Course: Python language

Ali ZAINOUL <ali.zainoul.az@gmail.com>

for IBM - Needemand
July 17, 2024

1 Fonctions, modules et paquets

- 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()
- Modules
  - Les Modules en Python
  - Les bonnes pratiques
- Paquets

- Bloc `if __name__ == "__main__"`
- Importation de paquet
- Création d'un paquet
- Création d'un paquet (__init__.py)

2 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`

** Fonctions, modules et paquets **

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 :

```
# Definition of our square function
def Square(n):
    return n**2

# Testing our function
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 :

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

```
# Example:
def print_foo(*args):
    for arg in args:
        print(arg)

# Test our function
print_foo("Hello", 1, 2, 3, "World")
''' # Output:
Hello
1
2
3
World
'''
```

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 :

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

```
# Example:
def printDict(**kwargs):
    for (k, v) in kwargs.items():
        print(k, v)

# Test our function
printDict(Day_1="Monday", Day_2="Tuesday")
''' # Output:
Day_1 Monday
Day_2 Tuesday
'''
```

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.

```python
# Define a generator function
def Generator():
    for i in range(10, 0, -1):
        yield i

# Create an instance of the generator
gen = Generator()

# Print the results
print(next(gen)) # Output : 10
print(next(gen)) # Output : 9
# ...
print(next(gen)) # Output : 1
# print(next(gen)) # Error, because we stop at 1
'''
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
'''
```

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

```python
# Exemple d'espace de noms local et global en Python
x = 10  # Variable globale

def fonction():
    y = 20  # Variable locale à la fonction
    print("Variables locales:", locals())

fonction()
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)

```python
# Scope 1: Global Scope
global_variable = "I am global"

def outer_function():
    # Scope 2: Outer Function Scope
    outer_variable = "I am in the outer function"

    def inner_function():
        # Scope 3: Inner Function Scope
        inner_variable = "I am in the inner function"
        print("Inner variable:", inner_variable)
        print("Outer variable inside the inner function:",
            outer_variable)
        print("Global variable inside the inner function:",
            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.

```
# Example of the notion of global variable
global_var = 42

def print_global():
    print("Global variable:", global_var)

print_global()  # Displays "Global variable: 42"
'''
The global_var is accessible inside print_global()'s scope even
   if print_global() method has no variable named global_var.
'''
```

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.

```
def example_function():
    local_variable = "I am local"
    print(local_variable)

example_function()  # Displays "I am local"
# Uncommenting the line below would result in an error
# 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

```
# Declaration of a global variable
index = 0

def modify_index():
    # Redefining the global variable inside the function
        modify_index()
    index = 42
    print("Value of index inside the function:", index) #
        Displays: 42

# Calling the function
modify_index()

# Displaying the value of index outside the function
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 :

```python
global_var = 0
print("Global Scope *before* calling foo() and bar(): global_var = ", global_var)
# Output : Global Scope *before* calling foo(): global_var =  0
def foo():
    global_var = 1
    print("Inside foo()'s Local Scope: global_var = ", global_var)

foo()
# Output : Inside foo()'s Local Scope: global_var =  1
print("Global Scope *after* calling foo(): global_var = ", global_var)
# Output : Global Scope *after* calling foo(): global_var =  0

def bar():
    global global_var
    global_var = 2
    print("Inside bar()'s Local Scope: global_var = ", global_var)

bar()
# Output : Inside bar()'s Local Scope: global_var =  2
print("Global Scope *after* calling bar(): global_var = ", global_var)
# 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()

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

```python
def add(a, b):
    result = a + b
    return result

sum = add(3, 5)
print(sum)  # Output: 8
```

```python
def square_and_cube(x):
    square = x ** 2
    cube = x ** 3
    return square, cube

result = square_and_cube(4)
print(result)  # Output: (16, 64)
```

```python
def hello():
    print("Hello World!")
    return

result = hello()
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

Exemples

```python
# Example of len() function
class CustomList:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

custom_list_instance = CustomList([1, 2, 3, 4, 5])
print(len(custom_list_instance))  # Output: 5
```

```python
class Dog:
    def sound(self):
        return "Woof!"

class Cat:
    def sound(self):
        return "Meow!"

class Duck:
    def sound(self):
        return "Quack!"

def make_sound(animal):
    return animal.sound()

# Usage of duck typing
dog_instance = Dog()
cat_instance = Cat()
duck_instance = Duck()

print(make_sound(dog_instance))     # Output: Woof!
print(make_sound(cat_instance))     # Output: Meow!
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

```python
def greet(name="World"):
    print("Hello", name)

greet()  # Output: Hello World
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

```python
def greet(first_name, last_name):
    print("Hello", first_name, last_name)

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)

```python
def sum(*args):
    result = 0
    for num in args:
        result += num
    return result

print(sum(1, 2, 3, 4))  # Output: 10
```

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(key + ":", value)

print_info(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York
```

Exemple 1 - Mélange de *args et **kwargs

```python
def print_info(*args, **kwargs):
    print("Positional arguments:")
    for arg in args:
        print(arg)
    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(key + ":", value)

print_info("Alice", 30, city="New York", occupation="Engineer")
# Output:
# Positional arguments:
# Alice
# 30
# Keyword arguments:
# city: New York
# occupation: Engineer
```

Exemple 2 - Mélange de *args et **kwargs

```python
def create_person(**kwargs):
    person = {}
    for key, value in kwargs.items():
        person[key] = value
    return person

def print_person_info(*args, **kwargs):
    print("Printing person information:")
    for person in args:
        print("Name:", person["name"])
        print("Age:", person["age"])
        for key, value in kwargs.items():
            if value:
                print(key.capitalize() + ":", person[key])
        print("\n")

# Creating persons using **kwargs
person1 = create_person(name="Alice", age=30, city="New York", occupation="Engineer")
person2 = create_person(name="Bob", age=25, city="Los Angeles", occupation="Artist")
person3 = create_person(name="Charlie", age=35, city="Chicago")

# Printing person information
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

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

```
# Example 2: lambda function with 2 parameters
fun = lambda a, b : a ** b
print(fun(7, 2))
#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:

```
# Define a multiplier function
def multiplier(n):
    return lambda x: x * n

# Create instances of the multiplier function
Doubler = multiplier(2)
Tripler = multiplier(3)

# Print the results
print(Doubler(33))  # Outputs: 66
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():

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

Exemple avec exec():

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

Exemple avec map():

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

Exemple avec filter():

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

Modules en Python : Introduction (1/3)

- **Qu'est-ce que les modules en Python ?** En Python, un module est un fichier contenant du code Python qui peut être importé dans d'autres scripts Python. Les modules sont utilisés pour organiser le code dans des fichiers distincts et fournissent un moyen de réutiliser le code dans différents programmes. Pour créer un module, il suffit d'écrire du code Python dans un fichier avec une extension `.py`. Vous pouvez ensuite importer le module dans un autre script Python à l'aide de l'instruction **import**.

Modules en Python : Importation (2/3)

- **Comment importer des modules en Python ?** Pour importer un module en Python, utilisez l'instruction **import** suivie du nom du module. Par exemple, si vous avez un module nommé `my_module.py`, vous pouvez l'importer dans un autre script Python avec l'instruction suivante :

```
1    import my_module
```

Modules en Python : Utilisation (3/3)

- Une fois que vous avez importé le module, vous pouvez utiliser ses fonctions et variables dans votre code. Pour appeler une fonction du module, utilisez la notation pointée, comme ceci :

```
1    my_module.my_function()
```

- Vous pouvez également utiliser le mot-clé **from** pour importer des classes, des fonctions ou des variables spécifiques d'un module, comme ceci :

```
1    from my_module import my_class, my_function, my_variable
```

- Cela vous permet d'utiliser les classes, les fonctions et les variables importées sans avoir à les préfixer par le nom du module.

Bonnes pratiques pour les modules en Python (1/2)

- Lors de l'importation de modules en Python, il existe certaines bonnes pratiques que vous devriez suivre pour vous assurer que votre code est propre, **lisible** et **maintenable**. Voici quelques conseils:

  - Utilisez des importations absolues pour spécifier le chemin complet du module que vous souhaitez importer. Cela permet de préciser le module que vous importez et aide à éviter les conflits de noms.
  - Évitez d'utiliser des importations par joker (par exemple: `from mon_module import *`) car elles peuvent rendre difficile la compréhension d'où proviennent les fonctions et variables.
  - Utilisez des alias pour raccourcir les noms de module et les rendre plus faciles à utiliser. Par exemple, vous pourriez importer le module `numpy` de cette manière : `import numpy as np`.

Bonnes pratiques pour les modules en Python (2/2)

- Suite :
  - Regroupez les importations liées ensemble en haut de votre script pour indiquer clairement les modules dont dépend votre code.
  - Évitez les importations circulaires, où deux modules dépendent l'un de l'autre. Cela peut créer des dépendances confuses et rendre votre code plus difficile à comprendre.
- En suivant ces bonnes pratiques, vous pouvez vous assurer que votre code Python est bien organisé, facile à lire et maintenable au fil du temps.
- Bonnes pratiques.

Bloc `if __name__ == "__main__"`

- En Python, le bloc `if __name__ == "__main__"` est utilisé pour déterminer si le fichier est exécuté en tant que script principal ou s'il est importé en tant que module dans un autre script.
- Le code à l'intérieur du bloc `if __name__ == "__main__"` ne sera exécuté que si le fichier est exécuté en tant que script principal.
- Cela empêche l'exécution non voulue du code lorsque le fichier est importé en tant que module.

Bloc `if __name__ == "__main__"` - Exemple

```python
# module.py
def function():
    print("Function called")

if __name__ == "__main__":
    # Code to execute only if this script is run as the main
       program
    print("Module is being run directly")
    function()
else:
    # Code to execute if this script is imported as a module
    print("Module is being imported")
```

Importation de paquet

- En Python, un paquet est simplement un répertoire contenant un fichier spécial `__init__.py`.
- L'importation de paquet permet d'organiser et de structurer le code en regroupant des modules connexes dans un même répertoire.
- L'importation de paquet se fait avec la syntaxe `import name_package.name_module`.
- Pour importer tous les modules d'un paquet, on utilise la syntaxe `from name_package import *`, mais cela est généralement déconseillé pour éviter les conflits de noms.

Importation de paquet - Exemple

```python
# Paquet: mypackage
# Structure:
# mypackage/
#       __init__.py
#       module1.py
#       module2.py

# Content of file __init__.py
# (may be empty or containing initializations)
# __init__.py

# Importing a module from a package
import mypackage.module1
# Using the module
mypackage.module1.function()

# Importing of all modules from a package
from mypackage import *
# Utilisation des modules importés
module1.function()
module2.function()

# Importing using alias
import mypackage.module1 as m1
# Using with alias
m1.function()
```

Création de paquet

- En Python, un paquet est simplement un répertoire contenant un fichier spécial `__init__.py`.
- Le fichier `__init__.py` peut être vide ou contenir des initialisations.
- La création de paquet permet d'organiser et de structurer le code en regroupant des modules connexes dans un même répertoire.
- Pour créer un paquet, il suffit de créer un répertoire et d'y ajouter un fichier `__init__.py`.

Création de paquet - Exemple

```
# Paquet: mypackage
# Structure tree:
# mypackage/
#       __init__.py
#       module1.py
#       module2.py

# Content of file __init__.py
# (may be empty or containing initializations)
# __init__.py

# Content of module1.py
# module1.py
def function1():
    print("Function 1 in module 1")

# Content of module2.py
# module2.py
def function2():
    print("Function 2 in module 2")
```

Création d'un paquet

- Pour créer un paquet en Python, il suffit de créer un répertoire avec un fichier `__init__.py` à l'intérieur.
- Le fichier `__init__.py` peut être vide ou contenir du code d'initialisation pour le paquet.
- Voici un exemple de structure de répertoire pour un paquet nommé `mypackage`:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

- Dans cet exemple, `mypackage` est un paquet Python contenant deux modules, `module1` et `module2`.

Example 1: Creating a Package - Structure

```
mypackage/
   __init__.py
   module1.py
   module2.py
```

Example 1: Package Initialization (__init__.py)

```
# __init__.py

from .module1 import MyClass1
from .module2 import MyClass2
```

Example 1: Module 1 (module1.py)

```python
# module1.py

class MyClass1:
    def __init__(self):
        print("Initializing MyClass1")

    def method1(self):
        print("Method 1 in MyClass1")
```

Example 1: Module 2 (module2.py)

```python
# module2.py

class MyClass2:
    def __init__(self):
        print("Initializing MyClass2")

    def method2(self):
        print("Method 2 in MyClass2")
```

Example 2: Creating a Package - Structure

```
mypackage/
   __init__.py
   utils/
       __init__.py
       helper1.py
       helper2.py
   main.py
```

Example 2: Package Initialization (__init__.py)

```python
# __init__.py

from .utils.helper1 import Helper1
from .utils.helper2 import Helper2
```

Example 2: Module 1 (helper1.py)

```python
# helper1.py

class Helper1:
    def __init__(self):
        print("Initializing Helper1")

    def method1(self):
        print("Method 1 in Helper1")
```

Example 2: Module 2 (helper2.py)

```python
# helper2.py

class Helper2:
    def __init__(self):
        print("Initializing Helper2")

    def method2(self):
        print("Method 2 in Helper2")
```

Example 2: Main Module (main.py)

```python
# main.py

from mypackage.utils import Helper1, Helper2

helper1 = Helper1()
helper1.method1()

helper2 = Helper2()
helper2.method2()
```

** 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.

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

```
import os
'''     Use the os module in Python to work with file paths.
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:       '''
file_path = os.path.join("mydirectory", "myfile.txt")
# Open the file
file = open(file_path, "r")
# Read the content of the file
data = file.read()
# Close the file
file.close()
# Display the data
print(data)
```

Travailler avec des fichiers en Python (4/5)

```python
import os
# Join path components to build a file path
file_path = os.path.join("mydirectory", "myfile.txt")
# Open the file
file = open(file_path, "r")
# Read the content of the file
data = file.read()
# Close the file
file.close()
# Display the data
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 :

```
# Open the file in append mode
file = open("myfile.txt", "a")
# Write data to the file
file.write("Hello, again!")
# Close the file
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 :

```
# Open the file in read mode
my_file = open("myFile.txt", "r")

# Read the contents of the file
data = my_file.read()

# Close the file
my_file.close()

# Display the data
print(data)
```

Exemple d'écriture de données dans les fichiers en Python

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

```
# Open the file in write mode
file = open("myFile.txt", "w")

# Write data to the file
file.write("Hello, world!")

# Close the file
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.

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

```python
import struct

# Packing data
packed_data = struct.pack('i4s', 25, b'abcd')

# Unpacking data
unpacked_data = struct.unpack('i4s', packed_data)

print("Packed data:", packed_data)
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`

```python
import csv

# Writing to a CSV file
with open('data.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['Name', 'Age'])
    writer.writerow(['Alice', 30])
    writer.writerow(['Bob', 25])

# Reading from a CSV file
with open('data.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        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`

```python
import json

# Writing to a JSON file
data = {"name": "Alice", "age": 30}
with open('data.json', 'w') as jsonfile:
    json.dump(data, jsonfile)

# Reading from a JSON file
with open('data.json') as jsonfile:
    data = json.load(jsonfile)
    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 =
      '<root><item>1</item><item>2</item><item>3</item></root>'
5 root = ET.fromstring(xml_string)
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 : Exemples de serialization, 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`

```python
import pickle

# Serializing data to a file
data = {'name': 'Alice', 'age': 30}
with open('data.pickle', 'wb') as file:
    pickle.dump(data, file)

# Deserializing data from a file
with open('data.pickle', 'rb') as file:
    loaded_data = pickle.load(file)
    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`

```python
import shelve

# Opening the shelve file
with shelve.open('data') as db:
    # Storing data
    db['name'] = 'Alice'
    db['age'] = 30

# Retrieving data
with shelve.open('data') as db:
    name = db['name']
    age = db['age']
    print(name, age)
```