Mini-curso Python

Yguaratã C. Cavalcanti

yguarata@gmail.com

28/02/2007

Sumário

- Introdução
- Interpretador Python
- Manipulando Variáveis
- Strings
- Estrutura de dados
  - Listas, dicionários e tuplas
- Estruturas de controle
  - if, for, função range(), while
- Funções
- Módulos e pacotes

Sumário(2)

- Orientação a objetos
  - classes, herança, etc
- Exceções

Python - Introdução

- Linguagem de programação com suporte a vários paradigmas:
  - estruturado
  - orientado a objetos
  - funcional
- Multiplataforma (portabilidade)
- Tipagem dinâmica (não é preciso declarar tipos)
- Compilação em tempo de execução (*on the fly*)
- Fortemente tipada
- Possui estruturas de dados de alto nível (listas, dicionários...)

Python - Introdução(2)

- Sintaxe bastante clara
- Oferece suporte a módulos e pacotes
- Ampla biblioteca padrão
- Tratamento de exceções
- Ótima linguagem para:
  - prototipagem
  - RAD (*Rapid Application Development*)
  - integração entre aplicações, componentes

Interpretador Python

Interpretador python como uma calculadora

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

Manipulando Variáveis

Em Python não se declara tipos de variáveis; os tipos são inferidos dinamicamente.

```
>>> umInteiro = 10
>>> type(umInteiro) # type() é uma função build-in
<type 'int'>
>>> umReal = 10.1
>>> umLong = 5L
>>> x = y = z = 0
>>> umInteiro * x
0
>>> umInteiro + umReal
20.1
```

Strings

Cadeia de caracteres imutável

```
>>> "Uma string válida"
“Uma string válida”
>>> 'Outra string válida'
'Outra string válida'
>>> "Boa" + " tarde"
'Boa tarde'
>>> print "Mais de \n uma linha"
Mais de
 uma linha
>>> x = 'Uma string válida'
>>> x[0]
'U'
>>> x[0:3]
'Uma'
>>> x[3:]
' string válida'
```

Strings(2)

```
>>> x = 'Uma string válida'
>>> x[-1] #acesso a partir do final
'a'
>>> x[:-2]
'Uma string vali'
>>> x[-4:]
'lida'
>>> print '''
Outra maneira de se
criar strings: com """ ou '''
'''
```

Estruturas de dados - Listas

```
>>> lista = ['maçã', 'banana', 'abacaxi', 'ovo']
>>> lista[0]
'maçã'
>>> lista[1:3] #selecionando pedaços
['banana', 'abacaxi']
>>> lista[0] = 'limão' #modificando
>>> lista
['limão', 'banana', 'abacaxi', 'ovo']
>>> len(lista)
4
>>> lista[0:2] = ['galinha', 'livro'] #substituindo vários itens
>>> lista
['galinha', 'livro', 'abacaxi', 'ovo']
```

Estruturas de dados - Lista(2)

```
>>> lista1 = ['maçã', 'banana', 'abacaxi', 'pêra']
>>> lista2 = ['mamão', 'kiwi']
>>> lista1 + lista2
['maçã', 'banana', 'abacaxi', 'pêra', 'mamão', 'kiwi']
>>> lista3 = [lista1, lista2]
>>> lista3[0][1]
'banana'
>>> listaMix = [1, 2, 'texto', open('someFile.txt', 'w')]
```

Estruturas de dados em Python podem comportar qualquer tipo de objeto.

Estruturas de dados - Dicionários

```
>>> dicionario = {'chave1' : 1, 'chave2' : 'um texto'}
>>> dicionario['chave1']
1
>>> dicionario['novaChave'] = [1, 2, 3, 4]
>>> dicionario
{'chave1': 1, 'novaChave': [1, 2, 3, 4], 'chave2': 'um texto'}
>>> dicionario['chave1'] = 'novo valor'
>>> dicionario
{'chave1': 'novo valor', 'novaChave': [1, 2, 3, 4], 'chave2': 'um
  texto'}
```

Estruturas de dados - Tuplas

Seqüência de objetos imutável

```
>>> tupla = (1, 2, 3, 4) #nova tupla
>>> tupla[0] #acessando um item da tupla
1
>>> tupla + (5, 6, 7, 8) #somando duas tuplas
(1, 2, 3, 4, 5, 6, 7, 8)
>>> tupla[0] = 20 #tuplas em Python são imutáveis
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

Estruturas de controle - if

```
>>> x = int(raw_input("Digite um número inteiro: "))
>>> if x < 0:
...         x = 0
...         print 'Negativo modificado para zero'
...     elif x == 0:
...         print 'Zero'
...     elif x == 1:
...         print 'Um'
...     else:
...         print 'Maior que um'
```

Em Python, qualquer coisa diferente de None e False é verdadeiro

```
>>> if 'uma string qualquer' and 20:
...         print 'Verdadeiro'
```

Estruturas de controle - for

```
>>> # Mostrando o tamanho de algumas strings:
...     a = ['gato', 'janela', 'erecomp']
>>> for x in a:
...         print x, len(x)
...
gato 4
janela 6
erecomp 7
```

Estruturas de controle – função range()

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]

>>> a = ['Chuck Norris', 'só', 'programa', 'em', 'Python']
>>> for i in range(len(a)):
...         print i, a[i]
...
0 Chuck Norris
1 só
2 programa
3 em
4 Python
```

Estruturas de controle - while

```
>>> while True:
...         x = int(raw_input('Digite 0 para sair: '))
...         if x is 0: # x == 0
...             break
...
```

Funções

Funções são criadas utilizando a palavra reservada *def*

funcao01.py

```
def funcao01(a, b):
    c = a + b
    print "O resultado de a + b é ", c

funcao01(2, 2)
```

prompt

python funcao01.py

O resultado de a + b é 4

Funções(2)

Parâmetros podem ter valor padrão

funcao02.py

```
def funcao02(a, b = 2, texto = 'O valor de a + b é '):
    c = a + b
    print texto, c

funcao02(2, 2)
funcao02(3)
```

prompt

python funcao02.py

```
O resultado de a + b é  4
O resultado de a + b é  5
```

Funções(3)

Pode-se passar apenas o parâmetro desejado

funcao03.py

```
def funcao03(a, b = 2, texto = 'O valor de a + b é ', msgErro = None):
    if a < 0:
        print msgErro
    else:
        c = a + b
        print texto, c

funcao03(2, 2)
funcao03(-3, -4, msgErro = 'Valor menor que zero!')
```

Obs.: após valores padrões não podem haver outros parâmetros senão parâmetros com valor padrão

prompt

python funcao03.py

```
O valor de a + b é  4
Valor menor que zero!
```

Funções(4)

Parâmetros arbitrários

funcao04.py

```
def funcao04(a, *args, **kw):
    print "Tupla de argumentos: ", args
    print "Dicionários de argumentos: ", kw

funcao04(1, 2, 3, 4, b=3, c=4, texto='ultimo parametro chave-valor')
```

prompt

python funcao04.py

```
Tupla de argumentos:  (2, 3, 4)
Dicionários de argumentos:  {'c': 4, 'b': 3, 'texto': 'ultimo parametro
chave-valor'}
```

Mais exemplos: funcao05.py, funcao06.py e funcao07.py

Módulos e Pacotes

- Módulos são arquivos python que possuem definições de função, classes, variáveis etc.

```
>>> import os #importa o módulo todo
>>> print os.name
... posix
>>> from os import name #importa apenas uma definição
>>> print name
... posix
```

- função dir() - lista todas as definições de um módulo.
  - >>> dir(os)

Módulos e Pacotes(2)

- Pacotes são uma maneira de definir *namespaces*, em Python, usando a notação de ponto.
  - Ex.: A.B referencia o sub-módulo B em um pacote chamado A

```
net/                        Pacote raiz
    __init__.py             Inicializa o pacote net
    compor/                 Sub-pacote
        __init__.py
        Component.py
    tests/                  Sub-pacote
        __init__.py
        ComponentTest.py

exemplo de acesso: import net.tests
```

Orientação o Objetos

- Classes

```
class AlgumaClasse:

    def __init__(self):
        self.atributoA = 'Atrubuto A'
        self.atributoB = 20
        self.atrubutoC = None

    def umMetodo(self, arg1):
        print arg1 + self.atributoB

algumaClasse = AlgumaClasse()

print algumaClasse.atributoA
algumaClasse.umMetodo(2)
```

Orientação o Objetos

- Herança

```
class AlgumaClasse(ClasseA):

    <definição-1>
    .
    .
    .
    <definição-N>
```

- Herança múltipla

```
class AlgumaClasse(ClasseA, ClasseA):

    <definição-1>
    .
    .
    .
    <definição-N>
```

Orientação o Objetos

- Atributos e métodos privados

Atributos provados são inseridos utilizando dois '_' antes do nome do atributo

```
class AlgumaClasse:

    def __init__(self):
        self.__privado = None

    def getPrivado(self):
        return self.__privado

    def setPrivado(self, privado):
        self.__privado = privado

    def __metodoPrivado(self):
        pass
```

Orientação o Objetos

- Propriedades

Uma maneira mais elegante para métodos get e set (*propriedades01.py*)

```
class ClasseA(object):
    def __init__(self):
        self.__privado = None

    def getPrivado(self):
        if self.__privado == None:
            return 10
        else:
            return self.__privado

    def setPrivado(self, arg):
        if arg >= 0:
            self.__privado = arg
        else:
            self.__privado = -1 * arg

    privado = property(getPrivado, setPrivado)
```

```
classeA = ClasseA()

print classeA.privado

classeA.privado = -2

print classeA.privado
```

Orientação o Objetos

- Atributos e métodos estáticos

atributos-estaticos.py

```
class ClasseA(object):

    contador = 0

    def __init__(self):
        ClasseA.contador += 1
        print 'ClasseA instaciada pela', ClasseA.contador, 'vez\n'

    def qtdInstancias():
        print 'A classe ClasseA possui', ClasseA.contador, 'instancia(s)'

    instancias = staticmethod(qtdInstancias)

a = ClasseA()
b = ClasseA()
c = ClasseA()
ClasseA.instancias()
```

Orientação a Objetos

- Métodos especiais

```
class ClasseA(object):

    def __init__(self, arg):
        self.arg = arg

    def __eq__(self, outroObj):
        '''__eq__ é um método especial para igualdade entre objetos'''
        if isinstance(outroObj, ClasseA):
            return outroObj.arg == self.arg
        return False

a = ClasseA(2)
b = ClasseA(2)
print a == b #True
c = ClasseA(3)
print a == c #False
```

Orientação a Objetos

- Métodos especiais(2)

__add__(...)
    x.__add__(y) <==> x+y

__contains__(...)
    x.__contains__(y) <==> y in x

__eq__(...)
    x.__eq__(y) <==> x==y

__ge__(...)
    x.__ge__(y) <==> x>=y

__getattribute__(...)
    x.__getattribute__('name') <==> x.name

__getitem__(...)
    x.__getitem__(y) <==> x[y]

...

Exceções

- Capturando exceções
  - Exceções podem ser capturadas em qualquer parte do código

excecoes01.py

```
while True:
    try:
        x = int(raw_input("Insira um numero: "))
    except ValueError, ve:
        print "Oops!  Numero invalida.  Tente novamente..."

    except Exception, e:
        print "Outra excecao foi gerada"
        raise e

    else: #nenhuma excecao foi gerada
        break
```

Exceções

- Capturando exceções(2) – finally

Códigos dentro da declaração *finally* sempre são executados

excecoes02.py

```
while True:
    try:
        try:
            x = int(raw_input("Insira um numero: "))

        except ValueError, ve:
            raise ve

    finally:
        print "Finally sempre eh executado"
```

Exceções

- Criando novas exceções

Tipicamente as exceções herdam de da classe Exception, porém não é obrigatório

```
>>> class MyError(Exception):
...         def __init__(self, value):
...                 self.value = value
...          def __str__(self):
...                 return repr(self.value)
...
>>> try:
...         raise MyError(2*2)
...     except MyError, e:
...         print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
```