python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

Project maintained by dunossauro
Hosted on GitHub Pages — Theme by mattgraham

Retornar «< 9. Usos variados de closures «< Página inicial »>

10. Decoradores

Agora que você e o Jaber já estão craques em closures os decoradores não apresentam medo, pois tudo em decoradores são closures. No tópico 9.5 você usou um decorador e, embora eles não tenham sido devidamente explicados, vamos entender tudo sobre eles agora.

10.1 Qual a cara de um decorador?

Um decorador nada mais é que um açúcar sintático para as closures. Viu? você já sabe tudo sobre eles, sem mesmo saber deles. Vamos entender essa composição diferente.

Uma closure é aplicada em Python assim:

```
closure(funcao(argumentos))
```

Invocamos a função externa como uma função e passamos como argumento a nossa função com seus argumentos. E os decoradores?

```
@closure
def funcao(argumentos):
    pass
```

Ou seja, é apenas açúcar sintático. Contudo a apresentação é muito explícita, pelo menos pra mim. Fica evidente que a função `closure` decora a `funcao`. Outro ponto importante e que difere, apenas em nível sintático, é que a função é decorada apenas quando é definida.

duas vezes na função.

```
@closure
def funcao(argumentos):
    pass

closure(funcao(argumentos)) # linha do problema
```

Agora todas as vezes que você encontrar um `@` em cima de uma definição de uma função, você já sabe do que se trata.

10.2 Montando nosso primeiro decorador

No tópico 8 iniciamos nossa discussão sobre o uso de closures, vamos implementar closures de uma maneira mais eficiente e mais agradável visualmente.

Para iniciar vamos pensar em uma simples função que soma dois números como fizemos ao introduzir o conceito de funções nos primeiros tópicos, para ficar evidentemente simples a utilização de decoradores, mas falar é fácil. Vamos ao código:

```
def soma(x, y):
    """
    Função que efetivamente soma dois números
    """
    return x + y
```

Não é preciso ser um gênio como o Jaber para saber como usar essa função:

```
>>> soma(1, 1) # 2
>>> soma(2.0, 2.0) # 4.0
>>> soma(3j + 3j) # 6j
```

Ela funciona efetivamente com todos os tipos de números em Python. Embora seja possível imaginar que nossa função usa o operador `+`. Ele faz com que nossos objetos numéricos invoquem seu método mágico interno `__add__` ou `__radd__`. O único problema é que outros objetos em Python também implementam esse método. Strings e listas podem usar o `__add__`, mas somente entre sí. Vale lembrar aqui que Python é uma linguagem fortemente tipada. Eu não vou conseguir somar uma string com um inteiro ou com uma

Vamos pensar que nossa função `soma()` só trabalha com números (complexos, inteiros e de ponte flutuante). Então a validação da entrada vai ter que ser feita, pois nós não queremos somar strings e listas.

Jaber diz: Por que não deixamos a função receber de qualquer coisa, assim nossa função pode ser usada para somar listas com listas, strings com strings? Nossa função vai ser muito mais poderosa.

Calma Jaber, existe um problema em não validar os valores. Se as entradas forem de tipos diferentes a função vai retornar um `TypeError` e não vai ser muito legal para quando o usuário da nossa função estiver usando. Vamos entender isso, para que fique claro:

```
>>> soma('Jaber', 2)
# TypeError: must be str, not int
```

Esse comportamento não é legal, imagina quantos tipos diferentes de erros podem ocorrer por isso? Vamos resolver de uma maneira simples:

```
from numbers import Number

def soma(x, y):
    """
    Função que soma dois números.

    isinstance faz uma comparação e valida se um valor
        é de uma determinada classe.

    Caso um deles não seja, um erro vai ser
        forçado e a mensagem vai ser exibida
    """
    if isinstance(x, Number) and isinstance(y, Number):
        return x + y
    raise TypeError('Insira somente números (int, complex, float)')
```

Tá, ficou bonito. Vamos usar:

```
>>> soma(1, 1)
```

```
    soma(1.0, 2.0)
# 4.0
>>> soma(3j, 3j)
# 6j
```

Até então tudo está exatamente igual, mas vamos tentar usar outros tipos de dados:

```
>>> soma(1, [1])
# TypeError: Insira somente números (int, complex, float)
>>> soma(1, 'Jaber')
# TypeError: Insira somente números (int, complex, float)
>>> soma((1, 2, 3), 1)
# TypeError: Insira somente números (int, complex, float)
```

```
Jaber diz: Hmmmmmmm. Muito bonito, falou muito e nada de decoradores
```

Ok, vamos lá. Você fez tudo isso, mas agora eu vou te pedir uma função que faz multiplicação e ela também só pode receber números:

```
from numbers import Number

def mul(x, y):
    if isinstance(x, Number) and isinstance(y, Number):
        return x * y
    raise TypeError('Insira somente números (int, complex, float)')


>>> mul(1, 2)
# 2
>>> mul(1, [1])
# TypeError: Insira somente números (int, complex, float)
```

Você entendeu tudo Jaber, mas esqueceu de tudo que falamos sobre closures? Agora vamos ser inteligentes e usar as closures que aprendemos:

```
def validate_numbers(func):
    """
    Closure que decora a função.
```

```
    def _validate(x, y):
        """
        Executa a validação e retorna a execução da função.
        """
        if isinstance(x, Number) and isinstance(y, Number):
            return func(x, y)
        raise TypeError('Insira somente números (int, complex, float)')
    return _validate
```

Você concorda que só temos o código em um único lugar e podemos decorar as duas funções e executar um código simples dentro da função? Vamos validar os valores com a closure `validate_numbers` e aplicar valores na função para testar:

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

@validate_numbers
def mul(x, y):
    return x * y

>>> soma(1, 1)
# 2
>>> soma(2.0, 2.0)
# 4.0
>>> soma(3j + 3j)
# 6j
>>> mul(1, 1)
# 2
>>> mul(2.0, 2.0)
# 4.0
>>> mul(3j, 3j)
# (-9+0j)
```

Agora `validate_numbers` além de decorar nossas funções com a closure pode ser usado para qualquer tipo de função que receba dois argumentos (claro a validação pode não ser a mesma, mas funciona). Mas e os erros?

```
>>> soma(1, 'Jaber')
```

```
    mdc(2, 2);
# TypeError: Insira somente números (int, complex, float)
```

Tudo funcionou muito bem. Vamos tentar entender um pouco mais sobre a natureza dos decoradores.

10.2 Usando um pouco melhor a função externa

Embora a função externa que leva o nome do decorador ganhe como argumento a função a ser executada e execute a função interna, ela pode ter outras funcionalidades. Vamos pensar um pouco e gerar algumas coisas perigosas, mas que podem ser usadas em contexto onde sejam de suma importância.

10.2.1 Um cache simples

Vamos pensar em uma função que executa algum tipo de cálculo mirabolante. Por exemplo, você pode fixar um cache para verificar se o número é par. Mas para isso você precisa do módulo da divisão por 2.

Então, vamos supor que a nossa função de soma só execute a soma quando o segundo valor passado for par, fora desse contexto não iremos executar a função.

Vamos tentar, e explicar enquanto fazemos:

```
def segundo_eh_par(func, cache={}):
    """
    cache é um dicionário que é iniciado vazio.

    A cada iteração ele executa (y%2 == 0)
        e armazena no dicionario o valor de y
    """
    def interna(x, y):
        if y not in cache:
            cache[y] = (y%2 == 0)

        if cache[y]:
            return func(x, y)
        raise Exception('Insira somente valores pares para y')
    return interna
```

```
>>> segundo_eh_par.__defaults__
# ({},)
```

usando `__defaults__` podemos ver o valor inserido no nosso dicionário. Vamos usar esse decorador em uma função:

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


>>> soma(2,2)
# 4
>>> segundo_eh_par.__defaults__
# ({2: True},)

>>> soma(2,5)
# Exception: Insira somente valores pares
>>> segundo_eh_par.__defaults__
# ({2: True, 5: False},)
```

Agora, todas as vezes que os valores forem usado novamente a computação não será necessária, pois ela já está no dicionário. Vamos fazer isso com fibonacci, é surpreendente a diferença de desempenho, mas vamos aprender a medir o tempo de execução de uma função antes.

10.2.2 Tempo de execução de uma função

Esse decorador também segue uma ideia simples, podemos gravar em arquivos, gerar logs, chamar bancos de dados. Vai além da imaginação, mas vamos tentar medir o tempo que nossa função leva para ser executada.

```
from time import time

def timeit(func):
    """
    Decorador para medir o tempo.
```

```
    """
    def inner(*args):
        ts = time() # pega a 'hora' atual
        result = func(*args) # Executa a função
        te = time() # pega a hora atual
        # Aqui vai rolar um print nesse formato:
        # <nome_da_função> <argumentos_da_função> <subtração_de_te_por_ts>
        print('{} {} {:.2}'.format(func.__name__, args, te - ts))
        return result
    return inner
```

Esse é um decorador bem simples de se entender. Ele vai decorar uma função e nós saberemos o tempo que ela levou para ser executada em segundos. Você pode pensar que esse decorador é mais do mesmo, porém, ele nos mostra quão genéricos devem ser os decoradores. A ídeia é de que possam ser usados em qualquer lugar. Nesse caso `eh_par` acaba sendo um contra-exemplo de um bom decorador. Mas, vale lembrar que o objetivo dele é totalmente didático. Sei que você já deve ter entendido tudo sobre decoradores. Porém, eles também podem receber argumentos, o que os tornariam mais genéricose potentes. Então, vamos lá...

10.3 Decoradores com parâmetros

Uma das coisas mais legais de quando se está aprendendo Python, é que em um certo momento você acaba entendendo a ideia de que não podemos fazer código com alto acoplamento. Por exemplo, nos ultimos tópicos você simplesmente definiu uma `def` dentro de outra `def` . Porém, as coisas podem ser mais simpáticas quando você simplesmente se dá o prazer de experimentar.

Por exemplo, e se fizessemos uma closure de uma closure?

```
def param(args):
    def funcao_externa(func):
        def funcao_interna(*args):
            return func(*args)
        return funcao_externa
    return funcao_interna
```

Nesse caso, parece um `Inception` , mas calma, não precisamos do Christopher Nolan para entender o que se passa nesse decorador. Vamos ler linha a linha (sim, foi por isso que

Na primeira linha foi definida uma função chamada `param`, é um nome bem descritivo na verdade. Lembre-se que nos exemplos passados usamos a função externa para ser nosso decorador. Agora nesse caso, essa camada, que chamamos `param`, vai ser nosso decorador. Mas uma coisa muito interessante sobre ela é que ela não recebe a função como parâmetro. Sim, ela recebe um parâmetro, mas não é a função.

Sim, eu sei, está confuso. Vamos fazer com exemplos, um bom código diz mais que mil palavras.

```
def verbose(level=0):
    def funcao_externa(func):
        def funcao_interna(*args):
            if level == 1:
                # Nesse caso, ele vai printar o nome da função decorada
                print(func.__name__)
            if level == 2:
                # Nesse caso, ele vai printar o nome da função decorada
                # junto com os argumentos que foram invocados
                print(func.__name__, args)
            return func(*args)
        return funcao_interna
    return funcao_externa
```

Definimos um novo decorador chamado `verbose`, ele recebe um argumento que é nível de verbosidade no qual o decorador vai exercer sobre as demais funções. Caso `level` seja `0`, seu valor default, ele não vai fazer nada. A única ação nesse caso seria retornar a função. Porém, caso os valores variem entre 1 e 2, diferentes coisas serão mostradas na tela. Caso a função decorada receba `level=1`, toda vez que a função for invocada o nome dela será mostrado na tela. (Sim, isso pode ser bem útil para um momento de desespero na hora de depurar seu código). Caso o valor enviado seja `level=2`, ou seja, mais verboso, ele vai nos retornar o nome da função junto dos argumentos que foram invocados. Vamos decorar uma função antes de retornar à explicação.

```
@verbose(2)
def soma(*args):
    return sum(args)
```

Embora quem faça a frente da nossa função seja `verbose`, o decorador real, a função que

`verbose` nesse caso, vai simplesmente adicionar uma camada a mais no escopo local da função `funcao_externa` e por consequência também no escopo da `funcao_interna`. Ou seja, você pode parametrizar a execução do decorador sem que a parametrização seja feita com os argumentos passados a função decorada.

10.4 Identidade das funções decoradas

Continuando esse tópico, uma coisa muito interessante acontece com funções decoradas. Ela perde sua identidade e isso pode ser um grande problema para a fase de depuração do seu código. Imagine que quando uma função decorada apresentar um erro, o erro sempre será mostrado no decorador. Vamos tentar olhar como isso acontece:

```
def sem_decorador(x, y):
    """Função sem decorador."""
    return x, y

>>> sem_decorador
# <function __main__.sem_decorador>
```

Ok, temos uma função no escopo `__main__`, sem problemas, esse é o esperado, porém se decorarmos essa função, vou usar um decorador genérico para fazer isso:

```
def decorator(f):
    def inner(args):
        """Função interna do decorador."""
        return f(args)
    return inner

@decorator
def com_decorador(x, y):
    """Função com decorador."""
    return x, y

>>> com_decorador
# <function __main__.decorator.<locals>.inner>
```

Ou seja, toda vez em que a função `com_decorador` é invocada ela é o decorador `decorator`, mas especificamente ela é a função `inner`, a função interna do decorador. Vamos olhar mais profundamente com alguns métodos do objeto.

```
# 'sem_decorador'
>>> com_decorador.__name__
# 'inner'
>>> sem_decorador.__doc__ # __doc__ nos mostra a docstring da função
# 'Função sem decorador.'
>>> com_decorador.__doc__
# 'Função interna do decorador.'
```

Fica evidente que na hora de depurar vamos ter vários problemas com isso, embora não seja o foco principal desse tópico, vamos usar um decorador do `functools`, o decorador de `wraps`.

```
from functools import wraps


def decorator(f):
    @wraps(f)
    def inner(args):
        """Função interna do decorador."""
        return f(args)
    return inner

@decorator
def com_decorador(x, y):
    """Função com decorador."""
    return x, y

>>> com_decorador
# <function __main__.com_decorador>
```

Com isso, uma cópia dos métodos `__module__`, `__name__`, `__qualname__`, `__annotations__` e `__doc__` será feita na função "embrulhada" (wraped) e as propriedades da função decorada continuarão a ser mantidas após o embrulho. Ou seja, poderemos tanto facilitar a vida quando for necessário depurar nosso código e também o autocomplete do seu editor, a função `help()` e todas as coisas que precisam determinar o comportamento da sua função continuariam a funcionar como se a função não estivesse decorada. Porém, ela agora será uma função embrulhada. Ou seja, quando a função for chamada ela vai ser invocada pelo embrulho e você perderá a visualização da representação sem o decorador de `wraps` (`<function __main__.decorator.<locals>.inner>`). Para

exatamente a função `<function __main__.decorador.<locals>.timer>` . Com isso, você agora pode usar a função sem se preocupar com o comportamento do decorador e caso precise desse tipo de interação, você pode invocar diretamente `com_decorador.__wrapped__` . Então você não precisa mais se preocupar com diferentes tipos de interação e manter a sanidade mental.

Mas, uma coisa um pouco diferente aconteceu nesse exemplo com `@wraps` , existe um novo decorador inserido dentro da função interna do decorador e é isso que vamos ver no proximo tópico.

10.5 Decorando decoradores

Como você deve ter percebido, não existem mais limites entre decorar funções e fazer encapsulamento das mesmas. Podemos decorar as funções que são decoradores e as funções decoradas também podem ter mais de um decorador. Por exemplo:

```
@decorador1
@decorador2
def funcao_que_redebe_dois_decoradores():
    pass
```

nesse caso, o `decorador1` decora o `decorador2` que decora a função `funcao_que_redece_dois_decoradores` . Sim, eu sei que você entendeu e isso pode ter um ciclo infinito de possibilidades. Uma coisa que deve ser levada em consideração é que conforme você aninha decoradores as funções tendem a ficar mais dificeis de depurar e de prever o comportamento.

E com isso terminamos aqui a primeira parte do nosso conteúdo sobre programação funcional sem imports. Tudo que construímos até aqui foi usando as funções builtin do python. Mas antes de prosseguir, gostaria de fazer algumas breves considerações matemáticas, para que seja possível criarmos código de mais qualidade.

Retornar «< 9. Usos variados de closures «< Página inicial »>