UFABC

Introdução ao Python

Prof. Fabrício Olivetti de França
Universidade Federal do ABC

Python

Linguagem interpretada criada em 1991

Objetivos:

- enfatizar a legibilidade do código e,
- encurtar o tamanho dos códigos.

UFABC

Python

Multiparadigma:

- Orientada a objetos
- Programação estruturada (imperativa)
- Funcional (suporte parcial)

UFABC

Declarações de Variáveis

Tipagem automática de acordo com valor atribuído:

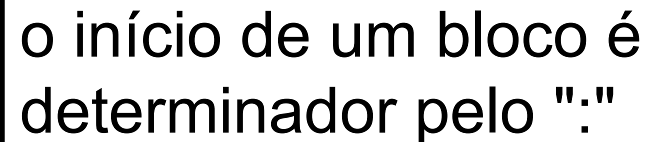
```
x = 10          # x é do tipo `int`
x = 1.4         # agora x é do tipo `float`
x = "Ola"       #  x se tornou `str`
x = [1, 4, 10]  # e x é do tipo `list`
```

UFABC

Blocos de comandos

Ao invés do tradicional { } para determinar um bloco de comandos, usa-se a identação:

```
# a instrução 3 é a única que não pertence ao bloco condicional
if x % 2 == 0:
    instrução 1
    instrução 2
instrução 3
```



UFABC

Funções

As funções são definidas pela instrução "def":

```
def soma(x,y):
    return x+y
```

UFABC

Funções

E podemos retornar quantos valores necessários:

```
def somaEmult(x,y):
    return x+y, x*y
```

UFABC

Variáveis (I)mutáveis

A maioria dos tipos de dados do Python é do tipo imutável:

```
def inc(x):
    x = x+1
    return x
```

```
x = 10
inc(x)
print x
>> 10
```

UFABC

Variáveis (I)mutáveis

Mas alguns tipos são mutáveis e passados como referência:

```
def inc(lista):
    lista[0] = -1
    return lista
```

```
lista = [1, 2, 3]
inc(lista)
print lista
>> [-1, 2, 3]
```

UFABC

Listas como containers

As listas, diferente das arrays, são containers de objetos:

```
lista = [1, 2.3, False, "ola mundo"]
```

UFABC

Iteradores

Alguns tipos no Python são iteráveis:

```
for x in lista:
    print x
```

```
for letra in string:
    print letra
```

```
f = open('arquivo')
for linha in f:
    print linha
f.close()
```

UFABC

Dicionários

Equivalente a tabela hash:

```
patas = {}
patas['gato'] = 4
patas['homem'] = 2
patas['passaro'] = 2
```

UFABC

Dicionários

```
print patas['homem']
>> 2

print patas.keys()
>> ['gato','homem','passaro']

print patas.values()
>> [4,2,2]
```

UFABC

Dicionários

```
for chave, valor in patas.items():
    print '{} possui {} patas'.format(chave, valor)

>> 'gato possui 4 patas'
   'homem possui 2 patas'
   'passaro possui 2 patas'
```

UFABC

IPython Notebook

Um "caderno de atividades" que pode misturar códigos em Python e textos (com suporte a Latex).

http://nbviewer.ipython.org/github/folivetti/PIPYTHON/blob/master/aula0.ipynb

IPython Notebook

Muito mais produtivo para análise de dados, combina:

- IDE para escrita de código-fonte
- Avaliação dos resultados (de forma parcial)
- Documentação textual
- Plotagem de gráficos

UFABC

Numpy e Álgebra Linear

Biblioteca para computação numérica e científica, suporte a vetores e matrizes.

Tipo básico: **ndarray**, array multidimensional que possui elementos de apenas um tipo.

UFABC

ndarray

```
import numpy as np

X = np.array( [1,2,3,4] )
X+1
>> [2,3,4,5]
X*2
>> [2,4,6,8]
```

UFABC

Operações básicas

Por padrão, as operações básicas são feitas elemento-a-elemento:

```
X = np.array( [1,2,3,4] )
Y = np.array( [2,3,4,5] )
X*Y
>> ndarray( [2,6,12,20] )
```

UFABC

Métodos

Os ndarray possuem métodos para operações mais avançadas:

```
X = np.array( [1,2,3,4] )
X.sum()
>> 10
print X.max(), X.min(), X.argmax(), X.argmin()
>> 4, 1, 3, 0
```

UFABC

Produto Interno

O produto interno de dois vetores é calculado como a soma do produto elemento-a-elemento entre dois vetores:

$$x . y = \sum_{i=1..n} x_i . y_i$$

UFABC

Produto Interno

Representa o produto da intensidade dos vetores multiplicados pelo cosseno do ângulo entre eles:

$$X \, . \, Y = ||X|| \, ||Y|| \cos \theta$$

UFABC

Produto Interno

O produto interno de dois vetores pode ser calculado com a função `dot()`:

```
X = np.array( [1,2,3,4] )
Y = np.array( [2,3,4,5] )
np.dot(X,Y)
>> 40
```

UFABC

Matrizes

Para operações com matrizes utilizamos o tipo `matrix` da biblioteca Numpy:

```
A = np.matrix([[1,2,3,4],[5,6,7,8]])
print A
```

```
[ [1,2,3,4]
  [5,6,7,8]]
```

UFABC

Matrizes

Algumas operações comuns:

A.T → transposta de A

np.linalg.inv(A) → inversa de A

UFABC

Produto de matrizes entre arrays

O produto externo entre dois vetores pode ser utilizado para o cálculo de multiplicação de matrizes entre duas `ndarray`:

```
X = np.array([1,2,3])
Y = np.array([1,1,1])
np.outer(X,Y)
```

```
>> [ [1,1,1]
     [2,2,2]
     [3,3,3]]
```

UFABC

Slices

Os ndarrays possuem uma indexação flexível:

X[0] → primeiro elemento

X[:10] → 10 primeiros elementos

X[-10:] → 10 últimos elementos

X[2:10] → elementos de índice 2 até 9

X[[1,3,5]] → elementos 1, 3 e 5

UFABC

Programação Funcional

Declarativo Vs. Imperativo:

```
Para x em X:
    if x%2==0:
        Y.append(x)
    else:
        Y.append(x+1)
```

UFABC

Programação Funcional

Declarativo Vs. Imperativo:

[x se x%2==0 senão x+1 para x em X]

ou

incSe = lambda x: x if x%2==0 else x+1
map(incSe, X)

UFABC

List comprehension

```
X = [1,2,3,4,5,6,7,8,9]

Xquad = [ x*x for x in X ]

print Xquad
>>[1,4,9,16,25,36,49,64,81]
```

UFABC

Generators

Apenas guarda as instruções para gerar o próximo elemento da lista:

```
Xquad = ( x*x for x in X)
print Xquad
>> <generator object <genexpr> at 0xb1fd7284>
```

UFABC

Generators

Como não armazena os dados na memória se torna eficiente e permite "guardar" listas de tamanho infinito.

```
for x in Xquad:
   print x
   if x > 40:
       break
>> 1 4 9 16 25 36
```

UFABC

Encadeando Generators

Podemos também encadear múltiplos generators para construir um pipeline de transformações:

```
XquadImpar = ( x+1 for x in Xquad if x%2!=0 )
```

(2, 4, 10, 16, 26, …)

UFABC

Encadeando Generators

Ao requisitar um elemento ele aplica as operações:

```
y = (x*x)
if y % 2 != 0:
    y = y + 1
```

UFABC

Função lambda

Funções lambda são funções anônimas de uma única instrução para aplicar operações simples.

lambda <entradas>: <saidas>

UFABC

Função lambda

Funções lambda são funções anônimas de uma única instrução para aplicar operações simples.

```
quad = lambda x: x*x
quad(10)
>> 100
```

UFABC

Função lambda

Funções lambda são funções anônimas de uma única instrução para aplicar operações simples.

```
quad = lambda x,y: x+y
quad(10,20)
>> 30
```

UFABC

Função lambda

Funções lambda são funções anônimas de uma única instrução para aplicar operações simples.

```
quad = lambda x,y: (x+y, x*y)
quad(10,20)
>> (30, 200)
```

UFABC

Map, reduce, filter

Três funções bastante utilizadas em programação funcional:

map(função, lista) → aplica função em cada elemento da lista

Xquad = map(quad, X)

UFABC

Map, reduce, filter

reduce(função, lista) → aplica função em pares de elementos até que reste apenas um elemento

```
reduce(lambda x,y: x+y, X)
>> 45
```

UFABC

Map, reduce, filter

filter(função, lista) → retorna apenas elementos em que a aplicação da função retorne verdadeiro.

```
filter(lambda x: x%2==0, X)
>> [2,4,6,8]
```

UFABC

Funções de Alta Ordem

Imagine que você tem uma função soma(x) que retorna uma função somaX(y) → x+y.

Com essa função podemos extrapolar para criar uma função genérica que some qualquer quantidade de variáveis.

UFABC

Funções de Alta Ordem

```
Soma2 = lambda a,b: Soma(a)(b)

Soma3 = lambda a,b,c: Soma(Soma(a)(b))(c)
```

sucessivas aplicações de Soma() permite realizar a somatória de uma lista de valores.

UFABC

Exemplo

Preciso calcular a somatória de uma lista de números multiplicados por 5, mas apenas daqueles cujo resultado é um número par.

Exemplo

```
mult5 = lambda x: x*5

par = lambda x: x%2==0

soma = lambda x,y: x+y
```

UFABC

Exemplo

```
# multiplicando por 5
Y = map(mult5, X)

# filtrando pares
Z = filter(par, Y)

# somando tudo
W = reduce(soma, Z)
```

UFABC

No PySpark

O tipo RDD do Spark contém map, reduce, filter, de tal forma que podemos:

```
W = (X
        .map(mult5)
        .filter(par)
        .reduce(soma)
        )
```

UFABC

Co-Rotinas

"Subrotinas são casos especiais de...co-rotinas" - D. Knuth

Co-rotinas são rotinas que permitem a suspensão e continuação da execução em qualquer instante de tempo.

UFABC

Co-Rotinas

Os generators do Python são exemplos de co-rotinas:

```
def GeraPar(x):
    if x%2!=0:
        x += 1
    while True:
        yield x
        x += 2
```

UFABC

Co-Rotinas

```
for x in GeraPar(2):
    print x

>> 2
>> 4
>> 6
>> 8
...
```

UFABC

Co-Rotinas

Quando chamamos a função GeraPar() ele executa as instruções e suspende sua execução retornando o valor x.

Ao chamar a função novamente, ou através de iteradores ou com o método __next()__, a função retoma de onde parou.

UFABC

Co-Rotinas

Uma cadeia de transformações de uma lista faz uso da co-rotina pelo seu poder de paralelização.

Ex.: vamos contar as palavras em um texto.

UFABC

Co-Rotinas

Sequencial:

- Tokenizar o texto em palavras
- Gerar tuplas (palavra, 1)
- Agrupar essas tuplas pelo primeiro elemento
- Somar os valores do segundo elemento
- Imprimir resultado

UFABC

Co-Rotinas

Nas co-rotinas todos esses processos ocorrem em paralelo como um fluxo contínuo.

UFABC

Co-Rotinas

TEXTO → lista de palavras → tuplas → contagem

Cada palavra extraída do texto já passa para a transformação em tuplas, não precisa esperar que todas as outras palavras sejam extraídas do texto.

UFABC