JOUR 1_1

Fonctions avancées et programmation fonctionnelle

Fonctions anonymes (lambda functions) et fonctions map, filter, reduce

Ce contenu illustre l'utilisation des fonctions anonymes (lambda functions) et des fonctions `map`, `filter`, et `reduce` en Python, montrant comment elles peuvent être utilisées pour effectuer des opérations sur des données de manière concise et efficace.

Fonctions Anonymes (Lambda Functions)

Les fonctions anonymes, également appelées fonctions lambda, sont des fonctions sans nom définies à l'aide de `lambda`.

```
lambda arguments: expression
```

Exemple d'utilisation :

```
# Addition de deux nombres avec une fonction lambda
addition = lambda x, y: x + y
print(addition(3, 5))  # Output: 8
```

Fonctions `map`, `filter`, `reduce`

Fonction `map`

La fonction `map` applique une fonction à chaque élément d'une séquence et renvoie un iterable contenant les résultats.

```
map(function, iterable)
```

```
# Doubler chaque élément d'une liste avec map et lambda
numbers = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)  # Output: [2, 4, 6, 8, 10]
```

Fonction `filter`

La fonction `filter` filtre les éléments d'une séquence en fonction d'une fonction de filtrage et renvoie les éléments pour lesquels la fonction renvoie `True`.

```
filter(function, iterable)
```

```
# Filtrer les nombres pairs d'une liste avec filter et lambda
numbers = [1, 2, 3, 4, 5]
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even)  # Output: [2, 4]
```

Fonction `reduce`

La fonction `reduce` applique une fonction de manière cumulative à la séquence pour retourner une valeur unique.

Remarque : À partir de Python 3, `reduce` n'est pas une fonction intégrée et doit être importée depuis le module `functools`.

Syntaxe :

```
from functools import reduce
reduce(function, iterable)
```

Exemple d'utilisation

```
# Additionner les éléments d'une liste avec reduce et lambda
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum = reduce(lambda x, y: x + y, numbers)
print(sum)  # Output: 15
```

List comprehensions, generator expressions

List comprehensions

Les list comprehensions sont une façon concise de créer des listes en Python en une seule ligne de code.

Syntaxe :

```
[expression for item in iterable if condition]
```

Exemple d'utilisation:

```
# Création d'une liste avec les carrés des nombres de 1 à 5
squares = [x ** 2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Generator expressions

Les generator expressions sont similaires aux list comprehensions mais génèrent des générateurs plutôt que des listes, ce qui les rend plus efficaces en termes de mémoire.

Syntaxe :

```
(expression for item in iterable if condition)
```

Exemple d'utilisation

```
# Création d'un générateur pour les nombres pairs de 0 à 9
even_generator = (x for x in range(10) if x % 2 == 0)
print(list(even_generator))  # Output: [0, 2, 4, 6, 8]
```

Différences entre List Comprehensions et Generator Expressions

- **List Comprehensions** créent une liste complète en mémoire.
- **Generator Expressions** génèrent des éléments un par un à la demande, économisant de la mémoire.

Exemple pour illustrer la différence :

```
# List Comprehension pour créer une liste de carrés
squares_list = [x ** 2 for x in range(1000000)]

# Generator Expression pour créer un générateur de carrés
squares_generator = (x ** 2 for x in range(1000000))

print(type(squares_list))      # Output: <class 'list'>
print(type(squares_generator)) # Output: <class 'generator'>
```

Les list comprehensions sont utiles pour créer des listes complètes tandis que les generator expressions sont utilisées pour générer des éléments un par un, économisant ainsi de la mémoire, notamment lorsqu'on travaille avec de grandes quantités de données.

Cas d'usage

- Idéal pour **gérer de grandes quantités de données** sans saturer la mémoire.
- Très utile lorsqu'on veut **lire un fichier ligne par ligne**, sans le charger entièrement.

```
# Lire un fichier ligne par ligne sans tout stocker en mémoire
lines = (line.strip() for line in open("fichier.txt"))
for line in lines:
    print(line)
```

Utilisation avancée des fonctions récursives

Une **fonction récursive** est une fonction qui **s'appelle elle-même** dans sa propre définition, jusqu'à ce qu'une **condition d'arrêt** soit atteinte.

Exemple :

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

1. Récursion Terminale

- **Définition :** Une récursion est dite terminale lorsqu'elle se produit à la fin de la fonction.
- **Avantage :** Permet une optimisation pour éviter l'accumulation de frames de pile.
- **Exemple :** La récursion terminale est souvent utilisée dans le cas de la factorielle.

```
def factorial(n):
    if n == 0 or n == 1:  # Condition d'arrêt
        return 1
```

```
    return n * factorielle(n - 1)  # Appel récursif

print(factorielle(5))  # 5 * 4 * 3 * 2 * 1 = 120
```

```
def factorial(n, result=1):
    if n == 0 or n == 1:
        return result
    else:
        return factorial(n - 1, n * result)

print(factorial(5))  # Output: 120
```

2. Récursion Mutuelle

- **Définition :** Plusieurs fonctions se rappellent mutuellement.
- **Utilisation :** Utile pour résoudre des problèmes impliquant des états alternés entre deux fonctions.
- **Exemple :** Paire et impaire qui s'alternent dans une séquence.

```
def est_pair(n):
    if n == 0:
        return True  # 0 est pair
    return est_impair(n - 1)  # Appel à la fonction est_impair

def est_impair(n):
    if n == 0:
        return False  # 0 n'est pas impair
    return est_pair(n - 1)  # Appel à la fonction est_pair

print(est_pair(10))  # True
print(est_impair(10))  # False
print(est_pair(7))  # False
print(est_impair(7))  # True
```

3. Récursion avec Mémoïsation (Memoization)

- **Définition :** Stocke les résultats des appels de fonction précédents pour éviter de recalculer.
- **Utilisation :** Accélère les fonctions récursives en réduisant les calculs répétitifs.
- **Exemple :** Fibonacci avec mémoïsation.

$F(n)=F(n-1)+F(n-2)$

Avec :

- $F(0)=0$
- $F(1)=1$

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(6))  # 8
```

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    else:
        memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2,
memo)
        return memo[n]

print(fibonacci(6))  # Output: 8
```

4. Récursion sur Arbres et Structures de Données Complexes

- **Définition :** Utilisation de la récursion pour traverser et manipuler des structures de données arborescentes.
- **Exemple :** Parcours d'un arbre binaire.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def print_tree(node):
    if node:
        print(node.value)
        print_tree(node.left)
        print_tree(node.right)

# Création d'un arbre binaire simple
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print_tree(root)
# Output:
# 1
# 2
# 4
# 5
```

```
# 3
```