PYTHON ACADEMY

# APRENDA COMO UTILIZAR DECORATORS NO PYTHON

Guia completo de Decorators em Python: conceito, sintaxe, nested functions, casos de uso reais (logging, timing, cache, autenticação, validação), @wraps e quando usar decorators.

PYTHONACADEMY.COM.BR

# Crie ebooks profissionais incríveis em minutos com IA


Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

**TESTE AGORA**

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025) Conteúdo enriquecido com casos de uso do mundo real (logging, timing, cache, autenticação) e boas práticas.*

Olá pessoal!

**Decorators** são uma das features mais elegantes do Python para metaprogramação. Eles permitem modificar o comportamento de funções ou classes de forma **limpa, reutilizável e Pythônica**, sem alterar o código original.

Neste guia, você vai aprender:

- ✅ Conceito de decorators e funções de primeira classe
- ✅ Sintaxe com `@decorator`
- ✅ **Casos de uso reais** (logging, timing, cache, autenticação)
- ✅ Decorators com parâmetros
- ✅ `@wraps` e preservação de metadados
- ✅ Quando usar e quando **NÃO** usar decorators

Ficou confuso? Vamos começar do básico. Eu garanto que tudo ficará mais claro até o fim! 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:

```python
# "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
        argumento")
        funcao()
        print ("Estou depois da execução da função passada como
        argumento")

    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!**

```
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*!

> 💡 *Criei o **Ebookr.ai**, uma plataforma que usa IA para gerar ebooks profissionais sobre qualquer tema — com capa gerada por IA, infográficos automáticos 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
        logado
            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.

# Casos de Uso do Mundo Real

Vamos ver exemplos práticos de decorators que você pode usar no dia a dia:

# 1. Logging Automático

```python
import functools
import logging

logging.basicConfig(level=logging.INFO)

def log_execution(func):
    """Decorator que loga entrada e saída de funções"""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        logging.info(f"Executando {func.__name__} com args={args},
        kwargs={kwargs}")
        result = func(*args, **kwargs)
        logging.info(f"{func.__name__} retornou: {result}")
        return result
    return wrapper

@log_execution
def calcular_preco(produto, quantidade, desconto=0):
    return produto['preco'] * quantidade * (1 - desconto)

# Uso
produto = {'nome': 'Notebook', 'preco': 2500}
total = calcular_preco(produto, 2, desconto=0.1)
# INFOExecutando calcular_preco com args=(...), kwargs={'desconto':
        0.1}
# INFOcalcular_preco retornou: 4500.0
```

## 2. Medir Tempo de Execução

```python
import functools
import time

def timing(func):
    """Decorator que mede tempo de execução"""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} levou {end - start:.4f}s")
        return result
    return wrapper

@timing
def processar_dados(n):
    return sum(i**2 for i in range(n))

resultado = processar_dados(1000000)
# processar_dados levou 0.1234s
```

## 3. Cache (Memoização)

```python
import functools

# Python 3.9+ tem @cache nativo!
from functools import cache

@cache  # Ou @functools.lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Sem cache: O(2^n) - muito lento
# Com cache: O(n) - super rápido!

print(fibonacci(100))  # Instantâneo!
# 354224848179261915075
```

# 4. Validação de Tipos

```python
import functools

def validate_types(*expected_types):
    """Decorator que valida tipos dos argumentos"""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for arg, expected_type in zip(args, expected_types):
                if not isinstance(arg, expected_type):
                    raise TypeError(
                        f"Esperado {expected_type.__name__}, "
                        f"recebido {type(arg).__name__}"
                    )
            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_types(str, int)
def criar_usuario(nome, idade):
    return {'nome': nome, 'idade': idade}

criar_usuario("Ana", 25)  # ✅ OK
criar_usuario("Ana", "25")  # ❌ TypeError!
```

## 5. Retry Automático

```python
import functools
import time

def retry(max_attempts=3, delay=1):
    """Decorator que reexecuta função em caso de erro"""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts:
                        raise
                    print(f"Tentativa {attempt} falhou: {e}.
        Retentando...")
                    time.sleep(delay)
        return wrapper
    return decorator

@retry(max_attempts=3, delay=2)
def fetch_api_data(url):
    import requests
    response = requests.get(url, timeout=5)
    response.raise_for_status()
    return response.json()
```

# Quando Usar Decorators?

✅ **Use decorators quando:**

1. **Cross-cutting concerns** - Logging, timing, autenticação
2. **Comportamento reutilizável** - Aplicar mesma lógica em várias funções
3. **DRY (Don't Repeat Yourself)** - Evitar duplicação de código

4. **Aspectos ortogonais** - Funcionalidade independente da lógica principal
5. **API design** - Frameworks (Flask, Django) usam extensivamente

❌ **Evite decorators quando:**

1. **Lógica complexa** - Dificulta debug e entendimento
2. **Estado mutante** - Decorators não devem modificar estado global
3. **Performance crítica** - Overhead de chamadas de função aninhadas
4. **Usado apenas uma vez** - Não justifica a complexidade
5. **Muito específico** - Se não é reutilizável, coloque dentro da função

***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

Neste guia completo sobre **Decorators**, você aprendeu:

✅ **Conceito fundamental** - Funções de primeira classe e nested functions ✅ **Sintaxe** `@decorator` - Forma Pythônica de aplicar decorators ✅ **Casos de uso reais** - Logging, timing, cache, validação, retry ✅ **Decorators com parâmetros** - Maior flexibilidade ✅ `@wraps` - Preservar metadados da função original ✅ **Quando usar** - Cross-cutting concerns e reutilização

**Principais lições:**

- Decorators são **funções que modificam outras funções**
- Use `@functools.wraps` para preservar metadados
- Ideais para **aspectos ortogonais** (logging, timing, auth)
- Python stdlib tem decorators prontos: `@cache`, `@property`, `@staticmethod`
- Frameworks (Flask, Django) usam decorators extensivamente

**Próximos passos:**

- Crie seus próprios decorators para logging e timing
- Explore `functools` para decorators avançados
- Estude decorators de classe ( `@dataclass` , `@property` )
- Combine múltiplos decorators (stacking)

Decorators são uma das ferramentas mais poderosas para **metaprogramação** em Python. Use-os para tornar seu código mais **limpo, reutilizável e Pythônico**!

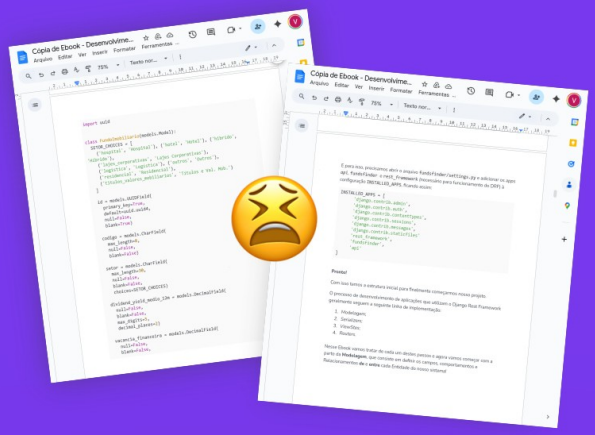
Vamos aprender juntos 👊

Até a próxima!

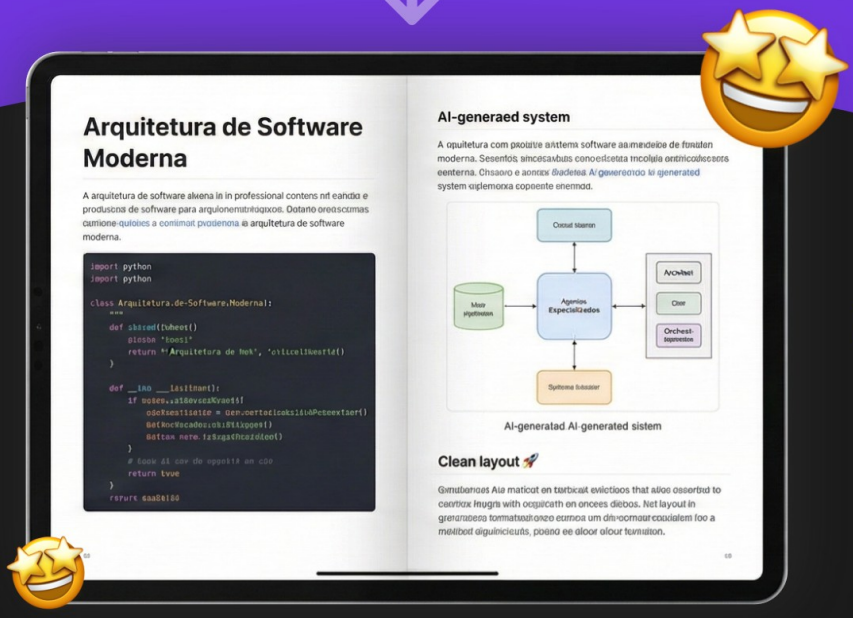
Não se esqueça de conferir!

Ebookr

# Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word




Capas gerados por IA

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