Conceitos Avançados em Python

Índice

Multiprocessing

O módulo `multiprocessing` permite a execução paralela de processos em Python, melhorando o desempenho em tarefas CPU-bound.

```
from multiprocessing import Process

def worker():
    print("Processo em execução")

if __name__ == "__main__":
    process = Process(target=worker)
    process.start()
    process.join()
```

Saída esperada:

```
Processo em execução
```

Threads

Threads permitem a execução paralela dentro de um único processo, ideal para tarefas I/O-bound.

```
import threading

def worker():
    print("Thread em execução")

thread = threading.Thread(target=worker)
thread.start()
thread.join()
```

Saída esperada:

```
Thread em execução
```

Singleton

O padrão Singleton garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a essa instância.

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
        return cls._instance

singleton1 = Singleton()
singleton2 = Singleton()

print(singleton1 is singleton2)  # True
```

Saída esperada:

```
True
```

Decorators

Decorators são uma forma de modificar o comportamento de funções ou métodos em Python, sem alterar seu código fonte.

```
def my_decorator(func):
    def wrapper():
        print("Algo acontece antes da função.")
        func()
        print("Algo acontece depois da função.")
    return wrapper

@my_decorator
def say_hello():
    print("Olá!")

say_hello()
```

Saída esperada:

```
Algo acontece antes da função.
Olá!
Algo acontece depois da função.
```

Context Managers

Context Managers permitem a alocação e liberação de recursos de forma eficiente, utilizando as palavras-chave `with` e `__enter__`/`__exit__`.

```
class MyContextManager:
    def __enter__(self):
        print("Entrando no contexto")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Saindo do contexto")

with MyContextManager():
    print("Dentro do bloco de contexto")
```

Saída esperada:

```
Entrando no contexto
Dentro do bloco de contexto
Saindo do contexto
```

Metaclasses

Metaclasses são classes de classes que definem como classes se comportam. Elas permitem modificar a criação de classes de uma maneira controlada.

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Metaclasse criando a classe: {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass
```

Saída esperada:

```
Metaclasse criando a classe: MyClass
```

Generators

Generators são uma forma de criar iteradores de maneira fácil e eficiente, utilizando a palavra-chave `yield`.

```
def my_generator():
    yield 1
    yield 2
    yield 3

for value in my_generator():
    print(value)
```

Saída esperada:

```
1
2
3
```

Asyncio

O módulo `asyncio` permite escrever código assíncrono, facilitando a execução de operações I/O-bound sem bloquear o loop de eventos.

```
import asyncio

async def say_hello():
    print("Olá!")
    await asyncio.sleep(1)
    print("Adeus!")

asyncio.run(say_hello())
```

Saída esperada:

```
Olá!
Adeus!
```

Comprehensions

Comprehensions são uma maneira concisa e eficiente de criar listas, conjuntos e dicionários a partir de *iteráveis*. *

```
# List comprehension
squares = [x**2 for x in range(5)]
print(squares)

# Dict comprehension
square_dict = {x: x**2 for x in range(5)}
print(square_dict)
```

Saída esperada:

```
[0, 1, 4, 9, 16]
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Explicações Resumidas

Multiprocessing: Executa tarefas em paralelo utilizando múltiplos processos, útil para tarefas que exigem muito da CPU.

Threads: Executa tarefas simultâneas dentro de um processo, útil para tarefas de I/O.

Singleton: Padrão que garante uma única instância de uma classe.

Decorators: Modifica o comportamento de funções ou métodos sem alterar o código original.

Context Managers: Gerencia recursos (como arquivos ou conexões) de forma automática com `with`.

Metaclasses: Define o comportamento de classes, permitindo controle sobre a criação de classes.

Generators: Funções que produzem uma sequência de valores de forma eficiente, utilizando `yield`.

Asyncio: Permite execução assíncrona, ideal para operações I/O que não bloqueiam o programa.

Comprehensions: Sintaxe concisa para criar listas, dicionários ou conjuntos.

Iteráveis *

Em Python, um **iterável** é um objeto que pode ser percorrido em um loop, como um `for`. Iteráveis são objetos que implementam o método especial `__iter__()`, que retorna um iterador. Um iterador, por sua vez, implementa o método `__next__()`, que retorna o próximo item do iterável ou levanta uma exceção `StopIteration` quando não há mais itens.

Conceitos importantes sobre iteráveis:

- **Iteráveis:** São objetos que suportam iteração e possuem um método `__iter__()` que retorna um iterador. Exemplos comuns de iteráveis incluem listas, tuplas, dicionários, conjuntos e strings.
- **Iteradores:** São objetos que realizam a iteração sobre um iterável. Eles implementam o método `__next__()` e, em Python 3, também podem implementar o método `__iter__()` que retorna o próprio iterador.

Exemplo de Iterável: Um exemplo clássico é uma lista. Quando você faz um loop sobre uma lista, está utilizando um iterador interno que percorre os elementos da lista.

```
# Exemplo de um iterável (uma lista)
    numbers = [1, 2, 3, 4, 5]

    for number in numbers:
        print(number)
```

Implementando um Iterável Personalizado: Você pode criar seu próprio iterável definindo uma classe que implementa o método `__iter__()` e retorna um iterador. Aqui está um exemplo simples:

```
class MyIterable:
        def __init__(self, limit):
            self.limit = limit

        def __iter__(self):
            self.current = 0
            return self

        def __next__(self):
            if self.current < self.limit:
                self.current += 1
                return self.current - 1
            else:
                raise StopIteration

    # Usando o iterável personalizado
    for num in MyIterable(5):
        print(num)
```

Explicação do Exemplo:

- **Classe MyIterable:** Define um iterável que produz números de 0 até `limit - 1`.
- **Método `__iter__`:** Inicializa o iterador e retorna o próprio objeto.
- **Método `__next__`:** Retorna o próximo número e levanta `StopIteration` quando todos os números forem retornados.

Dessa forma, você pode criar objetos personalizados que podem ser percorridos em loops, o que é muito útil para criar coleções de dados que se comportam como iteráveis.