Fundamentos de Programação @ LEIC/LETI

Semana 7

Funções revisitadas (Parte 2)

Funções de ordem superior. Funções como parâmetros. Funções Lambda. Funções como valor. Funcionais sobre listas.

Alberto Abad, Tagus Park, IST, 2018

Funções de ordem superior

- Em aulas anteriores vimos que as funções permitem-nos abstrair algoritmos e procedimentos de cálculo (abstracção procedimental).
- Em Python, tal como nas linguagens puramente funcionais, as funções são entidades de primeira ordem/classe (*first class*):
  - Podemos nomear, utilizar como parâmetro e retornar como valor.
- Isto significa que podemos expressar certos padrões de computo geral ou **funções de ordem superior**, isto é, funções que manipulam outras funções:
  - Funções como parâmetros:
    - Funções como métodos gerais (hoje)
    - Funcionais sobre listas (próximo dia)
  - Funções como valor (hoje)

Funções como parâmetros - Exemplo

```
def soma_naturais(l_inf, l_sup):
    resultado = 0
    for x in range(l_inf, l_sup + 1):
        resultado += x
    return resultado

def soma_quadrado(l_inf, l_sup):
    resultado = 0
    for x in range(l_inf, l_sup + 1):
        resultado += x*x
    return resultado
```

- Qual é o padrão comum?
- Será que podemos abstrair esse padrão comum?

Funções como parâmetros - Exemplo

- As duas funções `soma_naturais` e `soma_quadrado` diferem apenas na forma como o termo do somatório é calculado, ou seja, a função `f`:

$$\sum_{n=l_{inf}}^{l_{sup}} f(n) = f(l_{inf}) + f(l_{inf}+1) + \cdots + f(l_{sup})$$

- Por serem as funções entidades de primeira ordem, podemos passara `f` por parâmetro:

In [57]:
```
# Completar - Fazer com for e com while também
def soma(l_inf, l_sup, f):
    pass
```

Funções revisitadas

Funções como parâmetros - Exemplo

In [4]:
```
# Completar
def quadrado(x):
    pass

# Completar
def identidade(x):
    pass

# Completar
def soma(l_inf, l_sup, f):
    pass

soma(1, 10, quadrado)
```

Funções como parâmetros - Exemplo

- E se queremos ter algo mais abstracto?
- Por exemplo, podemos querer avançar o somatório com um passo diferente de somar uma unidade...

In [7]:
```
def s1(x):
    pass

def s4(x):
    pass

def soma(l_inf, l_sup, f, step):
    pass


soma(1, 10, quadrado, s4)
```

Funções Lambda (funções anónimas)

- O cálculo **lambda** é um modelo de computação universal inventado pelo matemático Alonzo Church em 1941 e que serviu de inspiração a várias linguagens de programação.
  - O cálculo lambda permite-nos modelar funções, e.g.
    $$\lambda x.\ x + 3$$
  - Para avaliar uma função em cálculo lambda escreve-se em geral:
    $$(\lambda x.\ x + 3)3$$
- Em Python, existe a possibilidade de definir funções anónimas recorrendo precisamente a uma notação inspirada no cálculo lambda, em BNF:

```
<função anónima> ::= lambda <parâmetros formais>: <expressão>
```

Funções Lambda (funções anónimas) - Exemplos

- Definição:

```
lambda x : x+3

lambda x : x*x

lambda x,y : x+y

lambda x : 2*x if x%2!=0 else x
```

- Avaliação:

```
(lambda x : x + 3)(3)
```

In [10]:
```
(lambda x: x + 1)(3)
```

Out[10]:
```
4
```

Funções Lambda (funções anónimas)

- E para que podem ser úteis estas *funções anónimas*?
- Por exemplo, para defiir as funções utiliadas como parâmetros em funções de ordem superior:

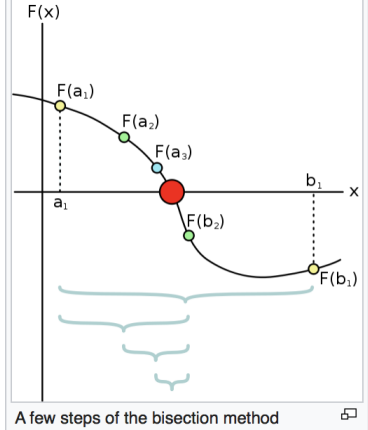
```
# soma naturais
soma(1, 10, lambda x:x, lambda x: x+1)

# soma quadrados
soma(1, 10, lambda x:x*x, lambda x: x+1)

# soma_inv_quadrados_impares!?!?
soma(1, 10, ???, ???)
```

In [11]:
```
soma(1, 10, lambda x : x*x, lambda x : x + 1)
```

Funções como parâmetros - Funções como métodos gerais

Método da Bissecção

- O método da bissecção (https://en.wikipedia.org/wiki/Bisection_method) (baseado no Teorema Bolzano (https://en.wikipedia.org/wiki/Intermediate_value_theorem)) permite-nos obter a raíz de uma função contínua $f(x)$ situada no intervalo $[a, b]$, sempre que $f(a) <= 0 <= f(b)$ ou $f(b) <= 0 <= f(a)$:



A few steps of the bisection method applied over the starting range $[a_1;b_1]$. The bigger red dot is the root Wipe of the function.

Funções como parâmetros - Funções como métodos gerais

Método da Bissecção

In [14]:
```
def metodo_bisseccao(f, a, b):

    if a > b:
        raise ValueError("metodo_bisseccao: a > b!?")

    def aproxima_raiz(f, a, b):
        while not abs(a - b) < 0.0001:
            m = a + (b - a) * 0.5
            if f(m) > 0:
                b = m
            elif f(m) < 0:
                a = m
            else:
                return m
        return a + (b - a) * 0.5

    x = f(a)
    y = f(b)
    if x < 0 < y:
        return aproxima_raiz(f, a, b)
    elif y < 0 < x:
        return aproxima_raiz(f, b, a)
    else:
        raise ValueError("metodo_bisseccao: sig(f(a)) == sig(f(b))!?")

metodo_bisseccao(lambda x: x*x - 2, 0, 2)
```

Out[14]:
```
1.414215087890625
```

Funções como valor de funções - Cálculo derivada

- As funções também podem produzir/retornar valores que são funções.
- Consideremos o cálculo da derivada de uma função de variável real `f`.
  - Por definição:
$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$
  - Substituindo $h = x - a$,
$$f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}$$
  - Se dx for um número suficientemente pequeno, podemos considerar a seguinte aproximação:
$$f'(a) \approx \frac{f(a + dx) - f(a)}{dx}$$

Funções como valor de funções - Cálculo derivada

- Definamos primeiro a função de derivada num ponto:

$$f'(a) \approx \frac{f(a+dx) - f(a)}{dx}$$

In [17]:
```
def derivada_num_ponto(f, a):
    dx = 0.00001
    return (f(a + dx) - f(a)) / dx

derivada_num_ponto(lambda x : x*x, 3)
```

Out[17]:
```
6.000009999951316
```

Funções como valor de funções - Cálculo derivada

- Podemos no entanto definir a função de **ordem superior** que retorna a derivada de f da seguinte forma (utilizando funções internas):

In [19]:
```
def derivada(f):
    dx = 0.00001
    def derivada_num_ponto(x):
        return (f(x + dx) - f(x)) / dx

    return derivada_num_ponto

g = derivada(lambda x: x*x)
g(3)
```

Out[19]:
```
6.000009999951316
```

Funções como valor de funções - Cálculo derivada

- Podemos definir a mesma função utilizando funções *lambda* :

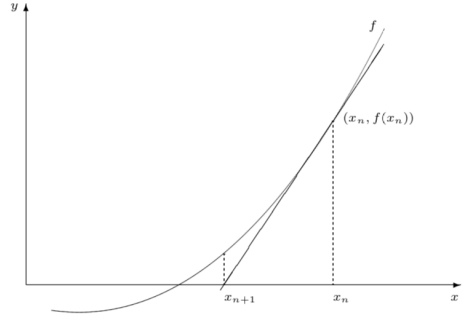
In [19]:
```
def derivada_lambda(f):
    pass

g = derivada_lambda(lambda x: x*x)
g(3)
```

Out[19]:
```
6.000009999951316
```

Funções como valor de funções - Método de Newton

- Método para determinar raízes de funções diferenciáveis:
  - Partir de uma aproximação, x_n, para a raiz de uma função f,
  - Calcular, nova aproximação: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- A função matemática que calcula uma nova aproximação é chamada transformada de Newton:

$$t_{Newton}(x) = x - \frac{f(x)}{f'(x)}$$



In [24]:
```
def transformada_newton(f):
    def t_n(x):
        return x - f(x)/derivada(f)(x)
    return t_n
```

Out[24]:
```
3.141592653678488
```

Funções como valor de funções - Método de Newton

In [25]:
```
def calcula_raizes(f, palpite):
    def bom_palpite(x):
        return abs(x) < 0.00001

    tf_N = transformada_newton(f)
    while not bom_palpite(f(palpite)):
        palpite = tf_N(palpite)

    return palpite


# calcula_raizes(lambda x : x * x * x - 2 * x - 3, 1.0)
# from math import sin
# calcula_raizes(sin, 2.0)
```

Funções como parâmetros - Funcionais sobre listas

- Quando utilizamos listas é comum fazer uso de vários funcionais, i.e., funções que recebem por parâmetro outras funções. Estes funcionais podem ser:
  - Transformadores,
  - Filtros, ou
  - Acumuladores.
- Existem já um número significativo de funcionais *built-in* em Python. Entre os mais comuns temos o `filter`, o `map` e o `reduce`, este último disponível no módulo `functools`. Estes podem ser utilizados sobre qualquer iterável e não apenas sobre listas.
- As *list comprehesions* podem ser uma alternativa conveniente aos funcionais.

Funções como parâmetros - Funcionais sobre listas

Transformadores

- Uma transformação ou transformador recebe com argumentos uma lista e uma função aplicável sobre cada elemento na lista.
- Devolve uma lista em que cada elemento resulta da aplicação da função a cada elemento da lista original.

In [32]:
```
# Versão iterativa
def transforma(f, lst):
    pass

#Versão recursiva?
# def transforma_rec(f, lst):


transforma(lambda x:x*x, [1, 2, 3, 4])
list(map(lambda x:x*x, [1, 2, 3, 4]))
```

Out[32]:
```
[1, 4, 9, 16]
```

Funções como parâmetros - Funcionais sobre listas

Transformadores - Exemplos

```
>>> transforma(lambda x : x*x, [2, 3, 5, 7])
[4, 9, 25, 49]
>>> map(lambda x : x*x, [2, 3, 5, 7])
<map object at 0x3ae62539f60>
>>> list(map(lambda x : x*x, [2, 3, 5, 7]))
[4, 9, 25, 49]
>>> map(lambda x,y : x*x + y, [2, 3, 5, 7], [1, 2, 3, 4])
<map object at 0x3ae62542128>
>>> list(map(lambda x,y : x*x + y, [2, 3, 5, 7], [1, 2, 3, 4]))
[5, 11, 28, 53]

>>> l = [2, 3, 5, 7]
>>> transforma(lambda x : x*x, l)
[4, 9, 25, 49]
>>> l
[2, 3, 5, 7]
>>> list(map(lambda x,y : x*x + y, l, l))
[6, 12, 30, 56]
>>> l
[2, 3, 5, 7]
>>>
```

Funções como parâmetros - Funcionais sobre listas

Transformadores - Alternativa com *List Comprehesions*

```
# Exemplo1
list(map(lambda x : x*x, [2, 3, 5, 7]))

# Exemplo2
l = [2, 3, 5, 7]
list(map(lambda x : x*x, l))

# Exemplo3
l1 = [2, 3, 5, 7]
l2 = [1, 2, 3, 4]
list(map(lambda x,y : x*x + y, l1, l2))
```

In [43]:
```
#Exemplo 1


#Exemplo 2


#Exemplo 3
```

```
[5, 11, 28, 53]
[5, 11, 28, 53]
```

Funções como parâmetros - Funcionais sobre listas

Filtros

- Um filtro é um funcional que recebe uma lista e um predicado aplicável sobre cada elemento da lista.
- Devolve a lista constituída apenas pelos elementos da lista original que satisfazem o predicado, i.e., os elementos para os quais o predicado retorna `True`.

In [45]:
```
# Versão iterativa
def filtra(f, lst):
    pass

# Versão recursiva?
def filtra_rec(f, lst):
    pass


filtra(lambda x : x % 2 == 0, [1, 2, 3, 4, 5, 6])
list(filter(lambda x : x % 2 == 0, [1, 2, 3, 4, 5, 6]))
```

Out[45]:
```
[2, 4, 6]
```

Funções como parâmetros - Funcionais sobre listas

Filtros - Exemplos

```
>>> filtro(lambda x : x%2 == 0, [0, 1, 1, 2, 3, 5, 8])
[0, 2, 8]
>>> filter(lambda x : x%2 == 0, [0, 1, 1, 2, 3, 5, 8])
<filter object at 0x3ae6255c5f8>
>>> list(filter(lambda x : x%2 == 0, [0, 1, 1, 2, 3, 5, 8]))
[0, 2, 8]

>>> l = [0, 1, 1, 2, 3, 5, 8]
>>> filtro(lambda x : x%2 == 0, l)
[0, 2, 8]
>>> list(filter(lambda x : x%2 == 0, l))
[0, 2, 8]
>>> l
[0, 1, 1, 2, 3, 5, 8]
>>>
```

Funções como parâmetros - Funcionais sobre listas

Filtros - Alternativa com *List Comprehesions*

```
#Exemplo 1
list(filter(lambda x : x%2 == 0, [0, 1, 1, 2, 3, 5, 8]))

#Exemplo 2
l = [0, 1, 1, 2, 3, 5, 8]
list(filter(lambda x : x%2 == 0, l))
```

In [58]:
```
#Exemplo 1


#Exemplo 2
```

Funções revisitadas

Funções como parâmetros - Funcionais sobre listas

Acumuladores

- Um acumulador recebe uma lista e um função aplicável aos elementos da lista.
- Aplica sucessivamente essa função aos elementos da lista e devolve o resultado da aplicação da mesma a todos os elementos.
- A função passada ao acumulador recebe em geral dois parâmetros, o resultado actual e o próximo elemento, devolvendo o valor resultante de incluir esse elemento no cálculo do resultado.

In [51]:
```
def acumulador(f, lst):
    pass


def acumulador_rec(f, lst):
    pass

acumulador(lambda r,x : r * x, [2, 3, 5, 7])
from functools import reduce
reduce(lambda r,x : r * x, [2, 3, 5, 7])
```

Out[51]:
```
210
```

Funções revisitadas

Funções como parâmetros - Funcionais sobre lista

Acumuladores - Exemplos

```
# Exemplo 1
l = [1, 2, 3, 7]
sum(l) # built in sum, equivalent with reduce?

# Exemplo 2
l = [True, False, False]
any(l) # built in any, equivalent with reduce?

# Exemplo 3
l = [True, False, False]
all(l) # built in all, equivalent with reduce?
```

In [62]:
```
# Exemplo 1

# Exemplo 2

# Exemplo 3
```