Fundamentos de Programação @ LEIC/LETI

Semana 6

Funções revisitadas (Parte 1)
Estructuração de funções. Scope de nomes. Programação funcional. Recursão.

Alberto Abad, Tagus Park, IST, 2018

Programação Funcional

- *Programação imperativa*: programa como conjunto de instruções em que a instrução de atribuição tem um papel preponderante.
- A **Programação funcional** é um paradigma de programação exclusivamente baseado na utilização de funções:
  - Funções calculam ou avaliam outras funções e retornam um valor/resultado, evitando alterações de estado e entidades mutáveis.
  - Não existe o conceito de atribuição e não existem ciclos.
  - O conceito de iteração é conseguido através de recursividade.

Elementos da Programação Funcional

- Na informática, diz-se que uma linguagem de programação tem funções de primeira classe (*first-class functions*) se a liguagem suporta utilizar funções como argumentos para outras funções, retornar funções como valor de outras funções, atribuir funções a variáveis, ou armazenar funções em estruturas de dados.
- O Python, tem funções de primeira classe o que nos fornece alguns dois elementos fundamentais dada programação funcional:
  - Funções internas
  - Recursão
  - Funções de ordem superior:
    - Funções como parâmetros
    - Funções como valor

Funções internas: Estrutura de uma função

- Quando vimos como definir funções observamos que o corpo de uma função poderia incluir a definição de outras funções.
- Em particular, vimos o seguinte em BNF:

```
<definição de função> ::=
    def <nome> (<parâmetros formais>): NEWLINE
    INDENT <corpo> DEDENT

<corpo> ::= <definição de função>* <instruções em função>
```

- **Em que situação isto pode ser útil?**

Funções internas, Exemplo 1

In [ ]:
```
def potencia(x, k):
    resultado = 1
    while k > 0:
        resultado = resultado * x
        k = k - 1
    return resultado

potencia(2,4)
```

- Que acontece com esta função se k for negativo?
- Como a podemos alterar para computar potências negativas?

Funções internas, Exemplo 1

In [ ]:
```
def potencia(x, k):
    resultado = 1
    if k >= 0:
        while k > 0:
            resultado = resultado * x
            k = k - 1
    else:
        k = -k
        while k > 0:
            resultado = resultado * x
            k = k - 1
        resultado = 1/resultado
    return resultado
potencia(2,-2)
```

- Muita repetição de código... vamos definir uma função auxiliar.

Funções internas, Exemplo 1

In [ ]:
```
def potencia_aux(x, k):
    resultado = 1
    while k > 0:
        resultado = resultado * x
        k = k - 1
    return resultado

def potencia(x, k):
    if k >= 0:
        return potencia_aux(x, k)
    else:
        return (1 / potencia_aux(x,-k))

potencia_aux(2,-4)
```

- Conseguimos computar potências negativas, mas o problema trasladou-se a `potencia_aux`.
- Sera que podemos *esconder* funções como `potencia_aux` que unicamente fazem sentido no âmbito de uma outra função?

Funções internas, Exemplo 1

In [ ]:
```
def potencia(x, k):
    def potencia_aux(x, k):
        resultado = 1
        while k > 0:
            resultado = resultado * x
            k = k - 1
        return resultado

    if k >= 0:
        return potencia_aux(x, k)
    else:
        return 1 / potencia_aux(x, -k)


potencia(2, 3)
```

- Neste caso temos uma função interna que nos permite estruturar melhor a nossa implementação e esconder funções que apenas fazem sentido no âmbito de uma outra função.

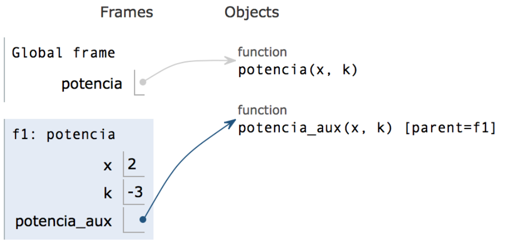
Funções internas, Exemplo 1 - Python Tutor

Python 3.6

```
def potencia(x, k):
    def potencia_aux(x, k):
        resultado = 1
        while k > 0:
            resultado = resultado * x
            k = k - 1
        return resultado

    if k >= 0:
        return potencia_aux(x, k)
    else:
        return 1 / potencia_aux(x, -k)


potencia(2, -3)
```

Edit this code

line that has just executed
next line to execute



Estrutura em blocos e domínio (*scope*) de nomes

- Este tipo de solução baseia-se no conceito de estrutura de blocos.
- O Python é uma linguagem *estruturada em blocos* onde os blocos são permitidos dentro de blocos, dentro de blocos, etc.
- O que quer que seja visível dentro de um bloco também é visível dentro dos blocos internos, mas não nos blocos externos.
- O domínio ou *scope* de um nome corresponde ao conjunto de instruções onde o nome pode ser utilizado. Falamos de domínio:
    - Local
    - Não-local:
        - Global
        - Livre (nomes definifidos em abientes/blocos exteriores aninhados)

Estrutura em blocos e domínio (*scope*) de nomes

In [ ]:
```
def teste():
    print(nome4)

nome4 = 'FP'
teste()
nome4 = 'FP avancado'
teste()
```

In [ ]:
```
nome = "fundamentos de programacao"
def teste():
    print("Dentro: " + nome)

print("Antes:" + nome)
teste()
print("Depois: " + nome)
```

- Se Python não encontra um nome no domínio local, procura nos não-locais de forma hierárquica (até chegar ao domínio global)

Estrutura em blocos e domínio (*scope*) de nomes

In [ ]:
```
nome = "fundamentos de programacao"
def teste():
    print(nome)
    nome = "programacao avancada"
    print("Dentro: " + nome)

print("Global antes: " + nome)
teste()
print("Global depois: " + nome)
```

- Alterações das associações de nomes locais não são propagadas para nomes não locais.
- Um nome não pode ser local e não-local (global) ao mesmo tempo.

Estrutura em blocos e domínio (*scope*) de nomes: *global*

- Se quisermos partilhar variáveis não locais entre funções, podemos utilizar a instrução global:

```
<instrução global>::= global <nomes>
```

In [ ]:
```
nome = "fundamentos de programacao"
def teste():
    global nome
    print("Dentro antes: " + nome)
    nome = nome + " ALTERADO"
    print("Dentro depois: " + nome)

print("Antes: " + nome)
teste()
print("Depois: " + nome)
```

- A instrução `global` não pode referir-se a parâmetros formais.
- **IMPORTANTE**: A utilização de nomes não locais (globais) deve ser evitado para manter a independência entre funções: abstracção procedimental.

Estrutura em blocos e domínio (*scope*) de nomes: *livres*

- Existem casos em que pode ser útil/importante a partilha de nomes entre blocos... **funções internas!!!**
- Exemplo `potencia`:

In [ ]:
```
def potencia(x, k):
    def potencia_aux(): # não precisamos o x como parâmetro da função interna,
                         # sendo que podemos ir a procurar a variavel não-local
        nonlocal k
        resultado = 1
        while k > 0:
            resultado = resultado * x
            k = k - 1
        return resultado

    if k >= 0:
        return potencia_aux()
    else:
        k = -k
        return 1 / potencia_aux()

potencia(2, 3)
```

Domínio (scope) de nomes: *globals*, *locals* e *nonlocals* - Python Tutor

In [ ]:

```python
a = 1

# Uses global because there is no local a
def f():
    print('Inside f() : ', a)

# Variable a is redefined as a local
def g():
    a = 2
    print('Inside g() : ', a)

# Uses global keyword to modify global a
def h():
    global a
    a = 3
    print('Inside h() : ', a)

# Variable a is redefined as a local, which is nonlocal (livre) for function j
def i():
    def j():
        print ('Inside j() : ', a)
    a = 4
    print ('Inside i() : ', a)
    j()


# Global scope
print('global : ', a)
f()
print('global : ', a)
g()
print('global : ', a)
h()
print('global : ', a)
i()
print('global : ', a)
```

Exemplos de funções internas:

- Vejamos de novo os exemplos do final da Semana 3:
    - Estruturar o código para utilizar funções internas
    - Utilizar varáveis não-locais:
- Exemplos:
    - Algoritmo da Babilónia para cálculo da raiz quadrada
    - Serie de Taylor da exponencial

Exemplo de funções internas: Algoritmo da Babilónia

- Em cada iteração, partindo do valor aproximado, p_i, para a raiz quadrada de x, podemos calcular uma aproximação ao melhor, p_{i+1}, através da seguinte fórmula:

$$p_{i+1} = \frac{p_i + \frac{x}{p_i}}{2}.$$

- Exemplo algoritmo para $\sqrt{2}$

Número da tentativa	Aproximação para $\sqrt{2}$	Nova aproximação
0	1	$\frac{1+\frac{2}{1}}{2} = 1.5$
1	1.5	$\frac{1.5+\frac{2}{1.5}}{2} = 1.4167$
2	1.4167	$\frac{1.4167+\frac{2}{1.4167}}{2} = 1.4142$
3	1.4142	...

Exemplo de funções internas: Algoritmo da Babilónia

In [ ]:
```
def raiz(x):

    def calcula_raiz(palpite):
        def bom_palpite():
            return abs(x-palpite*palpite) < 0.000000000001

        def novo_palpite():
            return (palpite + x/palpite)/2


        while not bom_palpite():
            palpite = novo_palpite()
        return palpite


    if x < 0:
        raise ValueError("raiz definida só para números positivos")
    return calcula_raiz(1)



raiz(4)
```

Exemplo de funções internas: Algoritmo da Babilónia

In [ ]:
```
def raiz(x):
    def calcula_raiz(palpite):
        def bom_palpite():
            return abs(x-palpite*palpite) < 0.0001

        def novo_palpite():
            return (palpite + x/palpite)/2

        while not bom_palpite():
            palpite = novo_palpite()
        return palpite

    if x < 0:
        raise ValueError("raiz definida só para números positivos")
    return calcula_raiz(1)


raiz(4)
```

Exemplo de funções internas: Série de Taylor, Exponencial

- Definição:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)$$

- Exemplo da aproximações da exponencial:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Exemplo de funções internas: Série de Taylor, Exponencial

In [ ]:

```
def exp_aproximada(x, delta):
    def proximo_termo():
        return x*termo/n

    n = 0
    termo = 1
    resultado = termo

    while termo > delta:
        n = n + 1
        termo = proximo_termo()
        resultado = resultado + termo

    return resultado

print("Aprox",exp_aproximada(3,0.001))
from math import exp
print("Exacto",exp(3))
```

Funções recursivas

- Uma solução recursiva para um problema depende da combinação de soluções para instâncias mais pequenas desse mesmo problema.
- Uma dada entidade é recursiva se ela for definida em termos de si própria.
- Python, tal como a maioria das linguagens de programação, suporta explicitamente soluções recursivas permitindo que as funções possam invocar-se a si mesmas.
- Em *programação funcional* e em linguagens puramente funcionais, estamos limitados ao uso de funções recursivas, não sendo possível o uso de ciclos iterativos.

Funções recursivas. Exemplos de entidades recursivas

- BNFs:

```
<nomes> ::= <nome> |
  <nome>, <nomes>
```

- Na matemática, por exemplo a Série de Fibonacci:

$$fib(n) = \begin{cases} 0 & \text{se } n = 1, \\ 1 & \text{se } n = 2, \\ fib(n-1) + fib(n-2) & \text{se } n > 2 \end{cases}$$

- O que têm em comum estas definições...
  - Um caso base ou **caso terminal**, que corresponde à versão mais simples do problema;
  - Um passo recursivo ou **caso geral**, que corresponde à definição recursiva de uma solução para o problema em termos de soluções para sub-problemas deste mas mais simples.

Funções recursivas. Exemplo 1, *potencia*

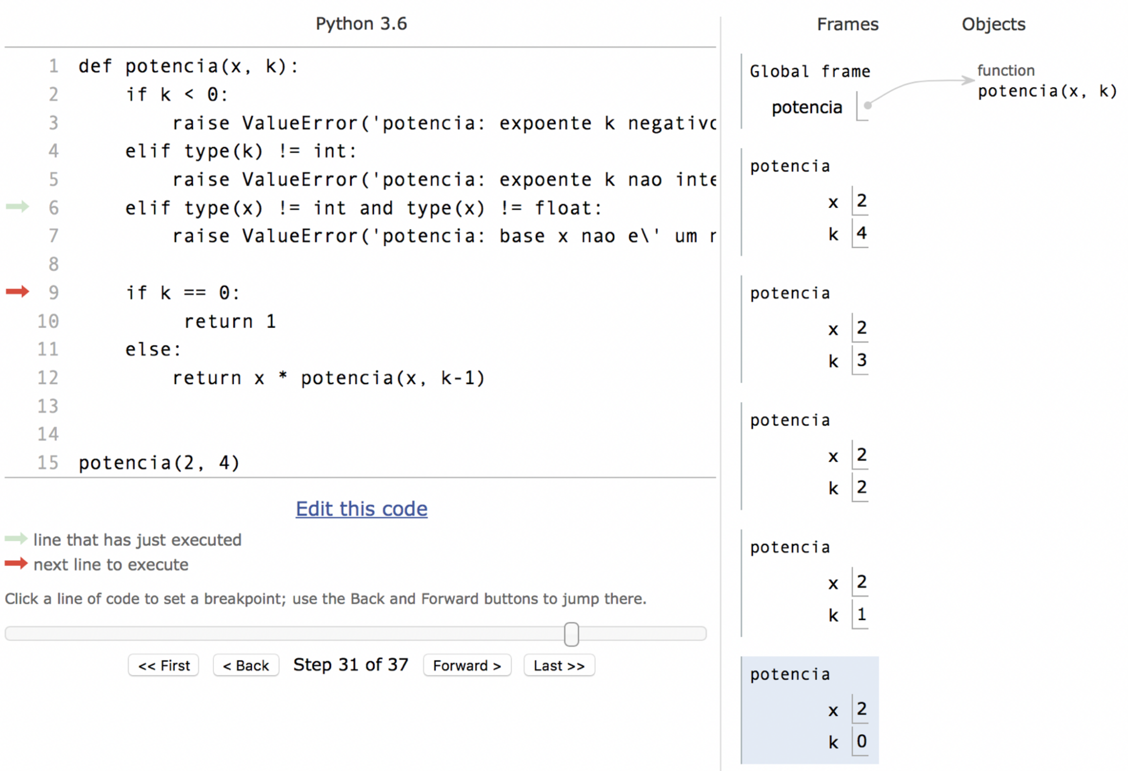
$$pow(x,n) = \begin{cases} 1 & \text{se } n = 0, \\ x * pow(x, n-1) & \text{se } n > 0 \end{cases}$$

In [24]:
```
def potencia(x, k):
    res = 1
    for i in range(1,k+1):
        res = res*x
    return res

def potencia_rec(x, k):
    if k == 0:
        return 1
    else:
        return x * potencia_rec(x, k -1)

potencia_rec(2,4)
```

Out[24]:
```
16
```

Funções recursivas. Exemplo 1, *potencia*. Python Tutor

[http://pythontutor.com/visualize.html (http://pythontutor.com/visualize.html)](http://pythontutor.com/visualize.html)



Mais exemplos de funções recursivas

- Soma de digitos
- Factorial
- Progressão aritmetica
- Maximo divisor comum
- Alisa

Funções recursivas. Exemplo 2, *soma_digitos* de um inteiro

In [26]:
```
def soma_digitos(num):
    soma = 0
    while num!=0:
        soma += num % 10
        num = num // 10
    return soma

def soma_digitos_rec(num):
    if num < 10:
        return num
    else:
        return num % 10 + soma_digitos_rec(num//10)

soma_digitos_rec(567)
```

Out[26]:
```
18
```

Funções recursivas. Exemplo 2, *soma_digitos* de um string

In [10]:

```
def soma_digitos(num):
    soma = 0
    for c in num:
        soma += int(c)
    return soma

def soma_digitos_rec(num):
    if num == '':
        return 0
    else:
        return int(num[0]) + soma_digitos_rec(num[1:])

soma_digitos_rec('567')
```

Out[10]:

```
18
```

Funções recursivas. Exemplo 3, *factorial*

- O Factorial $n! = 1 * 2 * \ldots * n$ pode também ser definido de forma recursiva:

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

In [31]:
```python
def factorial(n):
    res = 1
    for i in range(1,n+1):
        res *= i
    return res

def factorial_rec(n):
    if n == 0:
        return 1
    else:
        return n * factorial_rec(n-1)


# factorial(100000)
factorial_rec(100000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-31-2dcfd59892ac> in <module>()
     13
     14 # factorial(100000)
---> 15 factorial_rec(100000)

<ipython-input-31-2dcfd59892ac> in factorial_rec(n)
      9         return 1
     10     else:
---> 11         return n * factorial_rec(n-1)
     12
     13

... last 1 frames repeated, from the frame below ...

<ipython-input-31-2dcfd59892ac> in factorial_rec(n)
      9         return 1
     10     else:
---> 11         return n * factorial_rec(n-1)
     12
```

Funções recursivas. Exemplo 4, soma progressão aritmética

In [32]:

```
def soma(n):
    res = 0
    for i in range(1,n+1):
        res += i
    return res

def soma_rec(n):
    if n == 1:
        return 1
    else:
        return n + soma_rec(n-1)

soma_rec(10)
```

Out[32]:

```
55
```

Funções recursivas. Exemplo 5, Máximo divisor comum

1. O máximo divisor comum entre um número e zero é o próprio número: `mdc(m,0) = m`
2. Quando dividimos um número *m* por um menor *n*, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor: `mdc(m, n) = mdc(n, m%n)`

```
def mdc(m,n):
    while n != 0:
        m, n = n, m % n
    return m
```

In [33]:
```python
def mdc(m,n):
    while n != 0:
        m, n = n, m % n
    return m


def mdc_rec(m, n):
    if n == 0:
        return m
    else:
        return mdc_rec(n, m%n)


mdc_rec(12, 8)
```

Out[33]: 4

Funções recursivas. Exemplo 6, Função *alisa*

In [34]:

```
def alisa(t):
    i = 0
    while i < len(t):
        if isinstance(t[i], tuple):
            t = t[:i] + t[i] + t[i+1:]
        else:
            i = i + 1

    return t

def alisa_rec(t):
    if t == ():
        return ()
    elif type(t[0]) == tuple:
        return alisa_rec(t[0]) + alisa_rec(t[1:])
    else:
        return (t[0],) + alisa_rec(t[1:])

a = (2, 4, (8, (9, (7, ), 3, 4), 7), 6, (5, (7, (8, ))))
alisa_rec(a)
```

Out[34]:

```
(2, 4, 8, 9, 7, 3, 4, 7, 6, 5, 7, 8)
```

Funções recursivas. Exemplo 7, Longest common subsequence (LCS)

https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
(https://en.wikipedia.org/wiki/Longest_common_subsequence_problem)

Sejam duas sequencias s e t tal que $|s| = n$ e $|t| = m$, a LCS é:

$$lcs(s,t) = \begin{cases} \emptyset & \text{se } s \text{ ou } t \text{ vazio,} \\ lcs(s_{1..n-1}, t_{1..m-1}) \cup s_n & \text{se } s_n = t_m \\ longest(lcs(s, t_{1..m-1}), lcs(s_{1..n-1}, t)) & \text{se } s_n \neq t_m \end{cases}$$

Funções recursivas. Exemplo 7, Longest common subsequence (LCS)

In [ ]:

```
def lcs(a, b):
    def longest(a, b):
        if len(a) > len(b):
            return a
        else:
            return b

    if len(a) == 0 or len(b) == 0:
        return type(a)()
    elif a[-1] == b[-1]:
        return lcs(a[:-1], b[:-1]) + a[-1:]
    else:
        return longest(lcs(a[:-1], b), lcs(a, b[:-1]))
lcs('alberto', 'mara')
```