Aula 10

Decoradores

Em Python, decoradores são uma forma de modificar o comportamento de uma função ou classe sem a necessidade de alterar seu código interno. Eles permitem adicionar funcionalidades extras a uma função ou classe de maneira transparente, tornando o código mais modular e flexível.

Um decorador é uma função que recebe outra função como argumento e retorna uma nova função, geralmente com modificações. Isso é possível porque, em Python, as funções são tratadas como objetos de primeira classe, ou seja, podem ser passadas como argumentos, retornadas por outras funções e atribuídas a variáveis.

Mas antes de mergulharmos mais a fundo no assunto, vamos voltar um pouco às funções.

Funções

Antes de entender os decoradores, é necessário primeiro entender como as funções funcionam. Para melhor compreensão, vamos usar uma função retorna um valor com base nos argumentos fornecidos. Aqui está um exemplo muito simples de uma função:

```python
def soma_cinco(valor: int) -> int:
    """recebe um valor e retorna o
        valor somado a 5"""
    return valor + 5

if __name__ == '__main__':
    print(f'soma_cinco(10) = {soma_cinco(10)}')
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
soma_cinco(10) = 15
PS C:\Users\gutoh\OneDrive\0.Python>
```

Em geral, as funções em Python também podem ter efeitos colaterais, em vez de apenas transformar uma entrada em uma saída. A função print() é um exemplo básico disso: ela retorna None enquanto tem o efeito colateral de imprimir algo no console.
Veja isso aplicado na função que criamos anteriormente:

```python
def soma_cinco(valor: int) -> None:
    """recebe um valor e imprime o
        valor somado a 5"""
    print(f'valor + 5 = {valor + 5}')

if __name__ == '__main__':
    print(f'soma_cinco(10) = {soma_cinco(10)}')
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
valor + 5 = 15
soma_cinco(10) = None
PS C:\Users\gutoh\OneDrive\0.Python>
```

Como visto ver, agora o retorno da função é None.

Objetos de Primeira Classe

Em Python, funções são objetos de primeira classe. Isso significa que as funções podem ser passadas e usadas como argumentos, assim como qualquer outro objeto (string, int, float, lista e assim por diante). Considere as três seguintes funções:

```python
def ola(nome: str) -> str:
    """sauda alguém"""
    return f'Olá, {nome}'

def tchau(nome: str) -> str:
    """se despede de alguém"""
    return f'Tchau, {nome}'

def sauda_tom(func_sauda) -> Any:
    """cria a função de acordo
        com a chamada"""
    return func_sauda('Tom')

if __name__ == '__main__':
    print(f'sauda_tom(ola) = {sauda_tom(ola)}')
    print(f'sauda_tom(tchau) = {sauda_tom(tchau)}')
```

Aqui, ola() e tchau() são funções regulares que esperam um nome fornecido como uma string. No entanto, a função sauda_tom() espera uma função como seu argumento. Podemos, por exemplo, passar a função ola() ou a função tchau():

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
sauda_tom(ola) = Olá, Tom
sauda_tom(tchau) = Tchau, Tom
PS C:\Users\gutoh\OneDrive\0.Python>
```

Note que sauda_tom(ola) se refere a duas funções, mas de maneiras diferentes: sauda_tom() e ola. A função ola é nomeada sem parênteses. Isso significa que apenas uma referência à função é passada. A função não é executada. A função sauda_tom(), por outro lado, é escrita com parênteses, então ela será chamada como de costume.

Funções Internas

É possível definir funções dentro de outras funções. Essas funções são chamadas de funções internas (inner functions). Aqui está um exemplo de uma função com duas funções internas:

```python
def pai() -> None:
    """função pai"""
    print('Texto da função pai().')

    def primeiro_filho() -> None:
        """função primeiro_filho"""
        print('Texto da função primeiro_filho().')

    def segundo_filho() -> None:
        """função segundo_filho"""
        print('Texto da função segundo_filho().')

    segundo_filho()
    primeiro_filho()

if __name__ == '__main__':
    pai()
```

O que acontece quando você chama a função pai()? Pense sobre isso por um momento. A saída será a seguinte:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Texto da função pai().
Texto da função segundo_filho().
Texto da função primeiro_filho().
PS C:\Users\gutoh\OneDrive\0.Python>
```

Observe que a ordem em que as funções internas são definidas não importa. Como em qualquer outra função, a impressão só ocorre quando as funções internas são executadas.

Além disso, as funções internas não são definidas até que a função pai seja chamada. Elas têm escopo local em pai(): elas só existem dentro da função pai() como variáveis locais. Tente chamar primeiro_filho(). Você deverá receber um erro:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Traceback (most recent call last):
  File "C:\Users\gutoh\OneDrive\0.Python\main.py", line
21, in <module>
    primeiro_filho()
    ^^^^^^^^^^^^^^
NameError: name 'primeiro_filho' is not defined
PS C:\Users\gutoh\OneDrive\0.Python>
```

Sempre que você chama pai(), as funções internas primeiro_filho() e segundo_filho() também são chamadas. Mas, devido ao seu escopo local, elas não estão disponíveis fora da função pai().

Retornando Funções de Funções

Python também permite que use funções como valores de retorno. O exemplo a seguir retorna uma das funções internas da função externa pai():

```python
def pai(opcao: bool) -> Callable[[], str]:
    """função pai"""
    def primeiro_filho() -> str:
        """função primeiro_filho"""
        return 'Texto da função primeiro_filho().'

    def segundo_filho() -> str:
        """função segundo_filho"""
        return 'Texto da função segundo_filho().'

    if opcao:
        return primeiro_filho
    return segundo_filho

if __name__ == '__main__':
    primeiro: Callable[[], str] = pai(True)
    segundo: Callable[[], str] = pai(False)

    print(primeiro)
    print(segundo)
```

Observe que está retornando primeiro_filho sem os parênteses. Lembre-se de que isso significa que está retornando uma referência à função primeiro_filho. Em contraste, primeiro_filho() com parênteses se refere ao resultado da execução da função. Isso pode ser visto no exemplo a seguir:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
<function pai.<locals>.primeiro_filho at 0x000002497519D8A0>
<function pai.<locals>.segundo_filho at 0x000002497519D9E0>
PS C:\Users\gutoh\OneDrive\0.Python>
```

A saída um tanto enigmática significa simplesmente que a primeira variável se refere à função local primeiro_filho() dentro de pai(), enquanto a segunda aponta para segundo_filho().

Agora podemos usar primeiro e segundo como se fossem funções regulares, mesmo que as funções para as quais eles apontam não possam ser acessadas diretamente:

```python
if __name__ == '__main__':
    primeiro: Callable[[], str] = pai(True)
    segundo: Callable[[], str] = pai(False)

    print(primeiro)
    print(segundo)

    print(primeiro())
    print(segundo())
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
<function pai.<locals>.primeiro_filho at 0x0000021700B8C9A0>
<function pai.<locals>.segundo_filho at 0x0000021700B8D940>
Texto da função primeiro_filho().
Texto da função segundo_filho().
PS C:\Users\gutoh\OneDrive\0.Python>
```

Por fim, observe que no exemplo anterior foi executado as funções internas dentro da função pai, por exemplo primeiro_filho(). No exemplo seguinte, não foi adicionado parênteses às funções internas (primeiro_filho) ao retorná-las. Dessa forma, você obteve uma referência a cada função que pode ser chamada no futuro. E no último exemplo, as funções são executadas ao serem usadas com os parênteses. Faz sentido?

Decorador Simples

Agora que viu que as funções são apenas como qualquer outro objeto em Python, está pronto para avançar e conhecer a criatura mágica que é o decorador em Python. Vamos começar com um exemplo:

```python
def meu_decorador(func) -> Callable[[], None]:
    """função que servirá de decorador"""
    def interna() -> None:
        """função interna que vai executar func"""
        print('Antes de executar func.')
        func()
        print('Depois de executar func.')

    return interna

def frase() -> None:
    """função frase"""
    print('Poder Imensurável!')
```

```
if __name__ == '__main__':
    frase = meu_decorador(frase)
    frase()
```

Consegue adivinhar o que acontece quando chama frase()? Experimente:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Antes de executar func.
Poder Imensurável!
Depois de executar func.
PS C:\Users\gutoh\OneDrive\0.Python>
```

Para entender o que está acontecendo aqui, volte aos exemplos anteriores. Estamos literalmente aplicando tudo o que aprendemos até agora.

A chamada de "decoração" acontece na seguinte linha:

```
if __name__ == '__main__':
    frase = meu_decorador(frase)
    frase()
```

Na verdade, o nome "frase" agora aponta para a função interna "interna()". Lembre-se de que você retorna "interna" como uma função quando chama "meu_decorador(frase) ":

```
if __name__ == '__main__':
    frase = meu_decorador(frase)
    frase()
    print(frase)
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Antes de executar func.
Poder Imensurável!
Depois de executar func.
<function meu_decorador.<locals>.interna at 0x000002415D8CD8A0>
PS C:\Users\gutoh\OneDrive\0.Python>
```

No entanto, interna() tem uma referência para a função frase() original como func e chama essa função entre as duas chamadas de print().

Em resumo: decoradores envolvem (encapsulam) uma função, modificando seu comportamento.

Antes de prosseguirmos, vamos dar uma olhada em um segundo exemplo.
Geralmente, essa função interna costuma se chamar de wrapper (assim como o self é usado nas classes). Como wrapper() é uma função normal em Python, a forma como um decorador modifica uma função pode mudar dinamicamente.

Para não incomodar seus vizinhos, o exemplo a seguir só executará o código decorado durante o dia:

```python
from datetime import datetime

def respeirando_horario_silencio(func):
    """função que respeira o horário de
    silêncio de um condomínio"""
    def wrapper():
        """função interna chamada de wrapper"""
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # respeitando os vizinhos
    return wrapper

def faz_barulho():
    """toca Iron Maiden"""
    print('Tocando Run To The Hills.')

if __name__ == '__main__':
    faz_barulho = respeirando_horario_silencio(faz_barulho)
    faz_barulho()
    print(faz_barulho)
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills.
<function respeirando_horario_silencio.<locals>.wrapper at 0x000002442EE5C860>
PS C:\Users\gutoh\OneDrive\0.Python>
```

É Bom, Mas Pode Melhorar

A maneira como decoramos a função faz_barulho() acima é um pouco complicada. Em primeiro lugar, acabamos digitando o nome faz_barulho três vezes. Além disso, a decoração acaba um pouco escondida abaixo da definição da função.

Em vez de usar o decorador dessa forma, o Python permite que usemos decoradores de uma maneira muito mais simples com o símbolo @, às vezes chamado de "pie" syntax (sintaxe de torta).
Veja mais em:

https://peps.python.org/pep-0318/#background

O exemplo a seguir faz exatamente a mesma coisa que o primeiro exemplo anterior:

```python
from datetime import datetime

def respeirando_horario_silencio(func):
    """funcao decoradora"""
    def wrapper():
        """função interna chamada de wrapper"""
        if 7 ≤ datetime.now().hour < 22:
            func()
        else:
            pass  # respeitando os vizinhos
    return wrapper

@respeirando_horario_silencio
def faz_barulho():
    """toca Iron Maiden"""
    print('Tocando Run To The Hills.')

if __name__ == '__main__':
    faz_barulho()
    print(faz_barulho)
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills.
<function respeirando_horario_silencio.<locals>.wrapper at 0x000001C129C4C860>
PS C:\Users\gutoh\OneDrive\0.Python>
```

Então, @respeirando_horario_silencio é apenas uma maneira mais fácil de dizer
faz_barulho = respeitando_horario_silencio(faz_barulho).
É como aplicamos um decorador a uma função.

Reutilizando Decoradores

Lembre-se de que um decorador é apenas uma função normal em Python. Todas as ferramentas usuais para facilitar a reutilização estão disponíveis. Vamos mover o decorador para seu próprio módulo, que pode ser usado em muitas outras funções.

Criamos um arquivo chamado decoradores.py com o seguinte conteúdo:

```python
"""módulo decoradores"""

def faz_duas_vezes(func):
    """função faz_duas_vezes"""
    def wrapper_faz_duas_vezes():
        """função wrapper"""
        func()
        func()
    return wrapper_faz_duas_vezes
```

PS: podemos nomear a função interna como quisermos, e um nome genérico como wrapper() geralmente está tudo bem. Como teremos muitos decoradores, para diferenciá-los, nomearemos a função interna com o mesmo nome do decorador, mas com um prefixo wrapper_.

Agora podemos usar esse novo decorador em outros arquivos utilizando um import:

```python
"""módulo main.py"""
from decoradores import faz_duas_vezes

@faz_duas_vezes
def faz_barulho():
    """toca Iron Maiden"""
    print('Tocando Run To The Hills.')

if __name__ == '__main__':
    faz_barulho()
```

Quando executarmos este exemplo, veremos que o faz_barulho() original é executado duas vezes:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills.
Tocando Run To The Hills.
PS C:\Users\gutoh\OneDrive\0.Python>
```

Função com Argumentos nos Decoradores

Digamos que agora temos uma função que aceita alguns argumentos. Ainda podemos decorar ela? Vejamos:

```python
"""módulo main.py"""
from decoradores import faz_duas_vezes

@faz_duas_vezes
def faz_barulho(nome):
    """toca Iron Maiden"""
    print(f'Tocando Run To The Hills do {nome}')

if __name__ == '__main__':
    faz_barulho('Gandalf')
```

Conforme esperado, isso irá gerar um TypeError.

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Traceback (most recent call last):
  File "C:\Users\gutoh\OneDrive\0.Python\main.py", line 10, in <module>
    faz_barulho('Gandalf')
TypeError: faz_duas_vezes.<locals>.wrapper_faz_duas_vezes() takes 0 posi
tional arguments but 1 was given
PS C:\Users\gutoh\OneDrive\0.Python>
```

O problema é que a função interna wrapper_faz_duas_vezes () não recebe nenhum argumento, mas nome="Gandalf" foi passado para ela. Você poderia corrigir isso permitindo que wrapper_faz_duas_vezes() aceite um argumento, mas então ele não funcionaria para a função frase() (página 4) que criamos anteriormente.

A solução é usar *args e **kwargs na função interna wrapper. Dessa forma, ela aceitará um número arbitrário de argumentos posicionais e argumentos de palavras-chave.
Veja como fica o arquivo decoradores.py:

```python
"""módulo decoradores"""

def faz_duas_vezes(func):
    """função faz_duas_vezes"""
    def wrapper_faz_duas_vezes(*args, **kwargs):
        """função wrapper"""
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_faz_duas_vezes
```

A função interna wrapper_faz_duas_vezes() agora aceita qualquer número de argumentos e os repassa para a função que ela decora. Agora, tanto os exemplos faz_barulho() quanto frase() funcionam:

```python
"""módulo main.py"""
from decoradores import faz_duas_vezes

@faz_duas_vezes
def faz_barulho(nome):
    """toca Iron Maiden"""
    print(f'Tocando Run To The Hills do {nome}')

@faz_duas_vezes
def frase():
    """função frase"""
    print('Poder Imensurável!')

if __name__ == '__main__':
    faz_barulho('Gandalf')
    frase()
```

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills do Gandalf
Tocando Run To The Hills do Gandalf
Poder Imensurável!
Poder Imensurável!
PS C:\Users\gutoh\OneDrive\0.Python>
```

Retornando Valores de Funções Decoradoras

O que acontece com o valor de retorno das funções decoradas? Bem, isso depende de o decorador decidir. Digamos que você decore uma função simples da seguinte forma:

```python
"""módulo main.py"""
from decoradores import faz_duas_vezes

@faz_duas_vezes
def faz_barulho(nome):
    """toca Iron Maiden"""
    print(f'Tocando Run To The Hills do {nome}')
    return f'Música do {nome}.'

if __name__ == '__main__':
    resultado = faz_barulho('Gandalf')
    print(resultado)
```

Vejamos o que acontece:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills do Gandalf
Tocando Run To The Hills do Gandalf
None
PS C:\Users\gutoh\OneDrive\0.Python>
```

Oops, o decorador "comeu" o valor de retorno da função.

Como a função wrapper_faz_duas_vezes () não retorna explicitamente um valor, a chamada faz_barulho('Gandalf') acabou retornando None.

Para corrigir isso, precisamos garantir que a função wrapper_* retorne o valor de retorno da função decorada. Vejamos como fica o arquivo decorators.py com essa coreção:

```python
"""módulo decoradores"""

def faz_duas_vezes(func):
    """função faz_duas_vezes"""
    def wrapper_faz_duas_vezes(*args, **kwargs):
        """função wrapper"""
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_faz_duas_vezes
```

Agora, o valor de retorno da última execução da função é retornado:

```
PS C:\Users\gutoh\OneDrive\0.Python> python main.py
Tocando Run To The Hills do Gandalf
Tocando Run To The Hills do Gandalf
Música do Gandalf.
PS C:\Users\gutoh\OneDrive\0.Python>
```

Exercícios para Praticar

1. Crie um decorador simples que imprima uma mensagem antes e depois da execução de uma função.
2. Escreva um decorador que calcule o tempo de execução de uma função.
3. Implemente um decorador que limite o número de vezes que uma função pode ser chamada.
4. Crie um decorador que registre o histórico de chamadas de uma função.
5. Escreva um decorador que faça cache dos resultados de uma função para entradas específicas.
6. Implemente um decorador que permita apenas a execução de uma função em dias úteis.
7. Crie um decorador que verifique se os argumentos de uma função são do tipo correto.
8. Escreva um decorador que restringe o acesso a uma função com base em permissões de usuário.
9. Implemente um decorador que adicione um cabeçalho HTML a uma string retornada por uma função.
10. Crie um decorador que transforme o resultado de uma função em maiúsculas.
11. Escreva um decorador que altere o comportamento de uma função dependendo do ambiente de execução (produção, desenvolvimento, teste).
12. Implemente um decorador que adicione uma contagem de chamadas a uma função.
13. Crie um decorador que converta os argumentos de uma função para um tipo específico.
14. Escreva um decorador que salve o resultado de uma função em um arquivo.
15. Implemente um decorador que adicione um prefixo a todas as mensagens impressas por uma função.
16. Crie um decorador que permita que uma função seja executada apenas se um arquivo específico existir.
17. Escreva um decorador que adicione uma validação de senha a uma função.
18. Implemente um decorador que adicione um log de erros a uma função.
19. Crie um decorador que imprima o número total de argumentos de uma função antes de sua execução.
20. Escreva um decorador que envie um e-mail com o resultado de uma função.
21. Implemente um decorador que verifique se uma função retorna um valor específico.
22. Crie um decorador que registre o tempo de execução médio de uma função ao longo de várias chamadas.
23. Escreva um decorador que permita que uma função seja chamada apenas uma vez.
24. Crie um decorador que converta o resultado de uma função para um tipo de dados específico.
25. Escreva um decorador que adicione um log de depuração a uma função.
26. Implemente um decorador que limite o tempo máximo de execução de uma função.
27. Crie um decorador que adicione um número de versão a uma função.
28. Escreva um decorador que adicione um identificador único a cada chamada de uma função.
29. Implemente um decorador que registre o número de exceções lançadas por uma função.
30. Crie um decorador que implemente uma autenticação simples para uma função.
31. Escreva um decorador que adicione um cabeçalho JSON a uma resposta de API.
32. Implemente um decorador que adicione um rodapé HTML a uma string retornada por uma função.
33. Crie um decorador que permita que uma função seja chamada apenas se uma determinada condição for atendida.
34. Escreva um decorador que adicione um temporizador a uma função.
35. Implemente um decorador que adicione um log de transações a uma função.
36. Crie um decorador que converta os argumentos de uma função para um formato específico (por exemplo, JSON).
37. Escreva um decorador que adicione uma mensagem de erro personalizada a uma exceção lançada por uma função.
38. Implemente um decorador que adicione um número de linha e um arquivo de origem a uma mensagem de log.
39. Crie um decorador que adicione um aviso de depreciação a uma função.
40. Escreva um decorador que adicione um código de retorno a uma função.
41. Implemente um decorador que transforme todos os argumentos de uma função em letras minúsculas.
42. Crie um decorador que adicione um log de depuração com informações do arquivo de origem e da linha de código.
43. Escreva um decorador que adicione um prefixo a todos os argumentos de uma função.
44. Implemente um decorador que adicione um limite máximo de tamanho para uma string retornada por uma função.
45. Crie um decorador que adicione um log de exceções capturadas por uma função.
46. Escreva um decorador que adicione uma mensagem de log antes e depois da execução de uma função.
47. Implemente um decorador que adicione um contador de execuções a uma função.
48. Escreva um decorador que permita que uma função seja chamada apenas se um determinado arquivo não existir.