to the file
4 file.write("Hello, again!")
5 # Close the file
6 file.close()
```

Ajouter des données à un fichier en Python - Suite

- Dans cet exemple, le fichier `myfile.txt` est ouvert en mode ajout en utilisant le modificateur de mode `a` (append).
- La méthode `write()` est utilisée pour ajouter la chaîne "Hello, again!" à la fin du fichier, sans écraser son contenu existant.
- N'oubliez pas de toujours fermer vos fichiers lorsque vous avez terminé de travailler avec, car cela libère toutes les ressources système associées au fichier et contribue à éviter la corruption des données.

Exemple de lecture de données dans les fichiers en Python

■ Voici un exemple de lecture de données à partir d'un fichier :

```
1 # Open the file in read mode
2 my_file = open("myFile.txt", "r")
3
4 # Read the contents of the file
5 data = my_file.read()
6
7 # Close the file
8 my_file.close()
9
10 # Display the data
11 print(data)
```

Exemple d'écriture de données dans les fichiers en Python

■ Et voici un exemple d'écriture de données dans un fichier :

```
1 # Open the file in write mode
2 file = open("myFile.txt", "w")
3
4 # Write data to the file
5 file.write("Hello, world!")
6
7 # Close the file
8 file.close()
```

Méthodes `tell()` et `seek()`

- La méthode `tell()` retourne la position actuelle du curseur dans le fichier.
- La méthode `seek()` permet de déplacer le curseur à une position spécifique dans le fichier.

```
1 # Exemple d'utilisation des méthodes tell() et seek()
2 with open("example.txt", "r") as file:
3     position = file.tell() # Récupère la position actuelle
4     file.seek(0)          # Déplace le curseur au début du fichier
```

- **Usage:** Les méthodes `tell()` et `seek()` sont utiles pour la gestion avancée des fichiers, notamment pour la lecture et l'écriture à des positions spécifiques dans le fichier.
- **Documentation:**
Documentation officielle des méthodes d'objets de fichier

Méthode writelines()

- La méthode `writelines()` permet d'écrire une liste de chaînes de caractères dans un fichier.

```
1 # Example of using writelines() method
2 lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
3 with open("example.txt", "w") as file:
4     file.writelines(lines)
```

- **Usage:** La méthode `writelines()` est utile lorsque vous avez une liste de chaînes de caractères à écrire dans un fichier, sans ajouter de saut de ligne supplémentaire.
- **Documentation:**
Documentation officielle des méthodes d'objets de fichier

Modules complémentaires : struct, csv, json, xml

- Python offre plusieurs modules pour travailler avec différents formats de fichiers et données structurées tels que CSV, JSON et XML.
- Les principaux modules sont `struct`, `csv`, `json` et `xml`.

Module struct

- Le module `struct` permet de convertir des objets Python en chaînes d'octets et vice versa, selon un format spécifié.
- Il est utile pour travailler avec des données binaires structurées.

Exemple d'utilisation du module struct

```
1 import struct
2
3 # Packing data
4 packed_data = struct.pack('i4s', 25, b'abcd')
5
6 # Unpacking data
7 unpacked_data = struct.unpack('i4s', packed_data)
8
9 print("Packed data:", packed_data)
10 print("Unpacked data:", unpacked_data)
```

Module `csv`

- Le module `csv` permet de lire et d'écrire des fichiers CSV (Comma-Separated Values).
- Il fournit des fonctions pour manipuler les données CSV de manière simple et efficace.

Exemple d'utilisation du module csv

```
1 import csv
2
3 # Writing to a CSV file
4 with open('data.csv', 'w', newline='') as csvfile:
5     writer = csv.writer(csvfile)
6     writer.writerow(['Name', 'Age'])
7     writer.writerow(['Alice', 30])
8     writer.writerow(['Bob', 25])
9
10 # Reading from a CSV file
11 with open('data.csv', newline='') as csvfile:
12     reader = csv.reader(csvfile)
13     for row in reader:
14         print(row)
```

Module json

- Le module `json` permet de lire et d'écrire des données au format JSON (JavaScript Object Notation).
- Il offre des fonctions pour convertir des objets Python en JSON et vice versa.

Exemple d'utilisation du module json

```
1 import json
2
3 # Writing to a JSON file
4 data = {"name": "Alice", "age": 30}
5 with open('data.json', 'w') as jsonfile:
6     json.dump(data, jsonfile)
7
8 # Reading from a JSON file
9 with open('data.json') as jsonfile:
10     data = json.load(jsonfile)
11     print(data)
```

Module `xml`

- Le module `xml` permet de manipuler des données au format XML (eXtensible Markup Language).
- Il fournit des fonctionnalités pour analyser, générer et manipuler des documents XML.

Exemple d'utilisation du module `xml`

```
1 import xml.etree.ElementTree as ET
2
3 # Parsing XML from a string
4 xml_string =
5     '<root><item>1</item><item>2</item><item>3</item></root>'
6
7 # Accessing XML elements
8 for item in root.findall('item'):
9     print(item.text)
```

Sérialisation avec le module `pickle`

- La **sérialisation** est le processus de conversion d'un objet Python en un format pouvant être stocké ou transféré, tel qu'une chaîne de caractères, un fichier ou un flux de données.
- Le module `pickle` de Python fournit des fonctions pour la sérialisation et la désérialisation des objets Python.
- Utilisations courantes de la sérialisation avec `pickle` :
 - Sauvegarde et restauration de données complexes.
 - Stockage de l'état des objets pour une utilisation ultérieure.
 - Transfert d'objets sur un réseau.
- Documentation : [Module pickle](#).

Module `pickle`

- Le module `pickle` permet de sérialiser et désérialiser des objets Python en un flux d'octets.
- Il est utile pour sauvegarder et restaurer des objets Python de manière efficace.

Exemple d'utilisation du module pickle

```
1 import pickle
2
3 # Serializing data to a file
4 data = {'name': 'Alice', 'age': 30}
5 with open('data.pickle', 'wb') as file:
6     pickle.dump(data, file)
7
8 # Deserializing data from a file
9 with open('data.pickle', 'rb') as file:
10     loaded_data = pickle.load(file)
11     print(loaded_data)
```


Sérialisation avec le module `shelve`

- Le module `shelve` de Python fournit une interface pour la sérialisation d'objets Python en utilisant un format de stockage de type dictionnaire.
- Il permet de stocker des objets Python dans une base de données de style clé-valeur sur le disque.
- Utilisations courantes de la sérialisation avec `shelve` :
 - Stockage de données structurées pour une utilisation ultérieure.
 - Gestion de configurations ou de paramètres d'application.
 - Mise en cache de données pour améliorer les performances.
- Documentation : [Module shelve](#)

Exemple d'utilisation du module shelve

```
1 import shelve
2
3 # Opening the shelve file
4 with shelve.open('data') as db:
5     # Storing data
6     db['name'] = 'Alice'
7     db['age'] = 30
8
9 # Retrieving data
10 with shelve.open('data') as db:
11     name = db['name']
12     age = db['age']
13     print(name, age)
```