

Luciano Ramalho
ramalho@python.pro.br

Funções como objetos



pythonpro

Objetos de primeira classe

pythonpro

Terminologia

en	pt-br
first class function	função de primeira classe
first class object	objeto de primeira classe
≈ first class citizen	≈ cidadão de primeira classe

Funções de primeira classe

- Podem manipuladas da mesma forma que outros objetos de primeira classe (ex: int):
 - criar em tempo de execução
 - atribuir a uma variável
 - armazenar em uma estrutura de dados
 - passar como argumento

Demonstração

```
>>> def fatorial(n):  
...     '''devolve n!'''  
...     return 1 if n < 2 else n * fatorial(n-1)  
...  
>>> fatorial(42)  
14050061177528798985431426062445115699363840000000000  
>>> fat = fatorial  
>>> fat(5)  
120  
>>> map(fat, range(10))  
<map object at 0x1006bead0>  
>>> list(map(fat, range(11)))  
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]  
>>> fat  
<function fatorial at 0x1006a2290>  
>>> type(fatorial)  
<class 'function'>  
>>>
```

Demonstração (cont.)

```
>>> dir(fatorial)
['__annotations__', '__call__', '__class__', '__closure__',
 '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__get__', '__getattr__', '__globals__', '__gt__',
 '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
 '__module__', '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>> fatorial.__name__
'fatorial'
>>> fatorial.__doc__
'devolve n!'
>>> fatorial.__code__
<code object fatorial at 0x100520810, file "<stdin>", line 1>
```

Demonstração (cont.)

```
>>> dir(fatorial.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', 'co_argcount',
 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
 'co_firstlineno', 'co_flags', 'co_freevars',
 'co_kwonlyargcount', 'co_lnotab', 'co_name', 'co_names',
 'co_nlocals', 'co_stacksize', 'co_varnames']
>>> fatorial.__code__.co_varnames
('n',)
>>> fatorial.__code__.co_code
b'|\x00\x00d\x01\x00k\x00\x00r\x10\x00d\x02\x00S|\x00\x00t
\x00\x00|\x00\x00d\x02\x00\x18\x83\x01\x00\x14S'
```

Demonstração (cont.)

```
>>> import dis
>>> dis.dis(fatorial.__code__.co_code)
      0 LOAD_FAST           0 (0)
      3 LOAD_CONST         1 (1)
      6 COMPARE_OP         0 (<)
      9 POP_JUMP_IF_FALSE   16
     12 LOAD_CONST         2 (2)
     15 RETURN_VALUE
>>    16 LOAD_FAST           0 (0)
     19 LOAD_GLOBAL         0 (0)
     22 LOAD_FAST           0 (0)
     25 LOAD_CONST         2 (2)
     28 BINARY_SUBTRACT
     29 CALL_FUNCTION         1 (1 positional, 0 keyword pair)
     32 BINARY_MULTIPLY
     33 RETURN_VALUE
```


Implicações de F1ªC

- Funções como objetos de primeira classe abrem novas possibilidades de organização de programas
- Paradigma funcional
 - > 50 anos de história
 - fundamentos matemáticos: cálculo lambda
- Repensar padrões de projeto!

Design Patterns in Dylan or Lisp

16 of 23 patterns are either invisible or simpler, due to:

- ◆ First-class types (6): Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility
- ◆ First-class functions (4): Command, Strategy, Template-Method, Visitor
- ◆ Macros (2): Interpreter, Iterator
- ◆ Method Combination (2): Mediator, Observer
- ◆ Multimethods (1): Builder
- ◆ Modules (1): Facade

Peter Norvig:
“Design Patterns in
Dynamic Programming”

Função de ordem superior

- Aceita funções como argumentos e/ou devolve função como resultado

```
>>> frutas = ['pequi', 'uva', 'caju', 'banana',  
'caqui', 'umbu']  
>>> sorted(frutas)  
['banana', 'caju', 'caqui', 'pequi', 'umbu', 'uva']  
>>> sorted(frutas, key=len)  
['uva', 'caju', 'umbu', 'pequi', 'caqui', 'banana']  
>>> def invertida(palavra):  
...     return palavra[::-1]  
...  
>>> sorted(frutas, key=invertida)  
['banana', 'uva', 'caqui', 'pequi', 'umbu', 'caju']
```

Objetos invocáveis

pythonpro

Documentação oficial

- **Language Reference > Data model**
 - <http://docs.python.org/dev/reference/datamodel.html>
- **Seção 3.2 - The standard type hierarchy > Callable types**

Callable types

These are the types to which the function call operation (see section [Calls](#)) can be applied:

User-defined functions

A user-defined function object is created by a function definition (see section [Function definitions](#)). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Special attributes:

Attribute	Meaning	
<code>__doc__</code>	The function's documentation string, or <code>None</code> if unavailable	Writable
<code>__name__</code>	The function's name	Writable
<code>__qualname__</code>	The function's <i>qualified name</i> <i>New in version 3.3.</i>	Writable
<code>__module__</code>	The name of the module the function was defined in, or <code>None</code> if unavailable.	Writable
<code>__defaults__</code>	A tuple containing default argument values for those arguments that have defaults, or <code>None</code> if no arguments have a	Writable

Tipos invocáveis

- User-defined functions: **def** ou **lambda**
- Instance methods: invocados via instâncias
- Generator functions: funções com **yield**
- Built-in functions: escritas em C (no CPython)
- Built-in methods: idem
- Classes: métodos **__new__** e **__init__**
- Class Instances: método **__call__**

def e lambda

```
>>> def fatorial(n):  
...     '''devolve n!'''  
...     return 1 if n < 2 else n * fatorial(n-1)  
...  
>>> fat2 = lambda n: 1 if n < 2 else n * fat2(n-1)  
>>>  
>>> fat2(42)  
14050061177528798985431426062445115699363840000000000  
>>> type(fat2)  
<class 'function'>  
>>> type(fatorial)  
<class 'function'>  
>>>
```


Funções anônimas

pythonpro

lambda

- Açúcar sintático para definir funções dentro de expressões
 - Normalmente: em chamadas de função

```
>>> sorted(frutas, key=lambda s: s[::-1])  
['banana', 'uva', 'caqui', 'pequi', 'umbu', 'caju']
```

- Limitadas a uma expressão em Python
 - Não podem usar estruturas de controle, atribuição etc.

lambda cria uma função anônima

```
>>> fat2 = lambda n: 1 if n < 2 else n * fat2(n-1)
>>> fat2.__name__
'<lambda>'
>>>
```

lambda cria uma função anônima

```
>>> fat2 = lambda n: 1 if n < 2 else n * fat2(n-1)
>>> fat2.__name__
'<lambda>'
>>>
```

“Funções anônimas têm um grave defeito: elas não têm nome.”

Anônimo

Método Lundh para refatorar lambdas

1. Escreva um comentário explicando o que a função anônima faz.
2. Estude o comentário atentamente, e pense em um nome que capture a essência do comentário.
3. Crie uma função usando o comando **def**, usando este nome.
4. Remova o **lambda** e o comentário.

Lambda

```
lista = sorted(frutas, key=lambda s: s[::-1])
```

Lambda refatorado

```
lista = sorted(frutas, key=lambda s: s[::-1])
```



```
def invertida(palavra):  
    return palavra[::-1]
```

```
lista = sorted(frutas, key=invertida)
```

Atributos de funções

pythonpro

Atributos de funções

RW	<code>__doc__</code>	str	documentação (docstring)
RW	<code>__name__</code>	str	nome da função
RW	<code>__module__</code>	str	nome do módulo onde a função foi definida
RW	<code>__defaults__</code>	tuple	valores default dos parâmetros formais
RW	<code>__code__</code>	code	bytecode do corpo da função + metadados
R	<code>__globals__</code>	dict	variáveis globais do módulo
RW	<code>__dict__</code>	dict	atributos criados pelo programador
R	<code>__closure__</code>	tuple	associações para as variáveis livres
RW	<code>__annotations__</code>	dict	anotações de parâmetros e retorno
RW	<code>__kwdefaults__</code>	dict	valores default dos parâmetros nomeados

Annotations

pythonpro

Porque anotações?

Dica: não é para transformar Python em Java!

- Usos interessantes citados por Raymond Hettinger e outros no StackOverflow [1]:
 - documentação
 - verificação de pré-condições
 - multi-métodos / sobrecarga [2]

[1] <http://bit.ly/py-why-annotations>

[2] <https://pypi.python.org/pypi/overload>

Exemplo anotações

```
def truncar(texto:str, largura:'int > 0'=80) -> str:
    '''devolve o texto truncado no primeiro espaço até a largura,
       ou no primeiro espaço após a largura, se existir'''
    termino = None
    if len(texto) > largura:
        pos_espaco_antes = texto.rfind(' ', 0, largura)
        if pos_espaco_antes >= 0:
            termino = pos_espaco_antes
        else:
            pos_espaco_depois = texto.rfind(' ', largura)
            if pos_espaco_depois >= 0:
                termino = pos_espaco_depois
    if termino is None:
        return texto.rstrip()
    else:
        return texto[:termino].rstrip()
```

Demo anotações

```
>>> truncar.__annotations__
{'largura': 'int > 0', 'texto': <class 'str'>,
 'return': <class 'str'>}
>>> truncar.__defaults__
(80,)
>>> from inspect import signature
>>> assi = signature(truncar)
>>> assi.parameters
mappingproxy(OrderedDict([('texto', <Parameter at
0x1003c7680 'texto'>), ('largura', <Parameter at 0x1006a43c0
'largura'>)]))
>>> for nome, par in assi.parameters.items():
...     print(nome, ':', par.name, par.default, par.kind)
...
texto : texto <class 'inspect._empty'> POSITIONAL_OR_KEYWORD
largura : largura 80 POSITIONAL_OR_KEYWORD
```

Decorators

pythonpro

Decorador simples

- Função de ordem superior: recebe uma função como argumento, retorna outra

```
from time import strftime

def logar(func):
    def logadora(*args, **kwargs):
        res = func(*args, **kwargs)
        print(strftime('%H:%M:%S.%m'), args, kwargs, '->', res)
        return res
    return logadora
```

```
@logar
def dobro(n):
    return n*2
```

```
>>> x = dobro(21)
13:32:42.10 (21,) {} -> 42
>>> dobro.__name__
'logadora'
```

Como é interpretado

```
@dec  
def func():  
    pass
```

é o mesmo que

```
def func():  
    pass  
func = dec(func)
```

```
@dec2(arg)  
def func():  
    pass
```

é o mesmo que

```
def func():  
    pass  
func = dec2(arg)(func)
```

```
@d1(arg)  
@d2  
def func():  
    pass
```

é o mesmo que

```
def func():  
    pass  
func = d1(arg)(d2(func))
```


Demo decorator

```
>>> l = [dobro(x) for x in range(4)]
13:15:24.10 (0,) {} -> 0
13:15:24.10 (1,) {} -> 2
13:15:24.10 (2,) {} -> 4
13:15:24.10 (3,) {} -> 6
>>> l
[0, 2, 4, 6]
>>>
>>> g = (dobro(x) for x in range(4))
>>> g
<generator object <genexpr> at 0x1006a8af0>
>>> list(g)
13:15:44.10 (0,) {} -> 0
13:15:44.10 (1,) {} -> 2
13:15:44.10 (2,) {} -> 4
13:15:44.10 (3,) {} -> 6
[0, 2, 4, 6]
>>>
```

Decorator: memoizar

```
import functools
```

```
def memoizar(func):  
    cache = {}
```

```
    @functools.wraps(func)
```

```
    def memoizada(*args, **kwargs):
```

```
        chave = (args, str(kwargs))
```

```
        if chave not in cache:
```

```
            cache[chave] = func(*args, **kwargs)
```

```
        return cache[chave]
```

```
    return memoizada
```

Exemplo apenas didático.
A biblioteca padrão já
tem `functools.lru_cache`

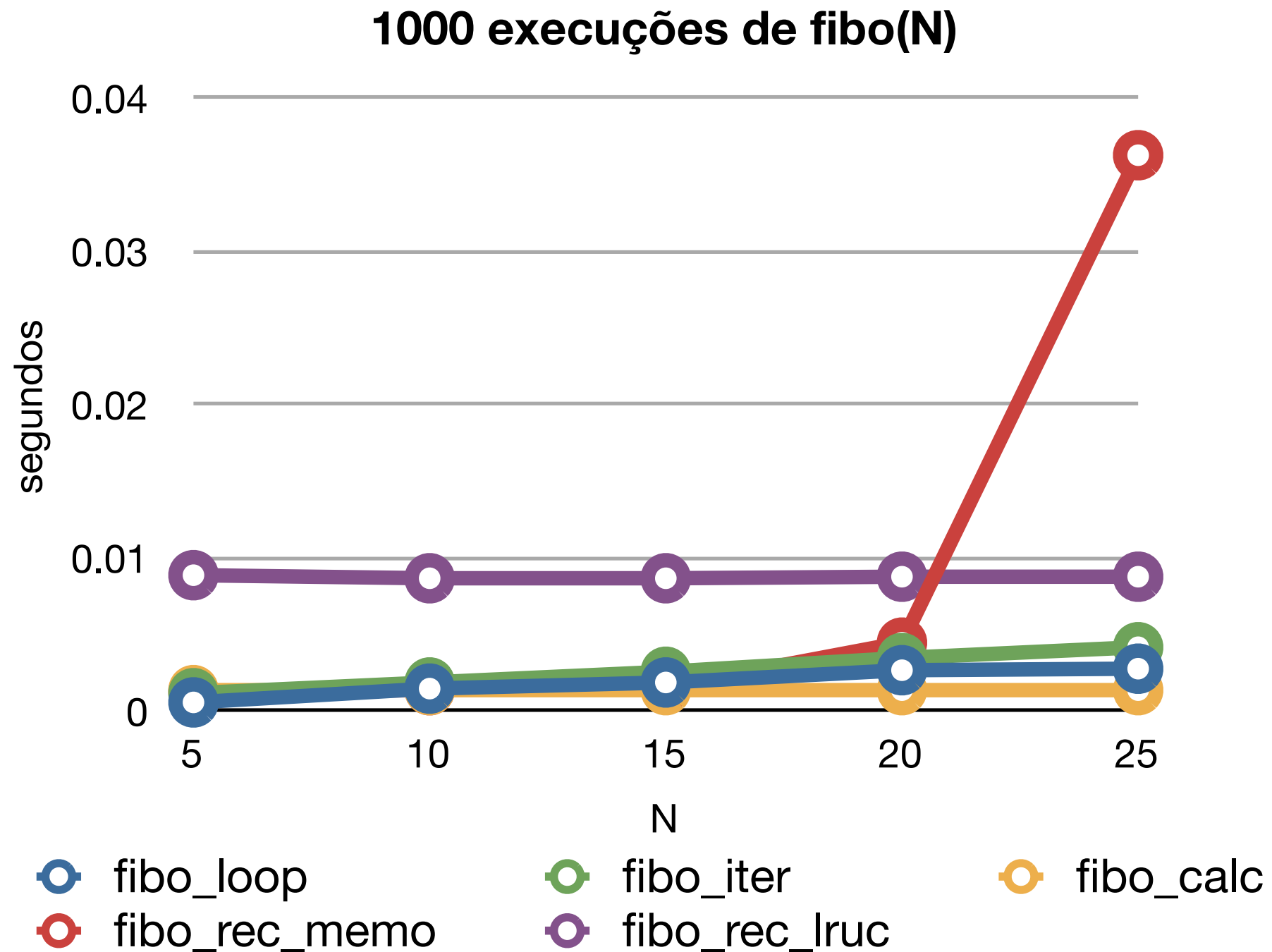
Demo @memoizar

```
def fibo_rec(n):  
    if n < 2:  
        return n  
    return fibo_rec(n-2) + fibo_rec(n-1)
```

```
@memoizar  
def fibo_rec_memo(n):  
    if n < 2:  
        return n  
    return fibo_rec(n-2) + fibo_rec(n-1)
```

```
$ python3 fibonacci.py  
fibo_loop          0.0006      0.0011      0.0016      0.0029      0.0030  
fibo_iter          0.0013      0.0020      0.0026      0.0034      0.0043  
fibo_calc          0.0014      0.0014      0.0015      0.0015      0.0014  
fibo_rec           0.0021      0.0288      0.2822      3.2485      35.2203  
fibo_rec_memo      0.0015      0.0015      0.0017      0.0045      0.0367  
fibo_rec_lruc      0.0092      0.0085      0.0086      0.0089      0.0087  
$
```

Comparando



Decoradores prontos

- Built-ins
 - **property, classmethod, staticmethod**
- Pacote functools
 - **lru_cache, partial, wrap**
- Outros...
- Python Decorator Library (*in* [wiki.python.org](https://wiki.python.org/moin/PythonDecoratorLibrary))

<https://wiki.python.org/moin/PythonDecoratorLibrary>

Closures

pythonpro

Qual é o problema?

- O corpo de uma função pode conter variáveis livres (não locais)
- Funções de primeira classe podem ser definidas em um contexto e usadas em outro contexto totalmente diferente
- Ao executar uma função, como resolver as associações das **variáveis livres**?
- Escopo dinâmico ou escopo léxico?

Demo closure

```
>>> conta1 = criar_conta(1000)
>>> conta2 = criar_conta(500)
>>> conta1()
1000
>>> conta1(100)
1100
>>> conta1()
1100
>>> conta2(-300)
200
>>> conta2(-300)
Traceback (most recent call last):
...
ValueError: Saldo insuficiente
>>> conta2()
200
```


Uma closure

- A closure da função **conta** preserva a associação da **variável livre contexto** com seu valor no escopo léxico (o local da definição da função, não de sua invocação)

```
def criar_conta(saldo_inicial):  
    # precisamos de um objeto mutável  
    contexto = {'saldo': saldo_inicial}  
    def conta(movimento=0):  
        if (contexto['saldo'] + movimento) < 0:  
            raise ValueError('Saldo insuficiente')  
        contexto['saldo'] += movimento  
        return contexto['saldo']  
    return conta
```

Demo closure

```
>>> conta1 = criar_conta(1000)
>>> conta1()
1000
>>>
conta1.__closure__[0].cell_contents
{'saldo': 1000}
>>> conta1(100)
1100
>>> conta1()
1100
```

Exemplo errado!!!

```
def criar_conta(saldo_inicial):  
    saldo = saldo_inicial  
    def conta(movimento=0):  
        if (saldo + movimento) < 0:  
            raise ValueError('Saldo insuficiente')  
        saldo += movimento  
        return saldo  
    return conta
```

```
>>> c1 = criar_conta(99)  
>>> c1()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "/Users/luciano/prj/python.pro.br/github/func_objects/  
saldo0.py", line 35, in conta  
    if (saldo + movimento) < 0:  
UnboundLocalError: local variable 'saldo' referenced before assignment
```

nonlocal

- Declaração de variável não-local, disponível somente a partir do Python 3.0

```
def criar_conta(saldo):  
    def conta(movimento=0):  
        nonlocal saldo  
        if (saldo + movimento) < 0:  
            raise ValueError('Saldo insuficiente')  
        saldo += movimento  
        return saldo  
    return conta
```

pythonpro

Python para profissionais

- Próximos lançamentos:
 - 1ª turma de **Python para quem estudou Java**
 - 5ª turma de **Objetos Pythonicos**
 - 6ª turma de **Python para quem sabe Python**

Para saber mais sobre estes cursos

ONLINE ao VIVO visite:

<http://python.pro.br>

siga: **@pythonprobr**