Université Mohammed V de Rabat
Faculté des sciences

La Programmation en Python
Notions Avancées

Pr. Nassim KHARMOUM

ESSAMADI Oussama

MSID
2023/2024

Python - Notions Avancées

Les Itérateurs

Les itérateurs sont un concept fondamental en programmation qui permet de parcourir séquentiellement les éléments d'une séquence de données, tels que des listes, des tuples, des chaînes de caractères, ou même des structures de données plus complexes. En Python, les itérateurs sont largement utilisés pour parcourir des objets itérables, c'est-à-dire des objets qui peuvent être parcourus un élément à la fois.

Voici comment les itérateurs fonctionnent en Python :

```
1   my_list = [1, 2, 3, 4, 5]
2   my_iterator = iter(my_list)
3
```

- Création d'un itérable : Tout objet en Python qui implémente la méthode spéciale **`__iter__()`** est considéré comme un itérable. Cette méthode renvoie un objet itérateur, généralement le même objet lui-même ou un nouvel objet itérateur associé à l'itérable.
- Obtenir un itérateur : Pour obtenir un itérateur à partir d'un itérable, on utilise la fonction **`iter()`**. Par exemple :

Python - Notions Avancées

Les Itérateurs

Parcours des éléments : Une fois que vous avez un itérateur, vous pouvez utiliser la méthode __next__() ou la fonction next() pour obtenir l'élément suivant de la séquence. Lorsque tous les éléments ont été parcourus, l'itérateur génère une exception StopIteration pour signaler la fin de la séquence.

Exemple de code pour créer et utiliser des itérateurs en Python :

```python
# Création d'une liste et d'un itérateur associé
my_list = [1, 2, 3, 4, 5]
my_iterator = iter(my_list)

# Parcours des éléments de la liste à l'aide de l'itérateur
try:
    while True:
        element = next(my_iterator)
        print(element)
except StopIteration:
    pass

```

Python - Notions Avancées

Les Itérateurs

Avantages et cas d'utilisation des itérateurs :

- **Économie de mémoire** : Les itérateurs permettent de parcourir des données de manière séquentielle sans avoir besoin de stocker la séquence entière en mémoire. Cela est particulièrement utile pour les données volumineuses.
- **Performance** : Les itérateurs peuvent améliorer la performance en permettant un accès efficace aux éléments d'une séquence sans avoir à copier ou à stocker la séquence complète.
- **Générateurs** : Les fonctions génératrices en Python utilisent des itérateurs pour générer des séquences d'éléments à la volée, ce qui permet de créer des itérations infinies ou de gérer des séquences paresseuses.
- **Boucles for** : Les boucles for en Python utilisent des itérateurs de manière transparente pour parcourir les éléments d'une séquence, ce qui simplifie le code et le rend plus lisible.

En résumé, les itérateurs sont un concept puissant en Python qui permet de parcourir des données de manière efficace, économique en mémoire et élégante. Ils sont largement utilisés dans la programmation Python pour gérer des séquences de données de toutes sortes.

Python - Notions Avancées

Les Générateurs

Les Générateurs.

Les générateurs sont des structures de données spéciales en Python qui permettent de créer des séquences d'éléments de manière paresseuse, c'est-à-dire au fur et à mesure des besoins, plutôt que de stocker la séquence complète en mémoire. Les générateurs sont définis à l'aide de fonctions spéciales appelées "fonctions génératrices" ou en utilisant des expressions génératrices.

Voici comment les générateurs fonctionnent et comment ils se comparent aux itérateurs :

- **Fonctionnement des générateurs :**
  - **Définition** : Un générateur est défini à l'aide d'une fonction qui contient un ou plusieurs mots-dés **`yield`**. Lorsqu'une fonction génératrice est appelée, elle ne s'exécute pas immédiatement, mais retourne un objet générateur qui peut être utilisé pour parcourir les valeurs générées.
  - **Suspension de l'exécution** : Lorsque la fonction génératrice atteint une instruction **`yield`**, elle suspend temporairement son exécution et renvoie la valeur indiquée par **`yield`**. L'état de la fonction est préservé, de sorte qu'elle peut être reprise là où elle s'était arrêtée lors du prochain appel à **`next()`**.
  - **Parcours paresseux** : Les valeurs sont générées au fur et à mesure des besoins, ce qui permet de gérer des séquences potentiellement infinies sans consommer beaucoup de mémoire.
- **Comparaison entre générateurs et itérateurs :**
  - Les itérateurs sont des objets qui permettent de parcourir séquentiellement des séquences de données, tandis que les générateurs sont des fonctions spéciales qui génèrent des valeurs au fur et à mesure des besoins.
  - Les itérateurs sont créés à partir d'objets itérables en utilisant la fonction **`iter()`**, tandis que les générateurs sont créés à l'aide de fonctions génératrices ou d'expressions génératrices.
  - Les générateurs permettent un parcours paresseux des données, tandis que les itérateurs parcourront la séquence entière et stockeront potentiellement toutes les valeurs en mémoire.

Python - Notions Avancées

Les Générateurs

Exemples de code pour créer et utiliser des générateurs en Python :

À l'aide d'une fonction génératrice :

```python
def generate_numbers(n):
    for i in range(n):
        yield i

# Création d'un générateur
my_generator = generate_numbers(5)

# Parcours des valeurs générées
for num in my_generator:
    print(num)

```

À l'aide d'une expression génératrice :

```python
my_generator = (x for x in range(5))

# Parcours des valeurs générées
for num in my_generator:
    print(num)

```

Python - Notions Avancées

Les Générateurs

Utilisations pratiques des générateurs :

- Traitement de données volumineuses : Les générateurs sont utiles pour traiter de grandes quantités de données en évitant de les charger toutes en mémoire.
- Génération de séquences infinies : Les générateurs peuvent être utilisés pour créer des séquences infinies, comme des nombres premiers ou des séquences de Fibonacci.
- Lecture de fichiers : Les générateurs peuvent être utilisés pour lire de grands fichiers ligne par ligne, ce qui évite de charger tout le fichier en mémoire.
- Travail avec des flux de données en temps réel : Les générateurs sont adaptés pour travailler avec des données en temps réel, telles que des données provenant de capteurs ou de flux de réseau.

En résumé, les générateurs sont une fonctionnalité puissante de Python qui permettent de gérer efficacement des séquences de données, en particulier lorsqu'il s'agit de données volumineuses ou de séquences potentiellement infinies. Ils offrent un moyen élégant de gérer le traitement paresseux des données.

Python - Notions Avancées

Les Closures

Les closures sont un concept avancé en programmation qui se réfère à une fonction qui capture des variables locales d'une fonction englobante (fonction parente) même après que la fonction englobante ait terminé son exécution. En d'autres termes, une closure est une fonction qui conserve un accès à l'état de son environnement lexical (les variables locales) au moment de sa définition, même lorsque cette définition n'est plus en cours d'exécution.

Voici comment et pourquoi utiliser les closures en Python :

Comment utiliser les closures en Python :

- Pour créer une closure en Python, vous définissez une fonction à l'intérieur d'une autre fonction, et la fonction interne fait référence à des variables locales de la fonction externe. La fonction externe renvoie la fonction interne, qui peut être appelée et utilisée plus tard, tout en maintenant l'accès aux variables locales de la fonction externe.

Pourquoi utiliser les closures en Python :

- **Encapsulation de données** : Les closures permettent d'encapsuler des données et des comportements liés dans une fonction, offrant une meilleure modularité et évitant les fuites d'information.
- **Gestion d'état** : Les closures sont utiles pour maintenir l'état interne des objets ou des fonctionnalités, notamment pour les gestionnaires d'état, les compteurs, les générateurs et les gestionnaires de contexte.
- **Fonctions de rappel (callbacks)** : Les closures sont couramment utilisées pour définir des fonctions de rappel, qui sont passées comme arguments à d'autres fonctions pour être exécutées ultérieurement.

Python - Notions Avancées

Les Closures

Exemples de code illustrant les closures en Python :

```python
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

# Création de closures
double = multiplier(2)
triple = multiplier(3)

# Utilisation des closures
print(double(5))  # Résultat : 10
print(triple(5))  # Résultat : 15

```

Dans cet exemple, **`multiplier()`** est une fonction englobante qui capture la variable **`factor`**. Lorsque vous appelez **`multiplier(2)`**, elle retourne la fonction **`multiply`**, qui est une closure. Cette closure conserve l'accès à la variable **`factor`**, ce qui permet d'utiliser **`double`** et **`triple`** pour multiplier les nombres par 2 et 3 respectivement.

Python - Notions Avancées

Les Closures

Avantages des closures dans la programmation Python :

- **Encapsulation de données** : Les closures aident à encapsuler des données et des comportements associés, ce qui favorise la modularité et la réutilisation du code.
- **Gestion de l'état** : Les closures sont idéales pour la gestion d'état, car elles permettent de maintenir des variables d'état sans avoir à les rendre globales.
- **Fonctions de rappel** : Les closures sont couramment utilisées pour créer des fonctions de rappel, ce qui facilite la mise en œuvre de la programmation asynchrone et de la programmation orientée événements.
- **Économie d'espace** : Les closures ne conservent que les variables nécessaires, ce qui peut économiser de la mémoire par rapport à la création de classes pour le même objectif.

En résumé, les closures sont un concept puissant en Python qui permettent d'encapsuler des données et des comportements, de gérer l'état et de créer des fonctions de rappel de manière élégante. Elles sont largement utilisées dans la programmation Python pour améliorer la modularité et la lisibilité du code.

Python - Notions Avancées

Les Decorators

Les décorateurs sont une fonctionnalité puissante en Python qui permettent de modifier ou d'étendre le comportement d'une fonction ou d'une méthode sans la modifier directement.

Ils sont souvent utilisés pour ajouter des fonctionnalités aux fonctions existantes, pour la gestion des autorisations, la journalisation, la mesure du temps d'exécution, ou d'autres tâches de manipulation de fonctions.

Définition et rôle des décorateurs en Python :

Un décorateur est une fonction Python qui prend une autre fonction comme argument et retourne une nouvelle fonction qui peut ajouter ou modifier le comportement de la fonction d'origine.

Les décorateurs sont couramment utilisés pour ajouter des fonctionnalités globales à plusieurs fonctions ou méthodes sans répéter le même code.

Python - Notions Avancées

Les Decorators

Exemples de décorateurs et leur utilisation :

Un exemple simple de décorateur pour mesurer le temps d'exécution d'une fonction. Dans cet exemple, **`timing_decorator`** est un décorateur qui mesure le temps d'exécution de la fonction **`example_function`**.

```python
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} a pris {end_time - start_time} secondes pour s'exécuter")
        return result
    return wrapper

@timing_decorator
def example_function():
    # Code de la fonction à mesurer
    time.sleep(2)

example_function()
```

Python - Notions Avancées

Les Decorators

Un exemple de décorateur pour vérifier les autorisations d'accès à une fonction :

Dans cet exemple, **check_permission** est un décorateur qui vérifie si l'autorisation est accordée avant d'exécuter la fonction **restricted_function**.

```python
def check_permission(permission_level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if permission_level == "admin":
                return func(*args, **kwargs)
            else:
                return "Permission refusée"
        return wrapper
    return decorator

@check_permission("admin")
def restricted_function():
    return "Accès accordé"

print(restricted_function())  # Résultat : "Accès accordé"

```

Python - Notions Avancées

Les Decorators

Création de décorateurs personnalisés :

Pour créer des décorateurs personnalisés, vous définissez simplement une fonction qui prend une autre fonction en argument, définissez une fonction intermédiaire (wrapper) à l'intérieur du décorateur, modifiez ou ajoutez le comportement souhaité à cette fonction intermédiaire, puis retournez cette fonction intermédiaire.

Utilisations courantes des décorateurs dans les projets Python : Les décorateurs sont largement utilisés dans les projets Python pour diverses tâches, notamment :

- La gestion des autorisations et de l'authentification.
- La journalisation (enregistrement des actions dans les journaux).
- La validation et la transformation des données d'entrée.
- La gestion des transactions et des bases de données.
- La mesure du temps d'exécution.
- La mise en cache pour l'optimisation des performances.
- L'internationalisation et la localisation.
- La gestion des erreurs et des exceptions.
- La gestion des états de session pour les applications web.

En résumé, les décorateurs sont un mécanisme puissant en Python pour ajouter ou modifier le comportement des fonctions ou des méthodes de manière propre et réutilisable. Ils sont largement utilisés dans la programmation Python pour améliorer la modularité et la maintenabilité du code.

Python - Notions Avancées

Property

La fonctionnalité **`property`** en Python permet de définir des méthodes d'accès spéciales (getter, setter, deleter) pour manipuler les attributs d'une classe. Elle offre un moyen de contrôler l'accès et la modification des attributs d'une manière plus flexible et encapsulée que l'utilisation directe des attributs.

Explication de la fonctionnalité property en Python :

- En Python, les attributs d'une classe sont généralement accessibles directement par l'intermédiaire de méthodes comme **`obj.attribut`**. Cependant, il peut y avoir des cas où vous souhaitez avoir plus de contrôle sur la manière dont les attributs sont accessibles et modifiés, ou vous voulez exécuter des actions spécifiques lors de ces opérations. C'est là que la fonctionnalité **`property`** entre en jeu.
- La fonction **`property()`** permet de définir des méthodes spéciales (**`getter`**, **`setter`**, **`deleter`**) pour un attribut particulier d'une classe. Cela permet de personnaliser le comportement d'accès, de modification et de suppression de cet attribut.

Python - Notions Avancées

Property

La syntaxe générale pour la définition d'une **`property`** est la suivante :

Avec cette approche, vous pouvez utiliser **`obj.mon_attribut`** pour accéder à l'attribut, **`obj.mon_attribut = valeur`** pour le modifier et **`del obj.mon_attribut`** pour le supprimer.

```python
class MaClasse:
    def __init__(self):
        self._mon_attribut = None  # Attribut interne (convention avec un nom préfixé par un underscore)

    def get_attribut(self):
        return self._mon_attribut

    def set_attribut(self, valeur):
        self._mon_attribut = valeur

    def del_attribut(self):
        del self._mon_attribut

    mon_attribut = property(get_attribut, set_attribut, del_attribut)

```

Python - Notions Avancées

Property

Exemples de code pour l'implémentation de property :

```python
class Personne:
    def __init__(self, nom, prenom):
        self._nom = nom
        self._prenom = prenom

    # Getter
    @property
    def nom(self):
        return self._nom

    # Setter
    @nom.setter
    def nom(self, nouveau_nom):
        if isinstance(nouveau_nom, str):
            self._nom = nouveau_nom
        else:
            raise ValueError("Le nom doit être une chaîne de caractères.")

    # Getter
    @property
    def prenom(self):
        return self._prenom

    # Setter
    @prenom.setter
    def prenom(self, nouveau_prenom):
        if isinstance(nouveau_prenom, str):
            self._prenom = nouveau_prenom
        else:
            raise ValueError("Le prénom doit être une chaîne de caractères.")

# Utilisation des propriétés
personne = Personne("Doe", "John")
print(personne.nom)  # Accès à l'attribut nom via la propriété
personne.nom = "Smith"  # Modification de l'attribut nom via la propriété
print(personne.nom)

```

Python - Notions Avancées

Property

Avantages de l'utilisation de property pour l'encapsulation des données :

- **Encapsulation** : La fonction **`property`** permet de masquer les détails de l'implémentation de l'attribut tout en offrant un contrôle complet sur son accès et sa modification.
- **Validation** : Vous pouvez ajouter des vérifications et des validations dans les méthodes getter et setter pour garantir que les données sont correctes.
- **Compatibilité avec le code existant** : Vous pouvez introduire des propriétés dans une classe sans perturber le code client existant, car l'accès à l'attribut est transparent.
- **Lisibilité** : Les propriétés rendent le code plus lisible en remplaçant l'accès direct à l'attribut par un accès à la propriété, ce qui permet de mieux comprendre ce qui se passe.

En résumé, la fonctionnalité **`property`** en Python est un moyen puissant d'encapsuler les données d'une classe et de personnaliser l'accès et la modification de ces données. Elle permet de garantir l'intégrité des données et d'améliorer la lisibilité et la maintenance du code.