Fundamentos da Programação
LEIC/LETI

Funções de ordem superior

Funcionais sobre listas

Aula 25

Alberto Abad, Tagus Park, IST, 2021-22

Funções de ordem superior

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 computação geral a través de funções que manipulam outras funções, conehcios como **funções de ordem superior**:
  - Funções como parâmetros:
    - Funções como métodos gerais (ontem e amanhã)
    - **Funcionais sobre listas (hoje)**
  - Funções como valor (amanhã)

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 `map`, o `filter` 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 de ordem superior

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 [4]:
```python
# Versão iterativa
def transforma(f, lst):
    res = type(lst)()
    for e in lst:
        res = res + [f(e)]

    return res

#Versão recursiva?
def transforma_rec(f, lst):
    if not lst:
        return type(lst)()
    else:
        return [f(lst[0])] + transforma_rec(f, lst[1:])


transforma(lambda x:x*x, [1, 7, 3, 4, 2])
```

Out[4]:
```
[1, 49, 9, 16, 4]
```

Funções como parâmetros - Funcionais sobre listas

Transformadores - Exemplos

```
>>> l = [2, 3, 5, 7]
>>> transforma(lambda x : x*x, l)
????
>>> l
????

>>> transforma(abs, range(-5,6))
????
>>> transforma(lambda x: 0 if x < 0 else x, range(-5,6))
????
>>> transforma(int, '1234546')
????

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

```
In [284]: list(map(lambda x,y : (x,y), [2, 3, 5, 7], [1, 2, 3, 4]))
Out[284]: [(2, 1), (3, 2), (5, 3), (7, 4)]
```

Funções de ordem superior

Funções como parâmetros - Funcionais sobre listas

Transformadores - Alternativa com *List Comprehesions*

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

# Exemplo2
transforma(abs, range(-5,6))

# Exemplo3
transforma(lambda x: 0 if x < 0 else x, range(-5,6))

# Exemplo4
transforma(int, '1234546')
```

In [288]:
```
#Exemplo 1
[x*x for x in [2, 3, 5, 7]]
# [f(x) for x in lst]

#Exemplo 2
[abs(x) for x in range(-5,6)]

#Exemplo 3
[0 if x<0 else x for x in range(-5,6)]


#Exemplo 4
[int(x) for x in '1234546']
```

Out[288]:
```
[1, 2, 3, 4, 5, 4, 6]
```

Funções de ordem superior

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 [343]:
```
# Versão iterativa
def filtra(f, lst):
    res = []
    for e in lst:
        if f(e):
            res += [e]

    return res

# Versão recursiva?
def filtra_rec(f, lst):
    if not lst:
        return []
    else:
        if f(lst[0]):
            return [lst[0]] + filtra_rec(f, lst[1:])
        else:
            return filtra_rec(f, lst[1:])


filtra_rec(lambda x : x%2==0, [1, 2, 3, 4, 5, 6, -2, 1])
```

Out[343]:
```
[2, 4, 6, -2]
```

Funções como parâmetros - Funcionais sobre listas

Filtros - Exemplos

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

>>> filtra(lambda x: x%3==0, range(-10,11))
????
>>> filtra(lambda x: 'A' <= x <= 'Z', 'aAbBcC')
????
>>> filtra(lambda x: type(x) == int or type(x)==float, [1, 'a', (), True, 3.14, 2])
????

>>> filtra(lambda x : x%2 == 0, l)
[0, 2, 8]
>>> filter(lambda x : x%2 == 0, l)
<filter object at 0x3ae6255c5f8>
>>> list(filter(lambda x : x%2 == 0, l))
[0, 2, 8]
```

```
In [308]: list(filter(lambda x : x%2 == 0, l))
```

```
Out[308]: [0, 2, 8]
```

Funções como parâmetros - Funcionais sobre listas

Filtros - Alternativa com *List Comprehesions*

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

#Exemplo 2
filtra(lambda x: x%3==0, range(-10,11))

#Exemplo 3
filtra(lambda x: 'A' <= x <= 'Z', 'aAbBcC')

#Exemplo 4
filtra(lambda x: type(x) == int or type(x)==float, [1, 'a', (), True, 3
.14, 2])
```

In [313]:
```
#Exemplo 1
# [x for x in lista if f(x)]
[x for x in [0, 1, 1, 2, 3, 5, 8] if x%2==0]

#Exemplo 2
[x for x in range(-10,11) if x%3 == 0]

#Exemplo 3
cad = 'OLA, bom dia. Esta tudo bem?'
[x for x in cad if "A" <= x <= "Z"]

#Exemplo 4
# [x for x in [1, 'a', (), True, 3.14, 2] if type(x) == int or type
(x)==float]

cad = 'OLA, bom dia. Esta tudo bem?'
[x for x in cad if x in 'aAbBcC']
```

Out[313]:
```
['A', 'b', 'a', 'a', 'b']
```

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 [323]:
```
def acumulador(f, lst):
    if len(lst) == 0:
        raise ValueError('acumulador: lista vazia')
    res = lst[0] # Porque!?!
    for e in lst[1:]:
        res = f(res, e) # Ordem importante?
    return res


##
def acumulador_rec(f, lst):
    if len(lst) == 0:
        raise ValueError('acumulador: lista vazia')
    else:
        if len(lst) == 1:
            return lst[0]
        else:
            return f(acumulador_rec(f, lst[:-1]), lst[-1]) # Porque
?

l = [2, 3, 7, 1]
acumulador(lambda a,b : a / b, l)
```

Out[323]:
```
0.09523809523809523
```

- **NOTA**: Este acumulador (assim como o `reduce` das `functools`) é de esquerda para direita, também comhecido como **Fold Left** (`foldl`): https://www.burgaud.com/foldl-foldr-python (https://www.burgaud.com/foldl-foldr-python)

Funções como parâmetros - Funcionais sobre listas

Acumuladores - Exemplos

```
>>> l = [2, 3, 5, 7]
>>> acumulador(lambda r,x : r * x, l)
????
>>> l
[2, 3, 5, 7]

l = list('Fundamentos')
>>> l
['F', 'u', 'n', 'd', 'a', 'm', 'e', 'n', 't', 'o', 's']
>>> acumulador(lambda x, y: x+y, l)
????

>>> from functools import reduce
>>> reduce(lambda r,x : r * x, [2, 3, 5, 7])
???
>>> reduce(lambda a,b: a if (a > b) else b, [47,11,42,102,13])
???
```

```
In [335]: acumulador(lambda a,b: a if (a < b) else b, [47,11,42,102,13])
```

```
Out[335]: 11
```

Funções como parâmetros - Funcionais sobre listas

Acumuladores - Exemplos *buit-in*

```
# 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?

# Exemplo 4
l = [3, 1, 7, 0]
min(l) # built in min, equivalent with reduce?
```

In [344]:
```
# Exemplo 1
l = [1, 2, 3, 7]
acumulador(lambda a,b: a+b, l)

# Exemplo 2
l = [False, True, False]
acumulador(lambda a,b: a or b, l)

# Exemplo 3
l = [True, True, True]
acumulador(lambda a,b: a and b, l)

# Exemplo 4
l = [3, 1, 7, 0]
acumulador(lambda a,b: a if (a < b) else b, l)
```

Out[344]: 0

Funções como parâmetros - Funcionais sobre listas

Exemplos combinados

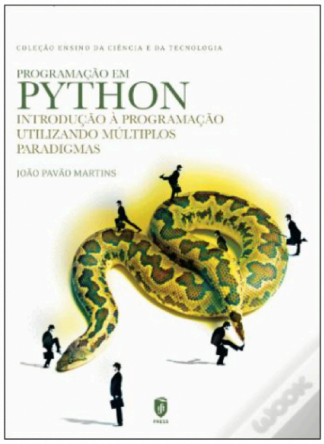
```
# Verifica se todos os elementos de uma lista são pares
def todos_pares(lista):
    return reduce(lambda x, y: x and y, map(lambda x: x%2==0, lista))

# Soma quadrados impares
def soma_quadrados_impares(lista):
    return acumulador( lambda x,y;x+y, transforma(lambda x: x*x, (filtr
a(lambda x: x%2!=0, lista))))

# Soma dos dígitos de um número
def soma_digitos(num):
    return reduce(lambda x, y: x +y, [int (e) for e in str(num)])

# Converte codigo binario (str) a decimal equivalente
def converte(codigo):
    return acumulador(lambda r, x: 2*r + x, transforma(int,codigo))
```

In [345]:
```
def ex1a(lst):
    return all([x%2==0 for x in lst])

def ex1b(lst):
    return acumulador(lambda x,y: x and y, transforma(lambda x: x%2
==0, lst))

def ex1c(lst):
    return filtra(lambda x:x%2==0, lst) == lst


def ex2a(lst):
    return acumulador(lambda a,b: a+b, transforma(lambda x:x*x, fil
tra(lambda x: x%2 != 0, lst)))

def ex2b(lst):
    return sum(x*x for x in lst if x%2 != 0)
```

Funções de ordem superior

Tarefas próxima aula

- Estudar matéria de funcionais sobre listas:
  - Completar exemplos
- Próxima aula nas aulas teóricas ==> Funções como parâmetros e como valor



In [ ]: