Fundamentos da Programação

Programação Funcional

Aula 20

José Monteiro

(slides adaptados do Prof. Alberto Abad)

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

- Em 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 dos elementos fundamentais da 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 observámos 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):
    pot = 1
    while k > 0:
        pot = pot * x
        k = k - 1
    return pot

potencia(2,3)
```

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

Funções Internas

Exemplo 1

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

    return pot

potencia(2,-3)
```

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

Funções Internas

Exemplo 1

In [ ]:

```
def potencia_aux(x, k):
    pot = 1

    while k > 0:
        pot = pot * x
        k = k - 1

    return pot

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

potencia(2, -3)
```

- Conseguimos calcular potências negativas, mas o problema passou para a `potencia_aux` .
- Será que podemos *esconder* funções como `potencia_aux` que unicamente fazem sentido no âmbito de uma outra função?

Funções Internas

Exemplo 1

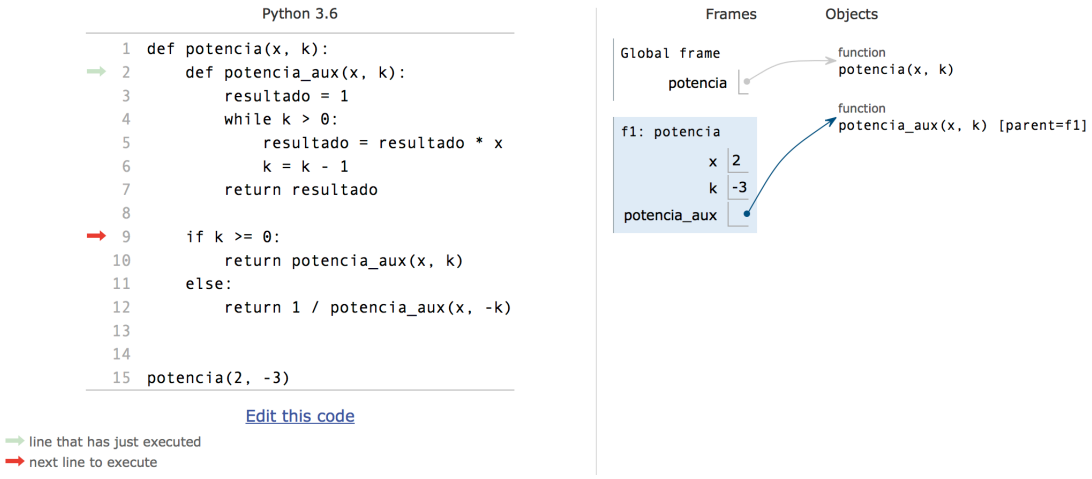
In [10]:

```
def potencia(x, k):
    def potencia_aux(x, k):
        # Função auxiliar para k > 0
        pot = 1

        while k > 0:
            pot = pot * x
            k = k - 1

        return pot

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

- 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



Domínio (*Scope*) de Nomes

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 ambientes/blocos exteriores aninhados)

Domínio (*Scope*) de Nomes

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

In [ ]:

```
def teste():
    nome4 = 'outro'
    print('Teste1', nome4)

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

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

Domínio (*Scope*) de Nomes

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

In [ ]:

```
def teste():
    print("Dentro inicio: " + nome) # global
    nome = "programacao avancada" # local
    print("Dentro fim: " + nome)

nome = "fundamentos da programacao"

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.

Domínio (*Scope*) de Nomes

Utilização das variáveis do quadro global, instrução *global*

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

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

In [ ]:

```
def teste():
    global nome # global
    print("Dentro antes: " + str(nome))
    nome = nome + " ALTERADO"
    print("Dentro depois: " + str(nome))

nome = "fundamentos da programacao"

print("Antes: " + str(nome))
teste()
print("Depois: " + str(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.

Domínio (*Scope*) de Nomes

Utilização de variáveis *livres*, instrução *nonlocal*

- Existem casos em que pode ser útil/importante a partilha de nomes entre blocos, **funções internas!!!**
  `<instrução nonlocal>::= nonlocal <nomes>`
- Exemplo `potencia` :

In [ ]:

```
def potencia(x, k):
    def potencia_aux():
        # Função auxiliar para k >= 0
        nonlocal k # equivalente ao global para variáveis livres
        pot = 1

        while k > 0:
            pot = pot * x
            k = k - 1

        return pot

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

potencia(2, -3)
```

Domínio (*Scope*) de Nomes

globals, *locals* e *nonlocals* - Python Tutor

In [ ]:

```
# 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 functi
def i():
    def j():
        print ('Inside j() : ', a)
    a = 4
    print ('Inside i() : ', a)
    j()

# Uses nonlocal keyword to modify nonlocal (livre) a
def k():
    def l():
        nonlocal a
        a = 1
        print ('Inside l() : ', a)
    a = 4
    print ('Inside k() : ', a)
    l()
    print ('Inside k() : ', a)

a = 1

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

Domínio (*Scope*) de Nomes

Exemplos de funções internas:

- Vejamos de novo os exemplos do final da aula 6:
  - Algoritmo da Babilónia para cálculo da raiz quadrada
  - Série de Taylor da exponencial
- Estruturar o código para utilizar funções internas
- Utilizar variáveis não-locais

Raiz Quadrada: Algoritmo da Babilónia

Exemplo de funções internas

- Em cada iteração, partindo do valor aproximado, p_i, para a raiz quadrada de x, podemos calcular uma aproximação 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	...

Raiz Quadrada: Algoritmo da Babilónia

Exemplo de funções internas

In [20]:

```
from math import sqrt

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

def calcula_raiz(x, palpite):
    while not bom_palpite(x, palpite):
        palpite = novo_palpite(x, palpite)
    return palpite

def bom_palpite(x, palpite):
    return abs(x - palpite*palpite) < 0.0001

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

print("Aprox",raiz(9))
print("Exacto",sqrt(9))
```

```
Aprox 3.000000001396984
Exacto 3.0
```

Raiz Quadrada: Algoritmo da Babilónia

Exemplo de funções internas

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(2)


raiz(25)
```

Série de Taylor, Função Exponencial

Exemplo de funções internas

- Definição:

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

- Exemplo da aproximação para a função exponencial:

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

Série de Taylor, Função Exponencial

Exemplo de funções internas

In [ ]:

```
def exp_aproximada(x):
    if x >= 0:
        return calc_exp_aproximada(x, 0.001)
    else:
        return 1/calc_exp_aproximada(-x, 0.001)


def calc_exp_aproximada(x, delta):
    n = 0
    termo = 1
    resultado = termo

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

    return resultado

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

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

Série de Taylor, Função Exponencial

Exemplo de funções internas

In [ ]:

```
def exp_aproximada(x):

    def calc_exp_aproximada():
        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

    delta = 0.0001
    if x >= 0:
        return calc_exp_aproximada()
    else:
        x = -x
        return 1/calc_exp_aproximada()


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

Tarefas próximas aulas

- Estudar matéria e completar exemplos
- **WARNING:** O deadline para entrega do 1º projeto é próxima sexta-feira dia 5 de Novembro até às 17h00!!