PYTHON ACADEMY

APRENDA COMO UTILIZAR DECORATORS NO PYTHON

Decorators modificam o comportamente do código antes e depois da execução de uma dada função, sem a necessidade de modificar o próprio código da mesma, adicionando funcionalidades.

Olá pessoal. Nesse post, iremos descobrir e desvendar alguns mistérios por trás dos poderosos *Decorators* em Python.

O conceito de *decorator* provê uma maneira simples de modificar o comportamento de uma função sem necessariamente alterá-la.

Ficou confuso? Então vamos começar com alguns conceitos básicos. Eu garanto que tudo ficará mais claro até o fim do texto! Primeiro, você sabe o que é uma função?

Funções

Funções são trechos de código que recebem parâmetros, realizam um conjunto de operações e então retornam algum valor ou conjunto de valores. Abaixo uma simples função de soma:

```
def sum(num_1, num_2):
    return (num_1 + num_2)
```

Em Python, funções são **objetos de primeira classe**.

Mas o que significa ser um Objeto de Primeira Classe?

Significa que funções podem ser passadas como parâmetro, utilizadas como retorno de funções, assim como qualquer outro valor (string, int, float).

Se bem utilizada, essa característica pode ser bem útil e poderosa!

Nested Functions

Por conta de sua característica de serem objetos de primeira classe, é possível definirmos funções dentro de outras funções. Esse é o conceito de *nested functions*. Abaixo um trecho de código exemplificando:

```
def party():
    print("Estou de fora =[")

    def start_party():
        return "Estamos dentro! Uhullll!"

    def finish_party():
        return "A festa acabou! =["

    print(start_party())
    print(finish_party())
```

Dessa forma, caso você chame a função `party()`, sua saída será:

```
Estou de fora =[
Estamos dentro! Uhullll!
A festa acabou! =[
```

E o que acontece caso eu tente executar a função `start_party()` ou `finish_party()`?

O mais óbvio é que não seja possível executá-las, certo?! Bom... É exatamente o que acontece! O seguinte erro aparecerá caso tente:

```
Traceback (most recent call last):
File "decorator.py", line 10, in <module>
start_party()
NameError: name 'start_party' is not defined
```

Portanto, tudo se resume ao escopo da função.

Isto é, as funções `start_party()` e `finish_party()` estão limitadas pelo escopo da função `party()`.

Por isso não conseguimos chamá-las fora desse escopo, apenas quando pedimos à função `party()` para executá-las para nós.

Utilizando Funções como Retorno de Outras Funções

Como disse anteriormente, funções são objetos de primeira classe em Python. Assim, nada nos impede de utilizar uma função como retorno de outra. Veja o exemplo abaixo:

```
# "Criador" de funções de potência
def cria_potencia(x):
    def potencia(num):
        return x ** num
    return potencia

# Potência de 2 e 3
potencia_2 = cria_potencia(2)
potencia_3 = cria_potencia(3)

# Resultado
print(potencia_2(2))
print(potencia_3(2))
```

A saída dos dois comandos é:

```
>>> 4   # 2 ** 2 = 4
>>> 9   # 3 ** 2 = 9
```

Com isso em mente, vamos finalmente conversar sobre o ator principal desse post: o ***Decorator***!

Decorators

Primeiro, vamos dar uma olhada na PEP 318 (***P**ython **E**nhancement **P**roposal* - Proposta de melhoria na linguagem Python) que propôs a adição dos *decorators* ao Python. Abaixo está transcrita uma breve descrição da proposta que o define (traduzido com alterações):

> *O método atual para transformar funções e métodos (por exemplo, declarando-os como classes ou métodos estáticos) é complicado e pode levar a código que é difícil de entender. Idealmente, essas transformações devem ser feitas no mesmo ponto do código onde a própria declaração é feita. Esta PEP introduz uma nova sintaxe para transformações de uma função ou declaração de métodos.*

Daí nasceu o *decorator*, que nada mais é que um **método para envolver uma função, modificando seu comportamento**.

Para explicar melhor, veja o código abaixo:

```
def decorator(funcao):
    def wrapper():
        print ("Estou antes da execução da função passada como argumen-
        to")
        funcao()
        print ("Estou depois da execução da função passada como argu-
        mento")

    return wrapper

def outra_funcao():
    print ("Sou um belo argumento!")

funcao_decorada = decorator(outra_funcao)
funcao_decorada()
```

A saída será:

```
>>> Estou antes da execução da função passada como argumento
>>> Sou um belo argumento
>>> Estou depois da execução da função passada como argumento
```

Dessa forma, conseguimos adicionar qualquer comportamento antes e depois da execução de uma função qualquer!

Vamos fazer agora um exemplo mais útil, algo que todo mundo que desenvolve software teve que fazer alguma vez vida: **calcular o tempo de execução de determinada função!**

```python
import time

# Define nosso decorator
def calcula_duracao(funcao):
    def wrapper():
        # Calcula o tempo de execução
        tempo_inicial = time.time()
        funcao()
        tempo_final = time.time()

        # Formata a mensagem que será mostrada na tela
        print("[{funcao}] Tempo total de execução: {tempo_total}".format(
            funcao=funcao.__name__,
            tempo_total=str(tempo_final - tempo_inicial))
        )

    return wrapper

# Decora a função com o decorator
@calcula_duracao
def main():
    for n in range(0, 10000000):
        pass

# Executa a função main
main()
```

Nossa função principal tem apenas um *loop* que não faz nada, apenas itera n de 0 à 10000000, o que dura um certo tempo.

Marcamos o tempo de início e de término da execução com o módulo `time` e então subtraímos o final pelo inicial, dando o tempo total de execuções.

A saída será (o tempo de execução depende da sua máquina):

```
[main] Tempo total de execução: 0.23434114456176758
```

Vamos ver agora um exemplo real de utilização de *Decorators*!

> 💡 *Estou construindo o **DevBook**, uma plataforma que usa IA para criar ebooks técnicos — com código formatado e exportação em PDF. Confere!*

Exemplo Real: Restringindo o Acesso em uma Aplicação Flask

Para quem não conhece, Flask é um framework web minimalista, muito simples de utilizar e bem divertido!

Bom, quando desenvolvemos sistemas para web, nós sempre nos preocupamos com o controle de acesso a determinadas páginas.

Por exemplo, não queremos que usuários não autorizados acessem a URL `/admin/`. Para isso, uma abordagem seria incluir a verificação de usuários no corpo de toda função que trata requisições. Exemplo (utilizando Flask):

```
@app.route('/admin')
def admin_index():
    # Verifica se session['logado'] já foi setado
    if ('logado' not in session):
        return redirect('index')

    # Caso usuário esteja logado, renderiza a página /admin/index.html
    return render_template('/admin/index.html')
```

O problema dessa abordagem é que teremos que repetir a operação de verificar se o usuário está logado em toda requisição para `/admin/*`.

Isso gera código duplicado em todas essas funções. Sem contar que qualquer erro no “copia e cola” desse código pode estar expondo uma URL para qualquer usuário. **Teeeenso**! 😯

Para evitar esse tipo de problema, podemos criar um *decorator* que verifica se o usuário que está requisitando aquela página já efetuou o *login* ou não da seguinte forma:

```python
# Decorator
def requer_autenticacao(f):
    @wraps(f)
    def funcao_decorada(*args, **kwargs):
        # Verifica session['logado']
        if ('logado' not in session):
            # Retorna para a URL de login caso o usuário não esteja lo-
        gado
            return redirect(url_for('index'))

        return f(*args, **kwargs)
    return funcao_decorada
```

Para utilizar esse *decorator*, precisamos apenas incluí-lo nas funções que queremos restringir o acesso. Exemplo:

```python
# Olha quem tá aqui... Outro decorator :D
@app.route('/admin/dashboard')
@requer_autenticacao
def admin_dashboard():
    # Renderiza o template dashboard.html
    return render_template('admin/dashboard.html')
```

Pronto! Com apenas uma linha adicional de código (verificada em tempo de compilação, portanto qualquer erro será acusado pelo Python) conseguimos adicionar uma nova funcionalidade ao nosso código.

E o decorator `@wraps(f)` na linha 3? O que faz?

Quando utilizamos um *decorator*, estamos substituindo uma função `X()` por outra função `Y()` que engloba a função `X()`.

E o que isso traz de problema?

Isso quer dizer que a nova função irá perder seus metadados (`__name__`, `__docstring__`, etc...). Esse não é um efeito que queremos que aconteça.

Por exemplo, caso você tente mostrar na tela qual o nome da função após ela ter sido decorada, `X.__name__` não irá apresentar seu nome original e sim o nome da função utilizada dentro do *decorator*.

Para evitar esse efeito colateral, utilizamos a função `functools.wraps`.

O que ela faz é copiar os metadados da função antes de ser decorada para sua versão decorada. Feito isso, utilizar um *decorator* não tira a identidade da sua função. Ela fica intacta!

Conclusão

Vimos neste *post* o poder dos *Decorators* e suas principais características.

Decorators adicionam uma maneira rápido de trabalhar com metaprogramação em Python. De um modo geral, a metaprogramação é toda programação que age sobre outro programa, seja em seu código fonte, binário, ou numa representação abstrata em memória.

Vimos também um exemplo real de utilização de *decoratos* juntamente com o framework web Flask para evitar que usuários não logados acessem uma URL indevida.

É isso pessoal! Espero que esse post tenha facilitado o seu entendimento sobre *decorators*! Qualquer dúvida, sugestão ou crítica, por favor não hesite em comentar aqui embaixo!

Vamos aprender juntos 👊

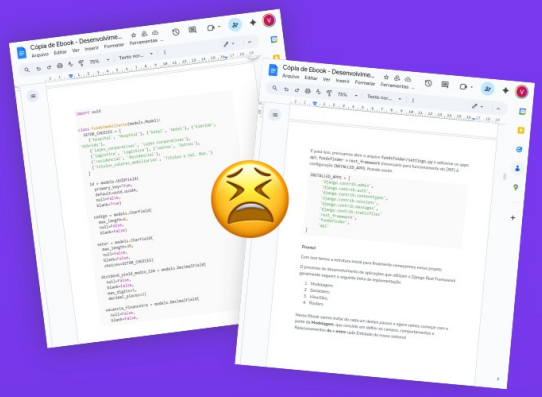
Até a próxima!

Não se esqueça de conferir!

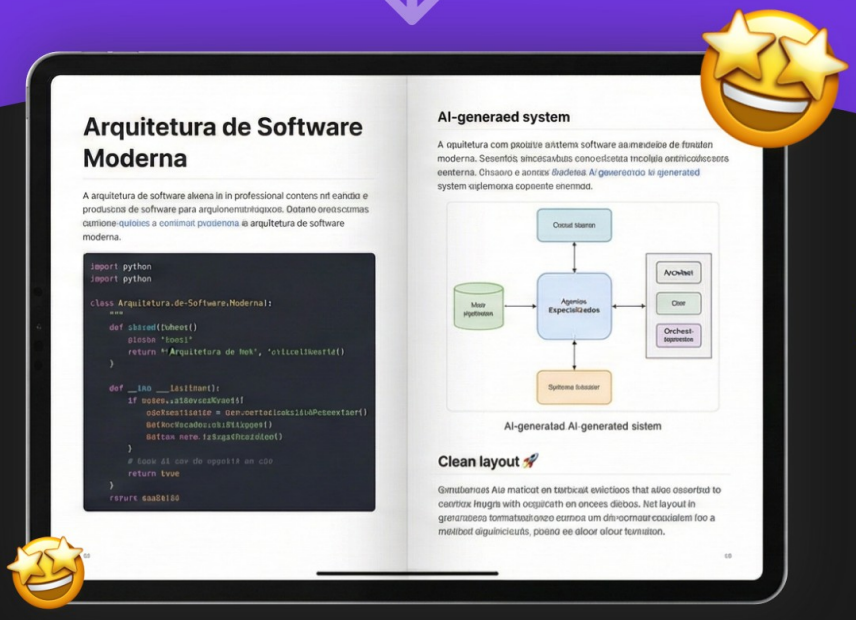
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS