nappacademy

# Napp Academy

Orlando Saraiva Júnior

# Decoradores de função e closures

# Decoradores de função e closures

nappacademy

Decoradores (*decorators*) de função nos permitem "marcar" funções no código-fonte para modificar o seu comportamento de alguma maneira.

```
@decorate
def target():
    print('Running target()')
```

É um recurso poderoso, mas dominá-lo exige compreender as *closures*.

# Closure

# Closures

nappacademy

*Closure* é uma função aninhada que tem acesso a uma *variável livre* de uma função envolvente que concluiu sua execução.

Três características de um *closure* :

- É uma função aninhada
- Tem acesso a uma *variável livre* no escopo externo
- Ele é retornado da função envolvente

  Uma variável livre é uma variável que não está vinculada ao escopo local. Para que os *closures* funcionem com variáveis imutáveis, como números e strings, temos que usar a palavra-chave *non-local*.

# Funções aninhadas

nappacademy

Funções internas, também conhecidas como funções aninhadas, são funções que você define dentro de outras funções.

Em Python, esse tipo de função tem acesso direto a variáveis e nomes definidos na função envolvente.

As funções internas têm muitos usos, principalmente como fábricas de *closure* e funções de decorador.

**inner.py**

# Regra para escopo de variáveis

nappacademy

Python não exige que você declare variáveis, mas supõe que uma variável cujo valor já tenha sido atribuído no corpo de uma função é local.

Observe como o interpretador trata *b*

**escopo.py**

# Closures

nappacademy

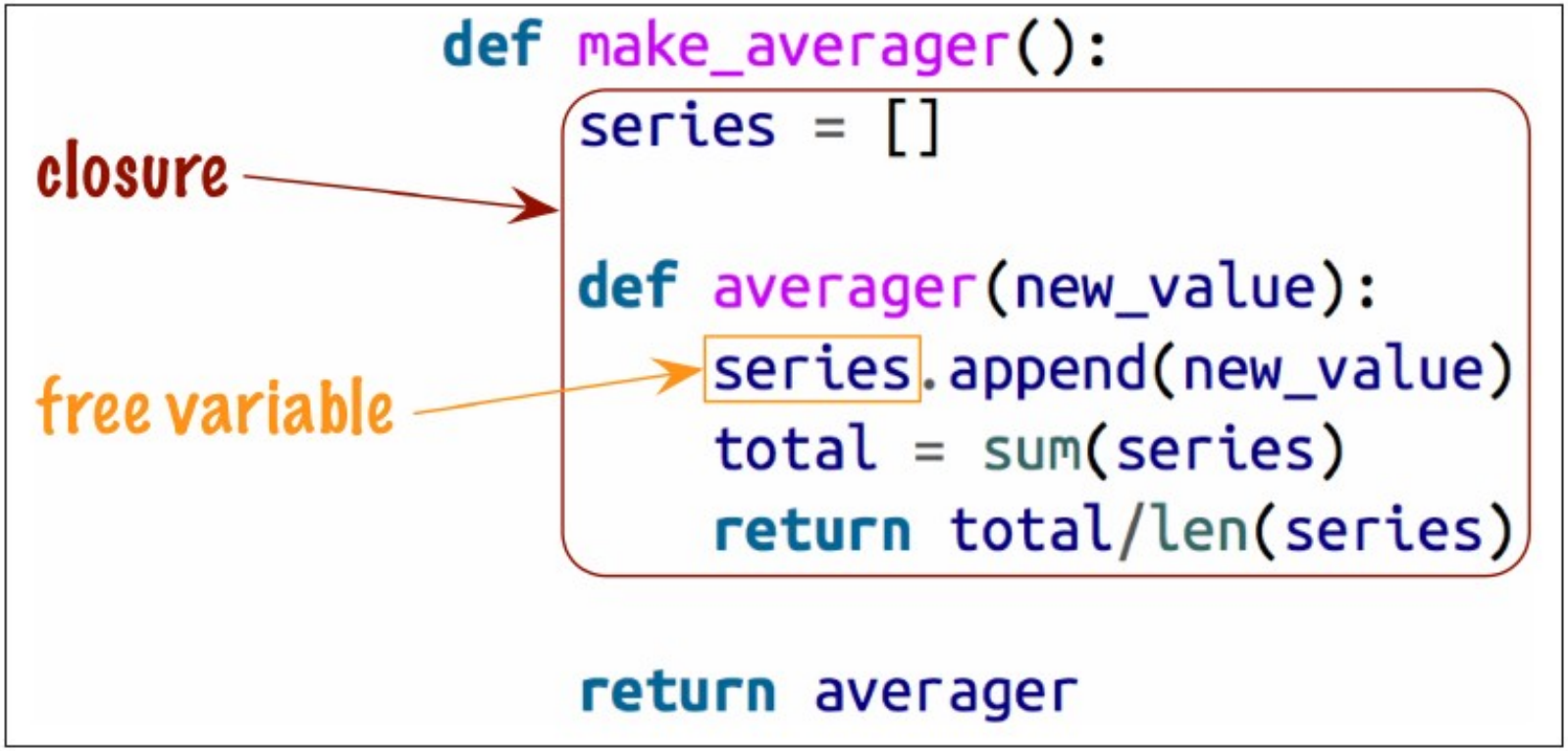
*Closure* é uma função com um escopo estendido, que engloba variáveis não globais referenciadas no corpo da função que não estão definidas ali.

Ela precisa acessar variáveis não globais definidas for a de seu corpo.

**avg.py**

# Closures

nappacademy



# Closures

nappacademy

Vamos inspecionar a função criada por make_averanger

```
>>> avg
<function make_averager.<locals>.averager at 0x7f1eb321a8b0>
>>> avg.__closure__
(<cell at 0x7f1eb3262f70: list object at 0x7f1eb322da40>,)
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
>>>
```

**avg.py**

# Closures

nappacademy

Nossa implementação de make_averager não é eficiente.

A próxima versão calcula a soma com sum sempre que averager é chamada.

Mas em que ponto falha ?

**avg2.py**

# Nonlocal

Estamos fazendo uma atribuição a ***count*** no corpo de ***averager***, e isso faz dela uma variável *local*. O mesmo ocorre com a variável ***total***.

Para contornar este problema, a declaração *nonlocal* foi introduzida no python 3.

**avg3.py**

**Pesquisar**
- Documentação
- PEP3104

# Decorador

# Decorador

nappacademy

Um decorador é um invocável (*callable*) que aceita outra função como argumento (função decorada). O decorador pode realizar algum processamento com a função decorada e devolvê-la ou substituí-la por outra função ou objeto invocável.

**decorador1.py**

# Quando os decoradores são executados

nappacademy

Uma característica fundamental dos decoradores é que eles são executados imediatamente após a função decorada ser definida. Normalmente isso ocorre em *tempo de importação* (isto é, quando um módulo é carregado pelo interpretador python)

**decorador2.py**

Os decoradores de uma função são executados assim que o módulo é importado, porém as funções decoradas executam somente quando são explicitamente chamadas.

Tempo de importação (*import time*) é diferente de tempo de execução (*runtime*)

**decorador3.py**

# Decoradores da biblioteca-padrão

nappacademy

Python tem três funções embutidas criada para decorar métodos:

- property
- classmethod
- staticmethod.

# Decoradores parametrizados

nappacademy

Ao fazer um parse de um decorador, python pega a função decorada e passa-a como primeiro argumento para a função decoradora. Como fazer um decorador aceitar outros argumentos ?

Crie uma fábrica (*factory*) de decoradores que aceite esses argumentos e devolva um decorador que, por sua vez, será aplicado à função a ser decorada.

**decorador4.py**

# Sobre padrões de projetos

nappacademy

O **Decorator** é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.

Os decoradores de função em python se enquadram na descrição geral de Decorator do livro GoF: “Confere responsabilidades adicionais a um objeto dinamicamente. Os decoradores oferecem uma alternativa à criação de subclasses para estender funcionalidades”

Em nível de implementação, os decoradores em python não lembramo padrão de projeto clássico Decorator.

# Gerenciadores de contexto

# Gerenciadores de contexto

nappacademy

Digamos, por exemplo, que você tenha certa funcionalidade com pré-condições e pós-condições que precisam ser atendidas ao executar essa funcionalidade.

Neste cenário, gerenciadores de contexto são interessantes.

**spoiler.py**

# Gerenciadores de contexto

nappacademy

- Normalmente, um Context Manager consiste em dois métodos mágicos: __*enter*__ e __*exit*__.
- A instrução *with* chama o *Context Manager*.
- A instrução with chama o método __*enter*__ e o valor retornado será atribuído à variável rotulada após a palavra-chave *as*.
- O método __enter__ não precisa retornar um valor.
- Depois que o método __*enter*__ foi chamado, o que quer que esteja dentro do bloco *with* será executado, então o python irá chamar o método __*exit*__

# Gerenciadores de contexto

nappacademy

@contextmanager é um decorador que permite criar um gerenciador de contexto  partir de uma função geradora simples, em vez de criar uma classe e implementar o protocolo. **(exemplo 2)**

Já ContextDecorator é uma classe-base para definir gerenciadores de contexto baseados em classes que também podem ser usados como decoradores de função, executando toda a função em um contexto gerenciado. **(exemplo 3)**

**context_manager1.py**
**context_manager2.py**
**context_manager3.py**

**Pesquisar**
- Documentação
  contextlib