Fundamentos da Programação
LEIC/LETI

Funções revisitadas

Programação funcional. Estructuração de funções. Scope de nomes

Aula 20

Alberto Abad, Tagus Park, IST, 2021-22

Funções revisitadas

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 (**hoje**)
  - Recursão (*esta semana*)
  - Funções de ordem superior: (*próxima semana*)
    - Funções como parâmetros
    - Funções como valor

Funções revisitadas

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 revisitadas

Funções internas, Exemplo 1

In [8]:
```
def potencia(x, k):
    pot = 1
    while k > 0:
        pot = pot * x
        k = k - 1
    return pot

potencia(2,-3)
```

Out[8]: 1

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

Funções revisitadas

Funções internas, Exemplo 1

In [9]:
```
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)
```

Out[9]: 0.125

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

Funções revisitadas

Funções internas, Exemplo 1

In [10]:
```
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_aux(2, -3)
del potencia_aux
```

- 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?

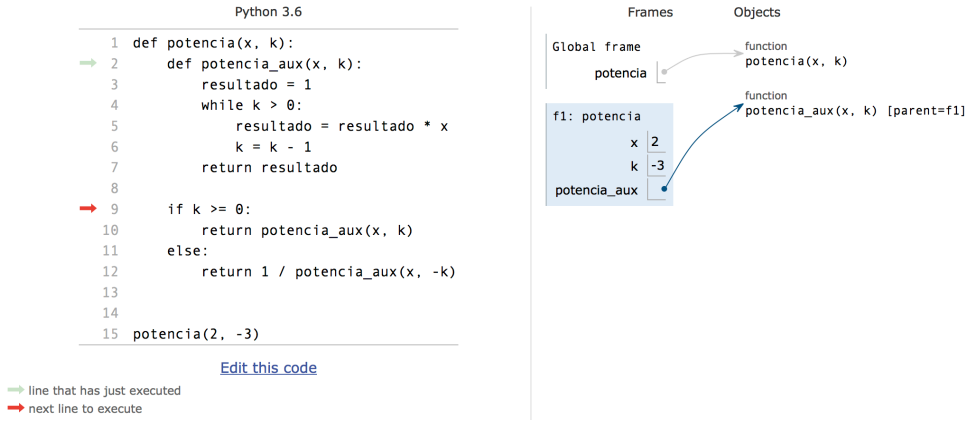
Funções revisitadas

Funções internas, Exemplo 1

In [11]:
```
def potencia(x, k):
    def potencia_aux(x, k):
        # Funcao 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



Funções revisitadas

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)

Funções revisitadas

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

In [12]:
```
def teste():
    nome4 = 'outro'
    print('Teste1', nome4)

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

```
Teste1 outro
FP
Teste1 outro
FP avancado
```

In [1]:
```
def teste():
    # nome = "outro"
    print("Dentro: " + nome)


nome = "fundamentos de programacao"
print("Antes: " + nome)
teste()
print("Depois: " + nome)
```

```
---------------------------------------------------------------------
---------
NameError                                 Traceback (most recent c
all last)
<ipython-input-1-0fc4f79d7739> in <module>
      4
      5
----> 6 print("Antes: " + nome)
      7 teste()
      8 nome = "fundamentos de programacao"

NameError: name 'nome' is not defined
```

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

Funções revisitadas

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

In [14]:
```
nome = "fundamentos de programacao"

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


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

```
Global antes: fundamentos de programacao
Dentro inicio: fundamentos de programacao
Dentro fim: fundamentos de programacao
Global depois: fundamentos de programacao
```

- 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.

Funções revisitadas

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 [2]:
```
def teste():
    global nome
    print("Dentro antes: " + nome)
    nome = nome + " ALTERADO"
    print("Dentro depois: " + nome)

nome = "fundamentos de programacao"
print("Antes: " + nome)
teste()
print("Depois: " + nome)
```

```
Antes: fundamentos de programacao
Dentro antes: fundamentos de programacao
Dentro depois: fundamentos de programacao ALTERADO
Depois: fundamentos de programacao ALTERADO
```

- 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.

Funções revisitadas

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 [4]:
```
def potencia(x, k):
    def potencia_aux(x,k):
        # Funcao auxiliar para k >= 0
        # nonlocal k # equivalente ao global para variaveis livres
        pot = 1

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

        return pot

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

x = 5
potencia(2, -3)
```

```
3
```

Out[4]:
```
0.125
```

- **Opcional/avançado**: A instrução `nonlocal` é o equivalente a `global` para nomes livres.

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

In [19]:

```
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()



# 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)

# 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)
```

```
global :  1
Inside f() :  1
global :  1
Inside g() :  2
global :  1
Inside h() :  3
global :  3
Inside i() :  4
Inside j() :  4
global :  3
Inside k() :  4
Inside l() :  1
Inside k() :  1
global :  3
```

Funções revisitadas

Exemplos de funções internas:

- Vejamos de novo alguns dos exemplos das funções da Semana 2:
  - 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	...

Funções revisitadas

Exemplo de funções internas: Algoritmo da Babilónia

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
```

Funções revisitadas

Exemplo de funções internas: Algoritmo da Babilónia

In [21]:
```
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)
```

Out[21]:
```
2.000000000000002
```

Funções revisitadas

Exemplo de funções internas: Série de Taylor, Exponencial

- Definição:

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

- Exemplo da aproximações da exponencial:

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

Funções revisitadas

Exemplo de funções internas: Série de Taylor, Exponencial

In [22]:

```
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))
```

```
Aprox 0.04978723773979558
Exacto 0.049787068367863944
```

In [23]:

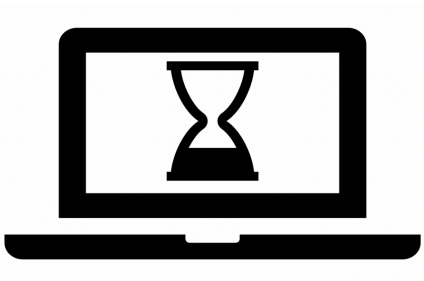
```
def exp_aproximada(x):

    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))
```

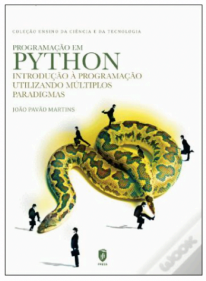
```
Aprox 0.04978710174443553
Exacto 0.049787068367863944
```

Tarefas próxima aula

- Estudar matéria e completar exemplos
- Ler seções 7.1 e 7.2 do livro (recursão de operações adiadas)
- Nas aulas práticas esta semana:
  - F4: Dicionários + TADs
  - L8: Ficheiros
  - L9: Recursão

- **WARNING:** O deadline para entrega do projeto 1 é esta sexta-feira dia 5 de Novembro até ás 17h00!!





In [ ]: