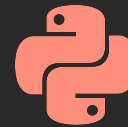




O que são @decoradores?

Live de Python # 213



1. Uma visão inicial dos decoradores

Um nivelamento inicial

2. HOFs

Funções como objeto de primeira classe

3. Closures

Um entendimento sobre contexto

4. Enfim, criando decoradores

Vimos aqui pra isso!



picpay.me/dunossauro



apoia.se/livedepython



pix.dunossauro@gmail.com



Ajude o projeto <3



Acássio Anjos, Ademar Peixoto, Adilson Herculano, Alexandre Harano, Alexandre Lima, Alexandre Souza, Alexandre Takahashi, Alexandre Villares, Alex Lima, Alynne Ferreira, Alysson Oliveira, Ana Carneiro, Ana Padovan, Andre Azevedo, André Rafael, André Rocha, Aquiles Coutinho, Arnaldo Turque, Aurelio Costa, Bruno Freitas, Bruno Guizi, Bruno Oliveira, Bruno Ramos, Caio Nascimento, Carlos Alipio, Christiano Moraes, Clara Battesini, Daniel Freitas, Daniel Haas, Danilo Segura, David Kwast, Delton Porfiro, Dhyeives Rodovalho, Diego Farias, Diego Guimarães, Dilenon Delfino, Dino Aguilar, Diogo Paschoal, Douglas Bastos, Douglas Braga, Douglas Zickuhr, Dutofanim Dutofanim, Emerson Rafael, Érico Andrei, Eugenio Mazzini, Euripedes Borges, Evandro Avellar, Everton Silva, Fabio Barros, Fábio Barros, Fabio Castro, Fábio Thomaz, Felipe Rodrigues, Fernanda Prado, Fernando Rozas, Flávio Meira, Flavkaze Flavkaze, Franklin Silva, Gabriel Barbosa, Gabriel Simonetto, Geandreson Costa, Guilherme Felitti, Guilherme Gall, Guilherme Ostrock, Guilherme Piccioni, Gustavo Dettenborn, Gustavo Suto, Heitor Fernandes, Henrique Junqueira, Hugo Cosme, Igor Taconi, Israel Gomes, Italo Silva, Jair Andrade, Jairo Lenfers, Janael Pinheiro, João Lugão, João Paulo, João Rodrigues, Joelson Sartori, Johnny Tardin, Jonatas Leon, Jonatas Oliveira, Jônatas Silva, José Gomes, Joseíto Júnior, Jose Mazolini, Juan Gutierrez, Juliana Machado, Júlio Gazeta, Julio Silva, Kaio Peixoto, Kaneson Alves, Leandro Miranda, Leonardo Mello, Leonardo Nazareth, Lucas Mello, Lucas Mendes, Lucas Oliveira, Lucas Polo, Lucas Teixeira, Lucas Valino, Luciano Silva, Luciano Teixeira, Luiz Carlos, Luiz Junior, Luiz Lima, Luiz Paula, Luiz Perciliano, Maicon Pantoja, Maiquel Leonel, Marcelino Pinheiro, Marcelo Matte, Márcio Martignoni, Marco Mello, Marcos Gomes, Marco Yamada, Maria Clara, Marina Passos, Mateus Lisboa, Matheus Cortezi, Matheus Silva, Matheus Vian, Mírian Batista, Murilo Andrade, Murilo Cunha, Murilo Viana, Natan Cervinski, Nicolas Teodosio, Osvaldo Neto, Otávio Barradas, Patricia Minamizawa, Paulo Braga, Paulo Tadei, Pedro Henrique, Pedro Pereira, Peterson Santos, P Muniz, Priscila Santos, Rafael Lopes, Rafael Rodrigues, Rafael Romão, Ramayana Menezes, Reinaldo Silva, Renato Veirich, Ricardo Silva, Riverfount Riverfount, Robson Maciel, Rodrigo Alves, Rodrigo Freire, Rodrigo Vaccari, Rodrigo Vieira, Rogério Sousa, Ronaldo Silva, Ronaldo Silveira, Rui Jr, Samanta Cicilia, Sara Selis, Thiago Araujo, Thiago Borges, Thiago Bueno, Thiago Curvelo, Thiago Moraes, Thiago Oliveira, Thiago S, Thiago Souza, Tiago Minuzzi, Tony Dias, Valcilon Silva, Valdir Tegon, Victor Wildner, Vinícius Bastos, Vítor Gomes, Vitor Luz, Vladimir Lemos, Walter Reis, Wellington Abreu, Wesley Mendes, William Alves, Willian Lopes, Wilson Neto, Wilson Rocha, Xico Silvério, Yury Barros



Obrigado você



Afinal, o que são

Decor
adores

Decoradores



Decoradores são formas de estender ou restringir as funcionalidades de uma função. Decoradores são funções que embrulham outras funções

```
@decorador  
def função():  
    ...
```

Decoradores



```
@decorador  
def função( ):  
    ...
```



Decoradores



```
@decorador  
def função( ):  
    ...
```



Exemplo de um medidor de tempo [exemplo_00.py]



```
@medidor_de_tempo
def delay(secs):
    """Bota o código para dormir por secs."""
    sleep(secs)
    return secs
```

```
>>> delay(2)
delay demorou 2.001282 segundos.
2
```

Exemplo de um medidor de tempo [exemplo_00.py]



```
@medidor_de_tempo
```

```
def delay(secs):
```

```
    """Bota o código para dormir por secs."""
```

```
    sleep(secs)
```

```
    return secs
```

```
>>> delay(2)
```

```
delay demorou 2.001282 segundos.
```

```
2
```

Exemplo de um decorador de cache [exemplo_01.py]



```
from functools import cache

@cache
def delay(secs):
    """Bota o código para dormir por `secs`."""
    sleep(secs)
    return secs

print(
    delay(1), delay(2),
    delay(1), delay(2)
)
```

Exemplo de um decorador de cache [exemplo_01.py]



```
from functools import cache

@cache
def delay(secs):
    """Bota o código para dormir por `secs`."""
    sleep(secs)
    return secs

print(
    delay(1), delay(2),
    delay(1), delay(2)
)
```

```
$ time python exemplo_00.py
1 2 1 2

real    0m3,136s
```

Como é o código de um decorador? [exemplo_00.py]



```
def medidor_de_tempo(func):  
    def aninhada(*args, **kwargs):  
        tempo_inicial = datetime.now()  
  
        resultado = func(*args, **kwargs)  
  
        tempo_final = datetime.now()  
        tempo = tempo_final - tempo_inicial  
        print(f'{func.__name__} demorou {tempo.total_seconds()} segundos.')  
  
        return resultado  
    return aninhada
```

Vocabulário de nicho



- Decoradores são um **açúcar sintático** para o funcionamento de **closures**.
- **Closures**: são um caso especial de **aninhamento de funções** de **ordem superior**, que armazenam **variáveis livres**
- **Funções aninhadas**: São funções definidas dentro de funções
- **Funções de ordem superior**: Quer dizer que função podem receber ou retornar funções de **primeira classe**
- **Função de primeira classe**: São funções como objetos. Elas podem ser colocadas em variáveis, colocadas em contêineres e passadas/retornadas por funções
- **Variáveis livres**: Variáveis que não pertencem ao escopo global ou local

Como é o código de um decorador? [medidor_de_tempo.py]



Função de ordem superior


```
def medidor_de_tempo(func):  
    def aninhada(*args, **kwargs):  
        tempo_inicial = datetime.now()  
  
        resultado = func(*args, **kwargs)  
  
        tempo_final = datetime.now()  
        tempo = tempo_final - tempo_inicial  
        print(f'{func.__name__} demorou {tempo.total_seconds()} segundos.')  
  
        return resultado  
    return aninhada
```

Função aninhada

Função de primeira classe

Uma abordagem de baixo pra cima



- 
- A large, solid black arrow pointing downwards, indicating a sequence or progression from top to bottom.
- Funções de primeira classe
 - Funções de ordem superior
 - Aninhamento de funções
 - Closures
 - Variáveis livres
 - Decoradores

Funções como
objetos e funções
de ordem superior

HOFs

Funções como objetos [exemplo_02.py]



Funções são objetos em python. Assim como int e list também são objetos. Isso significa que podemos atribuir funções a variáveis e colocá-las em contêineres.

```
def calculadora(op, x, y):  
    operações = {  
        '+': soma, '-': sub,  
        '*': mul, '/': div  
    }  
    return operações[op](x, y)
```

```
def soma(x, y):  
    return x + y  
  
def sub(x, y):  
    return x - y  
  
def mul(x, y):  
    return x * y  
  
def div(x, y):  
    return x / y
```

"Funções como cidadãos de primeira classe"



O termo funções como objetos também pode ser encontrado na literatura como "primeira classe".

O que é ser um cidadão de segunda classe?



Um **cidadão de segunda classe** é uma pessoa que é sistematicamente discriminada dentro de um Estado ou outra jurisdição política, apesar de sua condição nominal de **cidadão** ou residente legal nesse Estado.

[https://artsandculture.google.com > entity](https://artsandculture.google.com/entity)

<https://artsandculture.google.com/entity/m08q65w?hl=pt>

Funções de ordem superior [exemplo_03.py]



Funções de ordem superior, são funções que podem receber ou retornar uma função.

```
from functools import reduce

def soma(x, y):
    return x + y

resultado = reduce(soma, [1, 2, 3, 4, 5])

print(resultado)  # 15
```

Funções de ordem superior [exemplo_04.py]



Funções de ordem superior, são funções que podem receber ou retornar uma função.

```
def maior_que_5(valor):  
    return valor > 5  
  
resultado = filter(maior_que_5, [3, 4, 5, 6, 7])  
  
print(list(resultado))  # [6, 7]
```

Exemplo de retorno de função [exemplo_05.py]



Exemplo de uma função que retorna outra função

```
from functools import partial
```

```
def soma(x, y):  
    return x + y
```

```
soma_1 = partial(soma, 1)  
soma_10 = partial(soma, 10)
```

```
print(soma_1(10)) # 11
```

```
print(soma_10(11)) # 11
```

HOF personalizada [exemplo_06.py]



Um exemplo simples

```
def soma_1(x):  
    return x + 1  
  
def duas_vezes(func, val):  
    return func(func(val))  
  
print(duas_vezes(soma_1, 2)) # 4
```

HOF personalizada [exemplo_07.py]



Um exemplo útil

```
def soma(x, y):  
    return x + y  
  
def zip_with(func, iter_a, iter_b):  
    return list(map(func, iter_a, iter_b))  
  
print(zip_with(soma, [1, 2, 3], [1, 2, 3]))  
# [2, 4, 6]
```


HOF personalizada [exemplo_08.py]



Um exemplo insano

```
from functools import partial

def zip_with_2(func):
    return partial(map, func)

zip_soma = zip_with_2(soma)
zip_mul = zip_with_2(mul)

print(list(
    zip_soma([1, 2, 3], [1, 2, 3])
)) # [2, 4, 6]
print(list(
    zip_mul([1, 2, 3], [1, 2, 3])
)) # [1, 4, 9]
```

Closure

E funções
aninhadas

Funções aninhadas [exemplo_9.py]



Funções aninhadas são funções definidas dentro de funções



```
def ola(nome):  
    def func_interna(nome):  
        if nome.lower() == 'marilene':  
            print(f'Olá {nome}. A noite, tainha, vinho e muito ...')  
        else:  
            print(f'Olá {nome}, boas vindas!')  
    func_interna(nome)  
  
ola('Marilene')
```

Para que servem funções aninhadas?



- Função ajudantes
 - Evitar repetir código dentro da função
 - Colocar toda a complexidade em um lugar só
- Encapsulamento
 - A função interna não pode ser acessada globalmente
- Escopo de variáveis
 - A função interna pode usar variáveis da função externa

Um exemplo útil [exemplo_10.py]



```
from unicodedata import normalize
```

```
def normaliza(*palavras):
```

```
    saida = []
```

```
    for palavra in palavras:
```

```
        normalizado = normalize('NFKD', palavra)
```

```
        normalizada = normalizado.encode('ASCII', 'ignore').decode('ASCII')
```

```
        saida.append(normalizada)
```

```
    return saida
```

```
print(normaliza('Érico', 'Sabiá', 'João'))
```

```
# ['Erico', 'Sabia', 'Joao']
```

Um exemplo útil [exemplo_10.py]



```
def normaliza2(*palavras):  
  
    def ajudante(palavra):  
        normalizado = normalize('NFKD', palavra)  
        return normalizado.encode('ASCII', 'ignore').decode('ASCII')  
  
    return [ajudante(palavra) for palavra in palavras]  
  
print(normaliza2('Érico', 'Sabiá', 'João'))  
# ['Erico', 'Sabia', 'Joao']
```

Um exemplo útil [exemplo_10.py]



```
def normaliza2(*palavras):
```

Encapsulamento

```
    def ajudante(palavra):
```

```
        normalizado = normalize('NFKD', palavra)
```

```
        return normalizado.encode('ASCII', 'ignore').decode('ASCII')
```

```
    return [ajudante(palavra) for palavra in palavras]
```

```
print(normaliza2('Érico', 'Sabiá', 'João'))
```

```
# ['Erico', 'Sabia', 'Joao']
```

Escopo de variáveis [exemplo_11.py]



```
def soma_x(val_externo):  
    def interna(val_interno):  
        return val_externo + val_interno  
    return interna  
  
soma_1 = soma_x(1)  
soma_10 = soma_x(10)  
  
print(soma_1(10)) # 11  
print(soma_10(1)) # 11
```


Escopo de variáveis [exemplo_11.py]



```
def soma_x(val_externo):  
    def interna(val_interno):  
        return val_externo + val_interno  
    return interna
```

```
soma_1 = soma_x(1)  
soma_10 = soma_x(10)
```

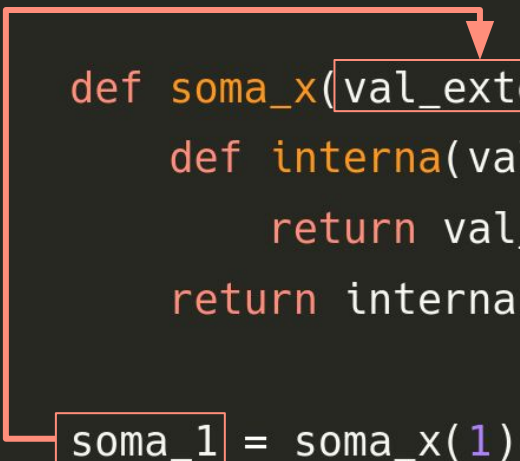
```
print(soma_1(10)) # 11  
print(soma_10(1)) # 11
```

Closures



Closures são funções que encapsulam uma função (função interna) e a retornam.

O grande passo das closures é que seu escopo é preservado. Ou seja, as variáveis de "fora" da função interna também são preservados



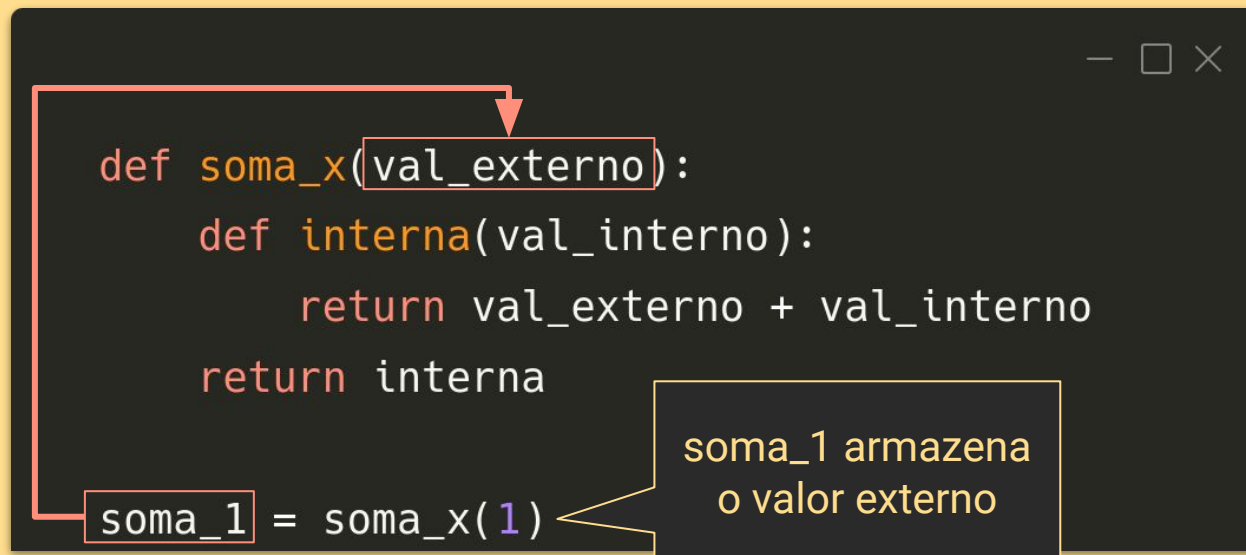
```
def soma_x(val_externo):  
    def interna(val_interno):  
        return val_externo + val_interno  
    return interna  
  
soma_1 = soma_x(1)
```

Closures



Closures são funções que encapsulam uma função (função interna) e a retornam.

O grande passo das closures é que seu escopo é preservado. Ou seja, as variáveis de "fora" da função interna também são preservados



```
def soma_x(val_externo):  
    def interna(val_interno):  
        return val_externo + val_interno  
    return interna  
  
soma_1 = soma_x(1)
```

soma_1 armazena o valor externo

Um exemplo mais divertido [exemplo_12.py]



```
1  def contador(start=0):
2      count = start
3
4      def interna():
5          count += 1
6          return count
7
8      return interna
9
10 c = contador()
11 print(c()) # 1
12 print(c()) # 2
```

count é uma variável definida no escopo de contador, mas modificada na função interna

interna é uma **closure**

Rodar esse código, ele tem um bug!

Qual o escopo de 'count'?



```
1  # Escopo global
2  a = 1  # Global
3
4  def contador(start=0):  # Global
5      # Escopo local de contador
6      count = start  # Local
7
8      def interna(): # local
9          # Escopo local de interna
10         xpto = 'batatinha'  # Local
11
12         count += 1  # Global ou Local?
13
14         return count
15
16     return interna
```

Escopo não local (nonlocal)



```
1  def contador(start=0):  
2      count = start  
3  
4      def interna():  
5          nonlocal count  
6          count += 1  
7          return count  
8  
9      return interna
```

soma_1 armazena
o valor externo

Escopo não local (nonlocal)



```
1 def contador(start=0):  
2     count = start  
3  
4     def interna():  
5         nonlocal count  
6         count += 1  
7         return count  
8  
9     return interna
```

count é uma
variável livre

soma_1 armazena
o valor externo

Por baixo do capô



Como saber se algo é uma closure, quais os valores das variáveis livres e quais os nomes das variáveis livres?

```
>>> c = contador()  
>>> c.__closure__  
(<cell at 0x7f72aaf21ae0: int object at 0x7f72adc1c150>,) — □ ×  
  
>>> c.__closure__[0].cell_contents  
0  
  
>>> c.__code__.co_freevars  
( 'count', )
```


Decor
adores

Viemos pra isso!

Closures especiais [exemplo_13.py]



Uma closure pode ser um decorador se:

- A função externa receber uma função
- A função recebida é a variável livre
- Retorna a função interna

```
def decorador(func):  
    def interna(*args):  
        resultado = func(*args)  
        return f'Sou uma closure e sua função retornou {resultado}'  
  
    return interna  
  
def soma(x, y):  
    return x + y  
  
decorada = decorador(soma)  
print(decorada(1, 2)) # Sou uma closure e sua função retornou 3
```

Closures especiais



```
def decorador(func):  
    def interna(*args):  
        resultado = func(*args)  
        return f'Sou uma closure e sua função retornou {resultado}'  
  
    return interna  
  
def soma(x, y):  
    return x + y  
  
decorada = decorador(soma)  
print(decorada(1, 2)) # Sou uma closure e sua função retornou 3
```

Sem a sintaxe @

O açúcar sintático @



```
def decorador(func):  
    def interna(*args):  
        resultado = func(*args)  
        return f'Sou uma closure e sua função retornou {resultado}'  
  
    return interna  
  
@decorador  
def soma(x, y):  
    return x + y  
  
print(soma(1, 2)) # Sou uma closure e sua função retornou 3
```

Pronto!

sóquenão

Agora vamos voltar onde começamos [medidor_de_tempo.py]



Função de ordem superior

```
def medidor_de_tempo(func):  
    def aninhada(*args, **kwargs):  
        tempo_inicial = datetime.now()  
  
        resultado = func(*args, **kwargs)  
  
        tempo_final = datetime.now()  
        tempo = tempo_final - tempo_inicial  
        print(f'{func.__name__} demorou {tempo.total_seconds()} segundos.')  
  
        return resultado  
    return aninhada
```

Função aninhada

Função de primeira classe

Decoradores com parâmetros



Mas como passamos parâmetros para os decoradores?

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

<https://flask.palletsprojects.com/en/2.1.x/quickstart/#a-minimal-application>

Um exemplo simples [exemplo_14.py]



```
def retry(erro, vezes):
    count = 0
    def intermediaria(func):
        def closure(*args, **kwargs):
            nonlocal count
            try:
                return func(*args, **kwargs)
            except erro as e:
                count += 1
                print(f'{func.__name__} error: {e} retry: {count}')
                if count < vezes:
                    closure(*args, **kwargs)
        return closure
    return intermediaria
```



```
from datetime import datetime

def debug(*, verbose=False, level=0):
    def intermediaria(func):
        def interna(*args, **kwargs):
            tstart = datetime.now()
            result = func(*args, **kwargs)
            t_final = datetime.now() - tstart
            if verbose:
                print(
                    f'Chamada {func.__name__}\n'
                    f'parâmetros posicionais: {args}\n'
                    f'parâmetros nomeados: {kwargs}\n'
                )
            if level > 0:
                print(f'Resultado: {result}')
            if level > 1:
                print(f'Tempo total: {t_final.total_seconds()}')
            return result
        return interna
    return intermediaria
```

exemplo_15.py

namespace de decoradores



Mostrar no shell, é mais fácil

Built-in python decorator

- `@abc.abstractmethod` A decorator indicating abstract methods.
- `@abc.abstractproperty` A subclass of the built-in `property()`, indicating an abstract property.
- `@asyncio.coroutine` Decorator to mark generator-based coroutines. New in version 3.4.
- `@atexit.register` Register func as a function to be executed at termination.
- `@classmethod` Return a class method for function.
- `@contextlib.contextmanager` Define a factory function for with statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.
- `@functools.cached_property` Transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the instance.
- `@functools.lru_cache` Decorator to wrap a function with a memoizing callable that saves up to the maxsize most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments. New in python 3.2.
- `@functools.singledispatch` Transforms a function into a single-dispatch generic function. New in python 3.4.

<https://github.com/lord63/awesome-python-decorator>



picpay.me/dunossauro



apoia.se/livedepython



pix.dunossauro@gmail.com



Ajude o projeto <3

