

ASIMOV

Decoradores

Os decoradores podem ser pensados como funções que modificam a *funcionalidade* de outra função. Eles ajudam a tornar seu código mais curto e mais "Pythonico".

Para explicar corretamente os decoradores, vamos contruí-los lentamente a partir de funções. Certifique-se de reiniciar o Python e os Notebooks para esta palestra para parecer o mesmo em seu próprio computador. Então, vamos quebrar as etapas:

Revisão de funções

```
In [1]: def func():  
        return 1
```

```
In [2]: func()
```

```
Out[2]: 1
```

Revisão de escopo

Lembre-se das palestras anteriores Python usa o Scope para saber o que um rótulo está se referindo. Por exemplo:

```
In [6]: s = 'Global Variable'  
  
def func():  
    print locals()
```

Lembre-se de que as funções do Python criam um novo escopo, o que significa que a função possui seu próprio namespace para encontrar nomes de variáveis quando são mencionados dentro da função. Podemos verificar variáveis locais e variáveis globais com as funções `local()` e `globals()`. Por exemplo:

```
In [4]: print(globals())  
  
{'__name__': '__main__', '__doc__': 'Automatically created module for IPython  
interactive environment', '__package__': None, '__loader__': None, '__spec__':  
None, '__builtin__': <module 'builtins' (built-in)>, '__builtins__': <module  
'builtins' (built-in)>, '_ih': ['', 'def func():\n    return 1', 'func()', 'pr  
int globals()', 'print(globals())'], '_oh': {2: 1}, '_dh': ['C:\\Users\\rodri  
\\Dropbox\\Udemy\\Python Completo\\Notebooks Traduzidos'], 'In': ['', 'def fun  
c():\n    return 1', 'func()', 'print globals()', 'print(globals())'], 'Out':  
{2: 1}, 'get_ipython': <bound method InteractiveShell.get_ipython of <ipykerne  
l.zmqshell.ZMQInteractiveShell object at 0x0000025D9EA72128>>, 'exit': <IPytho
```

```
n.core.autocall.ZMQExitAutocall object at 0x0000025D9EA86828>, 'quit': <IPython.core.autocall.ZMQExitAutocall object at 0x0000025D9EA86828>, '_': 1, '___': '', '____': '', '_i': 'print globals()', '_ii': 'func()', '_iii': 'def func():\n    return 1', '_i1': 'def func():\n    return 1', 'func': <function func at 0x0000025D9FB4AE18>, '_i2': 'func()', '_2': 1, '_i3': 'print globals()', '_i4': 'print(globals())'}
```

Aqui recebemos um dicionário de todas as variáveis globais, muitas delas predefinidas em Python. Então, vamos em frente e olhe para as chaves:

```
In [8]: print globals().keys()

['_dh', '___', '_i', 'quit', '___builtins__', 's', '_ih', '___builtin__', '_2', 'func', '___name__', '___', '___sh', '_i8', '_i7', '_i6', '_i5', '_i4', '_i3', '_i2', '_i1', '___doc__', '_iii', 'exit', 'get_ipython', '_ii', 'In', '_oh', 'Out']
```

Observe como **s** está lá, a variável global que definimos como uma string:

```
In [10]: globals()['s']
```

Out[10]: 'Global Variable'

Agora, execute nossa função para verificar quaisquer variáveis locais no func () (não deve haver nenhuma)

```
In [11]: func()

{}
```

Ótimo! Agora, vamos continuar construindo a lógica do que é um decorador. Lembre-se que em Python **tudo é um objeto** . Isso significa que as funções são objetos a qual podem ser atribuídos etiquetas e passados para outras funções. Comece com alguns exemplos simples:

```
In [5]: def hello(name='Jose'):
        return 'Hello ' + name
```

```
In [6]: hello()
```

Out[6]: 'Hello Jose'

Atribua um rótulo à função. Note-se que não estamos usando parênteses aqui porque não estamos chamando a função hello, em vez disso, estamos apenas colocando-a em uma variável.

```
In [7]: greet = hello
```

```
In [8]: greet
```

Out[8]: <function __main__.hello>

```
In [9]: greet()
```

```
Out[9]: 'Hello Jose'
```

Essa definição não está ligada a função original:

```
In [10]: del hello
```

```
In [11]: hello()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-11-a803225a2f97> in <module>()
----> 1 hello()

NameError: name 'hello' is not defined
```

```
In [12]: greet()
```

```
Out[12]: 'Hello Jose'
```

Funções dentro de funções

Ótimo! Agora que vimos como tratar funções como variáveis, podemos aprender como definir funções dentro de outras funções.

```
In [13]: def hello(name='Jose'):
          print('The hello() function has been executed')

          def greet():
              return '\t This is inside the greet() function'

          def welcome():
              return "\t This is inside the welcome() function"

          print(greet())
          print(welcome())
          print("Now we are back inside the hello() function")
```

```
In [14]: hello()
```

```
The hello() function has been executed
    This is inside the greet() function
    This is inside the welcome() function
Now we are back inside the hello() function
```

```
In [15]: welcome()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-15-efaf77b113fd> in <module>()
----> 1 welcome()

NameError: name 'welcome' is not defined
```

Perceba que devido ao escopo, a função welcome() não está definida fora de hello(). Agora aprenderemos sobre como retornar funções dentro de funções.

Retornando funções

```
In [18]: def hello(name='Rodrigo'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Rodrigo':
        return greet
    else:
        return welcome
```

```
In [19]: x = hello()
```

Agora vamos ver que função será retornada se definirmos `x = hello()`. Note como os parêntesis fechados significam que o nome foi definido como Rodrigo.

```
In [40]: x
```

```
Out[40]: <function __main__.greet>
```

Ótimo. Note como `x` está apontando para a função `greet`, dentro da função `hello`.

```
In [20]: print(x())

        This is inside the greet() function
```

Dê uma olhada rápida para o código novamente.

O `if/else` dentro da função estão retornando `greet` e `welcome`, não `greet()` e `welcome()`. Se você colocar o parêntesis acabará por executar as funções. Sem eles, é possível passar os mesmos para outras variáveis.

Quando escrevemos `x = hello()`, `hello()` é executado e o nome passado é Rodrigo, então a função `greet` é retornada. Se mudarmos a definição para `x = hello(name="Paulo")` a função retornada será `welcome`. Podemos também tentar printar `hello()()` que fará com que a função `greet` seja executada.

Funções como argumentos

Agora vamos ver como passar funções como argumentos para outras funções.

```
In [21]: def hello():
    return 'Hi Jose!'

    def other(func):
        print('Other code would go here')
        print(func())
```

```
In [22]: other(hello)
```

Other code would go here
Hi Jose!

Ótimo! Olhe como podemos passar funções como objetos e usá-los dentro de outras funções. Agora conseguimos construir nosso primeiro decorador:

Criando decoradores

No exemplo anterior nós criamos manualmete um decorador. Aqui nós iremos modificá-lo para clarificar as coisas:

```
In [23]: def new_decorator(func):  
  
    def wrap_func():  
        print("Code would be here, before executing the func")  
  
        func()  
  
        print("Code here will execute after the func()")  
  
    return wrap_func  
  
def func_needs_decorator():  
    print("This function is in need of a Decorator")
```

```
In [24]: func_needs_decorator()
```

This function is in need of a Decorator

```
In [25]: # Redefine a função  
func_needs_decorator = new_decorator(func_needs_decorator)
```

```
In [26]: func_needs_decorator()
```

Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()

O que aconteceu aqui? Um simples decorador entrou na função e modificou seu comportamento. Agora vamos entender como nós podemos reescrever este código usando o símbolo @, que Python usa como decorador.

```
In [27]: @new_decorator  
def func_needs_decorator():  
    print("This function is in need of a Decorator")
```

```
In [28]: func_needs_decorator()
```

Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()