Introdução à linguagem Python

Thiago Martins

Programação Funcional – O que é?

- Idéia central: Representar a computação como a avaliação de expressões compostas por funções.
- Funções são o construto central nestas linguagens. É possível gerar funções em tempo de execução, compor funções, usar funções como parâmetros para outras funções, etc.
- Funções são *puras.* Isso significa que a avaliação de uma função *não* afeta o estado do programa. Uma função avaliada com os mesmos parâmetros produz sempre o mesmo resultado.
- Em abordagens mais puras, o programa não tem *nenhum* estado! Variáveis são todas imutáveis. Não há laços. Algoritmos iterativos descritos com *recorrência*. Etapas intermediárias de computação são descritas com *sequências.*

Programação Funcional - Histórico

- Cálculo Lambda, proposto por Alonzo Church na década de 30. O mesmo período no qual Alan Turing propôs a Máquina de Turing.

  A dicotomia Funcional/Imperativa é tão antiga quanto a própria ciência da computação!

- Lisp desenvolvida em 1958, 5 anos após FORTRAN.
- Nunca “pegou” realmente. Lisp é a clássica linguagem amada por todos, usada por ninguém...
- No século XXI, explode a nova geração: Haskell, Erlang, R, JavaScript, Rust…
- Linguagens imperativas ganham aspectos funcionais (C++, C#,

Python é uma linguagem Funcional?

Não

- Estado não é puro, variáveis podem ser modificadas
- Funções podem ter efeitos colaterais.
- Laços são estruturas proeminentes na linguagem

Mas…

- Funções são entidades de “primeira classe” em Python. Variáveis podem conter funções, funções podem ser usadas como parâmetros…
- Suporta declarações de funções simples por expressões lambda.
- É possível criar funções dinamicamente com clausuras.
- A Biblioteca padrão possui várias funções da alta ordem, map, reduce, filter…
- Suporte a sequências com geradores.

Aspectos Funcionais de Python

- Funções são entidades com outras em Python.
- É possível passar funções como parâmetros.

```
def aplica_funcao(f, x):
    return f(x)

aplica_funcao(print, “Hello World!”)
```

Declarando Funções: Cálculo Lambda

Usa-se para declarar funções simples.

Sintaxe:

```
lambda var1, var2, … varn: (expressão)
```

Exemplo:

```
a = lambda x, y: x*y
print(a(2,2))
```

Declarando Funções: Clausuras

Cláusuras são funções que “capturam” parte do escopo no qual elas foram definidas. Este é um comportamento natural em Python.

Exemplo:

```
def cria_somador(x):
    def funcao(y):
        return x+y
    return funcao


a = cria_somador(2)
b = cria_somador(3)
print(a(5))
print(b(5))
```

Note como a e b são funções distintas!

Declarando Funções: Clausuras

Cláusuras podem ser usadas com expressões lambda:

Exemplo:

```
def cria_somador(x):
    return lambda y: x+y


a = cria_somador(2)
b = cria_somador(3)
print(a(5))
print(b(5))
```

Note como a e b são funções distintas!

Declarando Funções: Clausuras

Atenção: Clausuras estão vinculadas a símbolos, não valores!

Exemplo:

```
def cria_somadores(n):
    ret = []
    for i in range(n):
        ret.append(lambda x: x+i)
    return ret
a = cria_somadores(3)
for f in a:
  print(f(1))
```

Isso não funciona!
A variável i é sempre a mesma para todas as expressões lambda

Declarando Funções: Clausuras

Atenção: Clausuras estão vinculadas a símbolos, não valores!

Solução 1: compor funções (solução mais “funcional”)

```
def cria_somadores(n):
    ret = []
    for i in range(n):
      ret.append((lambda y:(lambda x: x+y))(i))
    return ret
a = cria_somadores(3)
for f in a:
  print(a(1))
```

Declarando Funções: Clausuras

Atenção: Clausuras estão vinculadas a símbolos, não valores!

Solução 2: Parâmetros padrão

```
def cria_somadores(n):
    ret = []
    for i in range(n):
        ret.append(lambda x, y=i: x+y)
    return ret
a = cria_somadores(3)
for f in a:
    print(a(1))
```

Note que neste caso as funções recebem *dois* argumentos (um deles com valor padrão)

Decoradores: modificam funções (e classes)

```
def roda_duas_vezes(funcao):
  def _f():
    funcao()
    funcao()

  return _f

@roda_duas_vezes
def hello():
  print("Hello World!")
```

Python: Iteradores e tipos iteráveis

Um iterador é a forma principal de se acessar sequencialmente tipos enumeráveis.

Na maior parte do tempo, iteradores são “escondidos” pela linguagem e o programador não toma conhecimento da sua existência.

Python: Iteradores e tipos iteráveis

Um iterador pode ser obtido para um objeto iterável por meio da função `iter()`.

```
a = [1, 2, 3, 4]
i = iter(a)
```

Entradas no objeto iterável são obtidos por chamadas sucessivas à função `next()`.

```
next(i)
next(i)
next(i)
next(i)
next(i)
```

Quando todas as entradas do objeto iterável chegam ao fim, é lançada uma exceção do tipo `StopIteration`

Tipos iteráveis: Geradores

Nem todo tipo iterável *armazena* dados. Alguns *calculam* os dados a medida que estes são necessários. Estes tipos de dados são chamados *geradores.*

Uma maneira de se declarar um gerador em Python é através de uma função com a instrução `yield`.

```
def fibonacci():
    x, y = 0, 1
    while True:
        x, y = y, x+y
        yield x
```

Este é um gerador “infinito”, ou seja, ao ser iterado com iter(), *nunca* produz uma exceção do tipo StopIteration.

Tipos iteráveis: Geradores

A função que contém `yield` *não* retorna um objeto gerador. A execução é interrompida no momento em que encontra `yield` e o seu estado de execução é preservado. A execução retoma quando é chamada `next()` sobre seu iterador. Uma instrução `return` *ou o* término da função *encerra* a iteração.

```
def fibonacci(n):
  x, y = 0, 1
  for i in range(n):
     x, y = y, x+y
     yield x
list(fibonacci(10))
```

Tipos iteráveis: Geradores do Python

A biblioteca padrão contém algumas funções que produzem objetos geradores. A mais conhecida é a já vista `range()`.

Outros geradores importantes:

`enumerate(x)` Gerador que produz duplas com o valor e o índice em cada posição de `x`.

`zip(x, y, z...)` Gerador que produz tuplas com valores em posições correspondentes em `x, y, z`...

Tipos iteráveis: Expressões Geradoras

Expressões geradoras são maneiras de se construir geradores simples a partir de outros objetos iteráveis.

Sintaxe:

`(expressao(var) for var in objeto)` Gerador que produz o valor da expressão aplicada a cada entrada em objeto.

Exemplo:

```
i = iter((2*x for var in gerador_fibonacci()))
next(i)
next(i)
next(i)
...
```

Tipos iteráveis: Expressões Geradoras

Sintaxe *avançada*:

```
(exp(v1, v2, v3,…) for v1 in o1 for v2 in o2 for v3 in o3… if cond)
```

Gerador que produz exp com todas as combinações dos elementos de o1, o2, o3,… (com índices variando do primeiro ao último) somente se a condição cond for verdadeira.

Exemplo:

```
list(([i,j] for i in range(5) for j in range(5) if i>j))
```

Tipos iteráveis: Compreensão de listas

Uma sintaxe similar a de expressões geradoras pode ser usada para se criar novas listas a partir de um objeto iterável finito.

Sintaxe:

`[expressao(var) for var in objeto]` Gera uma lista contendo o valor de expressão aplicada a cada entrada de objeto.

Exemplo: Soma de dois vetores

```
def soma(x, y):
    return [i + j for i, j in zip(x,y)]
```

Funções de alta ordem em Python

A biblioteca padrão de Python possui várias funções que operam sobre outras funções.

Algumas de destaque:

`map(fun, objiter)`: produz um objeto gerador com a aplicação sucessiva de `fun` a cada entrade de `objiter`.

Exemplo:

```
a = range(10)
for i in map(lambda x: x*x, a):
    print(i)
```

Funções de alta ordem em Python

`functools.reduce(fun2, objiter)`: aplica a função de dois argumentos `fun` sucessivamente aos objetos de `objiter`. Incialmente aplica aos dois primeiros objetos. Depois, aplica ao resultado da operação anterior e ao terceiro, assim por diante.

Exemplo (soma os números de 0 a 9):

```
a = range(10)
functools.reduce(lambda x, y: x+y, a)
```

Funções de alta ordem em Python

`filter(fun, objiter)`: Retorna um objeto iterador com todos os objetos de `objiter` para os quais a função `fun` retorna `True`.

Exemplo (mostra os inteiros pares menores do que 10)

```
a = range(10)
for i in filter(lambda x: x%2==0, a):
    print(a)
```

Combinando tudo

Este código calcula o produto interno entre dois vetores:

```
def produto_interno(x, y):
   return functools.reduce(lambda x, y: x+y, (i*j for i, j in zip(x,y))
```

Em notação imperativa:

```
def produto_interno(x, y):
   total = 0
   for i in range(len(x)):
      Total += x[i]*y[i]
```