Aspectos funcionais do Python

Carlos Tavares

28 de Outubro de 2024

Python é uma linguagem multiparadigma: imperativa e estruturada

Pode ser considerada uma linguagem funcional: possui muitas características de uma linguagem funcional.

Uma das características é a existência de (funções que recebem funções como argumentos). Exemplos: **mapas, filtros, reduções**.

Permite-nos que os programas estejam mais perto do "o quê"ao invés do "como"

Mertz, David. Functional Programming in Python. O'Reilly Media, 2015.

Listas em compreensão

Compreensões i

"Compreensão": redução de conjunto de elementos a uma definição geradora. Em matemática é possível definir um conjunto em extensão,

$$\{1, 2, 3, 4, 5, 6\},$$

e em compreensão:

$$\{a \in \mathbb{N} | a >= 5 \text{ and } a <= 6\}$$

Outro exemplo

$$\{(a, b) \in \mathbb{N} \times \mathbb{N} |\ a * b = 0\}$$

Compreensões ii

O Python permite a construção de vários objetos, como listas, conjuntos, ou dicionários, por compreensão, ou seja, através da sua definição.

Por exemplo, uma lista pode ser definida como

» [regra domínio]

Exemplos:

Uma lista definida em extensão para os numeros de 1 a 10

l = [1,2,3,4,5,6,7,8,9,10]

poderá ser definida em compreensão da seguinte maneira:

l = [i for in in range (1, 11)]

Compreensões iii

(i é a **regra**, for i in range (1,11) o **domínio**)

Compreensões iv

É possível utilizar **expressões** na definição do elemento constituinte da lista, como

```
[i**2 for i in range (1, 31)],
```

funções como,

```
[str (i) for i in range (1,31)]
```

ou até mesmo **condições**

```
[i if i % 2 == 0 else 1 for i in range (0, 101)]
```

Compreensões v

Do lado do "domínio", é possível aplicar condições sobre o mesmo

```
[i for i in range (1, 11) if i % 2 == 0]
```

Existem definições em compreensão para vários tipos de objectos em Python, como conjuntos ou dicionários. Sobre isso mais à frente...

Exercício 1. Construa uma lista dos cubos de todos os números no intervalo dado pelo utilizador.

Exercício 2. Construa uma lista de todos os números que não são uma potencia de 2.

Funções de ordem superior

Funções de ordem superior i

Considere-se uma funcao em Python

```
def funcao (a,b):
    return (a+b)*(a*b)
```

Uma função é um objecto em Python, e pode ser atribuída a uma variável:

```
» f = funcao
»f
<function funcao at 0x101208>
» f (5,10)
65
» f (5,0)
5
```

Funções de ordem superior ii

Dado que uma variável pode ser uma função *por aplicar*, uma função pode receber outra função como argumento. Isto é o que se denomina como uma **função de ordem superior**.

```
# f é uma função
def aplica_funcao (f, a, b):
    return f (a,b)
```

Isto torna o Python mais expressivo e por consequência mais poderoso...

Funções de ordem superior iii

Exercicio. Implemente uma mini-calculadora, que soma ou multiplica, lendo o operador e os operandos do utilizador e utilizando funções de ordem superior.

As funções map, filter e reduce

A função map i

Matematicamente, um mapa é um Functor: um método que mapeia uma coleção de elementos para outra coleção de elementos pelo uso de uma função.

$$F : C \times f \rightarrow C, \text{ onde C é uma coleção e f uma função} \quad (1)$$

Python possui uma construção que permite fazer isso. A função *map*:

map (função, coleção)

A função map ii

Exemplo:

```
def get_square_modulo_6 (s):
        return s % 6

l = [i for in range (1, 101)]
r = map (get_square_modulo_6, l)

O r é "preguiçoso", mas é "iterável"
for i in r:
        print (i)
```

A função map iii

listas em compreensão vs função map?

As listas não são preguiçosas, i.e. todos os elementos são gerados e ficam em memória, os mapas a geração de um novo elemento só decorre quando é necessário.

Exercício: Considere o intervalo de números desde 1 até 100. Calcule os quadrados % 31 desses números e mostre ao utilizador. Seguidamente calcule os cubos desses resultados e mostre ao utilizador. Utilize o menor número de recursos possível.

A função *filter* i

Um filtro é um caso especifico de um mapa, que mapeia objetos para existência e não-existência.

$$F : C \times f \rightarrow \{0, 1\}$$

O Python possui filtros através da função *ordem superior*

filter (função, coleção)

A função *filter* ii

Exemplo de um filtro

Excluir os elementos ímpares de uma lista:

```
def exclude_uneven (s):
    if s % 2 == 0:
        return True
    return False
```

```
» l = [i for i in range (1, 100)]
» r = filter (exclude_uneven, l)
```

O filter também é "preguiçoso"e iterável.

Exercício. Faça um programa para calcular os cubos modulo 22 de 1 a 100. Mostre ao utilizador os números menores que 13 de uma vez e os numeros menores que 21 de outra vez. Use filtros.

Função reduce i

Python permite a aplicação de construção de redução para coleções:

$$reduce : \mathcal{C} \times f \rightarrow S$$

onde S é um valor *singleton*, ou seja, um valor $\in \{\mathbb{N}, \mathbb{R}, ...\}$

O reduce, contrariamente ao map e ao filter, precisa de funções com dois argumentos.

$$f : X \times X \rightarrow S$$

Função reduce ii

Exemplo: A soma dos elementos de uma lista.

```
def sum (a, b):
    return a+b

l = [i for i in range (1, 101)]

sum_ = reduce (sum, l)
```

Função reduce iii

Como funciona o reduce? A rotina reduce é algo como:

```
def reduce (f, c):
    i = 1
    r = c [0]
    for i in range (1, len (c)):
        r = f (r, c [i])
```

Função reduce iv

Dada uma pequena lista:
l = [1, 2, 3, 4]

reduce(sum, [1,2,3,4]) = reduce (sum, [sum (1, 2), 3, 4])
= reduce (sum, [sum (sum(1,2), 3), 4])
= reduce (sum, [sum (sum (sum(1,2), 3), 4)])
= sum (sum (sum(1,2), 3), 4)
= 10

Exercício 1. Implemente um fatorial completamente preguiçoso, baseado nas funções anteriores.

Lambdas

Lambda i

Até agora foi necessário declarar funções antes de as utilizar em funções de ordem superior, como **map**, **filter** e **reduce**: sum, even, square.

Isto não é absolutamente necessário, se utilizarmos funções **lambda**: funções incógnitas (**unknown functions**)

lambda argumentos: expressão única sobre os argumentos

Exemplo:

» summation = **lambda** x, y: x + y
»rs = summation (10, 20)

»rs

Lambda ii

» 30
Exemplo da soma novamente
» r = **reduce** (summation, [i for i in range (1, 101))
» r
» 5050

Soma com uma lambda aplicada directamente (lambda inline)
» r = **reduce** (lambda x, y: x+y, [i for i in range (1, 101)])

Calculo dos quadrados com uma função lambda inline
» rm = **map** (lambda x: x * x, [i for i in range (1, 50)])

Soma dos quadrados dos números de 1 a 50
» **reduce** (lambda x, y: x+y, rm)

Exercício 2. Implemente um fibonacci completamente preguiçoso, baseado nas funções anteriores.

Perguntas?