Propriedades

Programação Orientada a Objetos

Python

Prof. Diógenes Furlan

Sumário

- Decoradores
- Propriedades
- Como usar @property para definir getters, setters e deleters.

DECORADORES

Decoradores / Anotações

- **Decorators** são funções que adicionam funcionalidades a outras funções ou métodos sem alterá-los permanentemente.
  - Eles "embrulham" a função original, permitindo executar código antes ou depois dela, e então retornam uma nova função com comportamento modificado.

- Inicia com @
- Colocado antes de nome de métodos

Exemplo 1 – Antes

- Decorador sem utilidade

```
def minha_função():
    print("A função original está sendo executada.")

minha_função()
```

```
SAÍDA
=====
A função original está sendo executada.
```

Exemplo 1 – Depois

- Decorador sem utilidade

```
def meu_decorador(funcao):
    def função_modificada():
        print("Algo antes da execução da função original.")
        funcao()
        print("Algo após a execução da função original.")
    return função_modificada

@meu_decorador
def minha_função():
    print("A função original está sendo executada.")

minha_função()
```

```
SAÍDA
=====
Algo antes da execução da função original.
A função original está sendo executada.
Algo após a execução da função original.
```

Aplicando Decoradores no Python

```
@decorador3
@decorador2
@decorador1
def minha_funcao():
    pass
```

- Decorador1 é aplicado primeiro

Benefícios dos Decoradores

- **Reusabilidade de Código**: Decoradores permitem reutilizar lógica comum a várias funções, mantendo o código DRY (Don't Repeat Yourself).

- **Separação de Preocupações**: Eles ajudam a separar as preocupações, mantendo a lógica de negócios separada da lógica de controle ou de monitoramento.

- **Sintaxe Clara e Expressiva**: A sintaxe dos decoradores (@) torna o código mais legível e expressivo, facilitando o entendimento de que uma função está sendo modificada ou aprimorada.

Exemplo 2 – Antes

- Logando as tarefas

```
def soma(a, b):
    return a + b

print(soma(2,3))
```

```
SAÍDA
=====
5
```

Exemplo 2 – Depois

- Logando as tarefas

```
def logging(func):
    def wrapper(*args, **kwargs):
        print(f"Nome da função: {func.__name__}")
        print(f"Argumentos: {args}")
        resultado = func(*args, **kwargs)
        print(f"Resultado: {resultado}")
        return resultado
    return wrapper

@logging
def soma(a, b):
    return a + b

soma(2,3)
```

```
SAÍDA
=====
Nome da função: soma
Argumentos: (2, 3)
Resultado: 5
```

Exemplo 3 – Antes

- Medição de desempenho

```
def operação_demorada(n):
    soma = 0
    for i in range(n):
        soma += i
    return soma

print( operação_demorada(1000) )
```

```
SAÍDA
=====
499500
```

Exemplo 3 – Depois

- Medição de desempenho

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = func(*args, **kwargs)
        fim = time.time()
        print(f"A função {func.__name__} \
                levou {fim-inicio:.4f} seg para executar.")
        return resultado
    return wrapper

@timer
def operação_demorada(n):
    soma = 0
    for i in range(n):
        soma += i
    return soma

print( operação_demorada(1000) )
```

```
SAÍDA
=====
A função operação_demorada
levou 0.0000 segundos para executar.
499500
```

Exemplo 4 – Antes

- Verificação de Login

```
def delete_database():
    print("Banco de dados deletado com sucesso.")

delete_database()
```

```
SAÍDA
=====
Banco de dados deletado com sucesso.
```

Exemplo 4 – Depois

- Verificação de Login

```
def check_permissão(user):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if user != 'admin':
                raise Exception("Acesso negado. Usuário não tem permissão.")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@check_permissão(user="admin")
def delete_database():
    print("Banco de dados deletado com sucesso.")

@check_permissão(user="guest")
def delete_database():
    print("Banco de dados deletado com sucesso.")

delete_database()
```

Exemplo 5 – Antes

- Cache de resultados

```
def fibonacci(n):
    if n < 2:
        return n
    print(f'Chamada de fib {n}')
    return fibonacci(n-1) + fibonacci(n-2)

fibonacci(5)
fibonacci(8)
```

```
SAÍDA
=====
Chamada de fib 5
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
Chamada de fib 3
Chamada de fib 2
```

```
SAÍDA
=====
Chamada de fib 8
Chamada de fib 7
Chamada de fib 6
Chamada de fib 5
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
Chamada de fib 3
Chamada de fib 2
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
Chamada de fib 5
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
Chamada de fib 3
Chamada de fib 2
Chamada de fib 6
Chamada de fib 5
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
Chamada de fib 3
Chamada de fib 2
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 2
```

Exemplo 5 – Depois

- Cache de resultados

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fibonacci(n):
    if n < 2:
        return n
    print(f'Chamada de fib {n}')
    return fibonacci(n-1) + fibonacci(n-2)

fibonacci(5)
fibonacci(8)
```

```
SAÍDA
=====
Chamada de fib 5
Chamada de fib 4
Chamada de fib 3
Chamada de fib 2
Chamada de fib 8
Chamada de fib 7
Chamada de fib 6
```

PROPRIEDADES

Classe sem encapsulamento

```
class Produto1:
    def __init__(self, preço):
        self.preço = preço

# main
obj = Produto1(500)
print(obj.preço)
obj.preço = 1000
print(obj.preço)
```

Classe com Encapsulamento

- Bom para códigos novos
- Sintaxe das chamadas é “deselegante"

```
class Produto2:
    def __init__(self, preço):
        self.__preço = preço

    def getPreço(self):
        return self.__preço

    def setPreço(self, valor):
        self.__preço = valor
```

```
# main
obj = Produto2(500)
print(obj.getPreço())
obj.setPreço(1000)
print(obj.getPreço())
```

Propriedades

- Você pode definir **três métodos** para uma propriedade:
  - Um **getter** - para acessar o valor do atributo.
    - Seletor
  - Um **setter** - para definir o valor do atributo.
    - Modificador
  - Um **deleter** - para excluir o atributo de instância.
    - Destrutor

Decoradores utilizados

- **@property:** Usado para indicar que vamos definir uma propriedade.
  - é usado sobre o método getter do atributo

- **@preço.setter:** Usado para indicar que esse é o método *setter* para a propriedade *preço*.

- **@preco.deleter:** Usado para indicar que esse é o método *deleter* para a propriedade *preço*.

Classe com Propriedades

```
class Produto3:
    def __init__(self, preço):
        self._preço = preço

    @property
    def preço(self):
        return self._preço

    @preço.setter
    def preço(self, novo):
        if novo > 0 and isinstance(novo, float):
            self._preço = novo
        else:
            print("Insira um preço válido (positivo)")

    @preço.deleter
    def preço(self):
        del self._preço
```

```
# main
obj = Produto3(500)
print(obj.preço)
obj.preço = -30.0
print(obj.preço)
```

EXERCÍCIOS

Exercício 1

- Utilize propriedades quando necessário

CONTA
- nome: string - valor: float - limite: float
+ depósito(valor) + saque(valor) + saldo() + transferência(valor, Conta)

Exercício 2

- Defina um decorador para a classe Caixa (aula passada).