Curso de Python em 5 Horas

Syntaxe Básica: controle de fluxo, laços e funções. Tipos Básicos

Gustavo Sverzut Barbieri

GPSL - UNICAMP

12 de maio de 2005

Resumo

Esta aula é a primeira parte técnica, ela aborda a sintaxe básica, com controle de fluxo, laços e funções. Esta aula também aborda os tipos básicos, como lista, tuplas, dicionários e strings.

O material de apoio a ser utilizado se encontra em:

- Python para já programadores: `http://www.gustavobarbieri.com.br/python/aulas_python/aula-01.pdf`
- Resumo: `http://www.gustavobarbieri.com.br/python/aulas_python/resumo.pdf`

Panorama

Panorama

Linhas de Código

Um programa em Python é constituído de linhas lógicas:

- Linhas Lógicas são terminadas com uma nova linha (“\n”)
- Exceto quando explicitamente continuadas na próxima linha física, para isto usa-se o “\”.
- Ou implicitamente continuadas na próxima linha física por expressões que usam parênteses, colchetes ou chaves

```
a = b + c
x = x * f + \
    x0 * f
l = [ 1, 2,
      3, 4 ]
d = { 'a': 1,
      'b': 2 }
if a > x and \
   b < y:
   print "text"
```

Comentários e Comentários Funcionais

- Após o caractere "*#*" até o final da linha, tudo é considerado um comentário e ignorado, exceto pelos comentários funcionais.
- Defina a codificação do arquivo fonte com o comentário funcional *#−∗− coding: <encoding−name> −∗−* (útil se usar Emacs) ou *#vim: fileencoding =<encoding−name>* (útil se usar Vim) ou qualquer comentário que case com o padrão "`coding[=:]\s*([-\w.]+)`".
- Em sistemas Posix pode-se usar o comentário funcional *#!/usr/bin/env python* para executar o arquivo com o comando python encontrado no ambiente. *Isto não é característico do Python, mas dos sistemas Posix*.

Indentação

- Em Python, os blocos de código são delimitados pelo uso de indentação.
- A indentação não precisa ser consistente em todo o arquivo, só no bloco de código, porém é uma boa prática ser consistente no projeto todo.
- Cuidado ao misturar TAB e Espaços: configure seu editor!
- Utilitário `tabnanny`[1] verifica indentação.

```
a = 1
b = 2
if a < b:
    print "a é menor"
else:
        print "b é menor" # Apesar de Ok, evite inconsistências!
```

[1] geralmente está na instalação do seu python, como /usr/lib/python2.3/tabnanny.py

Panorama

Identificadores (Nomes de Variáveis, Métodos, Funções, ...)

- Devem começar com uma letra (a-z, sem acentuação) ou sublinhado ("_")
- Depois pode ser seguido por uma letra (a-z, sem acentuação), dígitos e sublinhado ("_")
- Alguns identificadores são palavras-chave reservadas: `and`, `del`, `for`, `is`, `raise`, `assert`, `elif`, `from`, `lambda`, `return`, `break`, `else`, `global`, `not`, `try`, `class`, `except`, `if`, `or`, `while`, `continue`, `exec`, `import`, `pass`, `yield`, `def`, `finally`, `in`, `print`
- **Maiúsculas e Minúsculas são diferentes!**

Panorama

Literais

- **Strings**
  - Convencional: `'texto'` ou `"texto"`
  - Multi-Line: `'''texto várias linhas'''` ou `"""texto várias linhas"""`
  - Unicode: `u'texto unicode'` ou `u"texto"`, ...
  - Raw: `r'texto bruto\n'`
  - Strings em várias linhas são concatenadas.
- **Números** parecido com outras linguagens, ié C, C++, Java:
  - Inteiro: `123` (decimal), `0632` (octal), `0xff00` (hexadecimal)
  - Longo: `123L` ou `123l`
  - Ponto Flutuante: `3.14`, `10.`, `.12`, `1.23e-9`
  - Complexos: `10.0 + 3j`

Panorama

Operadores

Operador	Descrição	Método correspondente
+	adição	`__add__`
-	subtração	`__sub__`
*	multiplicação	`__mul__`
**	potenciação	`__pow__`
/	divisão	`__div__`
//	divisão por baixo (floor)	`__floordiv__`
%	módulo (resto)	`__mod__`
<<	deslocamento à esquerda	`__lshift__`
>>	deslocamento à direita	`__rshift__`
&	"e" lógico (and) bit-a-bit	`__and__`
|	"ou" lógico (or) bit-a-bit	`__or__`
^	"ou exclusivo" (xor) bit-a-bit	`__xor__`
~	Inverte	`__inv__`
<	menor	`__lt__`
>	maior	`__gt__`
<=	menor ou igual	`__le__`
>=	maior ou igual	`__ge__`
==	igual	`__eq__`
!=	diferente	`__ne__`

Para maiores informações `import operator; help( operator )`.

Panorama

Panorama

Variáveis e suas características no Python

- Python usa tipagem dinâmica: uma variável não tem tipo fixo, ela tem o tipo do objeto que ela contém.
- Para criar um novo conteúdo para a variável é necessário apenas uma atribuição
- Um conteúdo é destruído e recolhido pelo coletor de lixo quando nenhuma variável ou estrutura aponta mais para ele.

```
a = "texto" # 'a' contém uma string, então é do tipo 'string' (str)
a = 123     # 'a' contém um inteiro, então é do tipo 'inteiro' (int)

a = [ 1, 2, 3 ]
b = [ a, "123", 333 ]

d = { "chave": "valor", "teste": a, "b": 12345 }
```

Parada para Exercícios: Variáveis

Utilize o Python no modo interativo como calculadora e calcule:

1. 2^{50}
2. $2^{50} * 3$
3. $2^{50} * 3 - 1000$
4. $\frac{400.0}{2^{50}} + 50$

Respostas:

1. 2^{50}: `a = 2 ** 50, a == 1125899906842624L`
2. $2^{50} \times 30$: `b = a * 30, b == 33776997205278720L`
3. $2^{50} \times 30 - 1000$: `c = b - 1000, c == 33776997205278720L`
4. $\frac{400.0}{2^{50}} + 50 =$ `d = 400.0 / a + 50, d == 50.000000000000355`

Panorama

Controle de Fluxo

```
if CONDICAO_1:
   BLOCO_DE_CODIGO_1
elif CONDICAO_2:
   BLOCO_DE_CODIGO_2
else:
   BLOCO_DE_CODIGO_3
```

Parênteses só são necessários para evitar ambiguidades!

```
idade = int( raw_input( "Idade:" ) )
if  idade < 2:
    print "Bebe"
elif 2 <= idade <= 13:
    print "Criança"
elif 14 <= idade <= 19:
    print "Adolescente"
else:
    print "Adulto"
```

```
if a and ( b or c ):
   print "verdade"
```

O que avalia para verdadeiro e falso

Os seguintes valores são considerados falsos:

- `None`
- `False`
- Valor 0 de vários tipos: 0, 0.0, 0L, 0j
- Seqüências vazias: '', (), []
- Mapeamentos vazios:
- Instâncias de objetos que definam `__nonzero__()` que retorne valor `False` ou 0
- Instância de objetos que definem o `__len__()` o qual retorne 0.

Parada para Exercícios: Controle de Fluxo

Implemente o seguinte conjunto de regras em Python:

- Se `a` for verdadeiro e `b` for falso, imprima "Caso 1"
- Senão, Caso `a` for falso e `b` for verdadeiro, imprima "Caso 2"
- Caso contrário:
  - Caso `c` for maior que 10 e `d` estiver entre 0.0 e 100.0, imprima "Caso 3"
  - Caso `e` estiver na lista `lst`, imprima "Caso 4"
  - Senão imprima "Caso 5"

Parada para Exercícios: Controle de Fluxo (Resposta)

```
if a and not b:
    print "Caso 1"
elif not a and b:
    print "Caso 2"
else:
    if c > 10 and 0.0 <= d <= 100.0:
        print "Caso 3"

    if e in lst:
        print "Caso 4"
    else:
        print "Caso 5"
```

Panorama

Laços

```
while CONDICAO:
    BLOCO_DE_CODIGO
```

```
for VARIAVEL in SEQUENCIA:
    BLOCO_DE_CODIGO
```

```
from time import time
start = time()
while time() - start < 3.0:
    print "esperando..."
```

```
for fruta in [ "Banana", "Maça", "Uva" ]:
    print "Fruta:", fruta
```

```
d = { "a": 1, "b": 2 }
for chave, valor in d.iteritems():
    print "Chave:", chave, ", Valor:", valor
```

```
for i in xrange( 100, 200, 10 ):
    print "i:", i
```

Laços: Próxima iteração e Saída Forçada

- `continue` interrompe a execução da iteração atual e vai para a próxima, se esta existir.
- `break` interrompe a execução do laço.

```
texto = raw_input( "Entre com o texto:" )
for letra in texto:
    if letra == 'c':
        continue
    elif letra == 'b':
        break
    print "letra:", letra
```

Laços: usando o “else”

Visando facilitar a vida do programador, Python fornece a cláusula `else` para os laços. Esta será executada quando a condição do laço for falsa, eliminando a necessidade do programador manter uma variável de estado.

```
for elemento in lista:
    if elemento == parada:
        break
    print elemento
else:
    print "Laço chegou ao fim"
```

No exemplo acima, a mensagem “Laço chegou ao fim” só é imprimida caso não existir um elemento que seja igual a “parada”.

Parada para Exercícios: Laços

Dada uma lista de palavras "`lista`" e uma palavra "`chave`" imprima o índice do elemento que encontrou a palavra, senão imprima "Palavra Não Encontrada".

Parada para Exercícios: Laços (Resposta)

```
for indice, palavra in enumerate( lista ):
    if palavra == chave:
        print indice
        break
else:
    print "Palavra Não Encontrada"
```

Explicação

- `enumerate(sequencia)` é um iterador que retorna pares (indice, sequencia[indice])
- Em python, a construção a seguir é válida:

```
x, y = 1, 2
print x # 1
print y # 2
par = 1, 2 # idem a par = (1, 2)
print par # (1, 2)
x, y = par # idem a x, y = 1, 2
```

- Então `for indice, palavra in enumerate( lista ):` é válido!

Panorama

Funções

```
def NOME_DA_FUNCAO( LISTA_DE_PARAMETROS ):
    BLOCO_DE_CODIGO
```

```
def fatorial( numero ):
    if numero <= 1:
        return 1
    else:
        return ( numero * fatorial( numero - 1 ) )
```

Funções: Usando parâmetros com valor padrão

Pode-se ter parâmetros com valores padrão, estes devem vir depois dos parâmetros sem valor padrão.

```
def f( a, b, c=3 ):
    print "a:", a, "b:", b, "c:", c

f( 1, 2 )     # imprime: "a: 1 , b: 2 , c: 3"
f( 1, 2, 0 ) # imprime: "a: 1 , b: 2 , c: 0"
```

Cuidado!

O valor do padrão para um parâmetro é calculado somente uma vez quando o programa é carregado, caso você use um objeto mutável, todas as chamadas usarão a mesma instância.
Vide exemplos no próximo slide!

Funções: Usando parâmetros com valor padrão (2)

```
def f( v, l=[] ): # l é instanciado quando o python lê esta linha
    l.append( v )
    return l

print f( 1 ) # imprime [1]
print f( 2 ) # imprime [1, 2]
```

Talvez seja este o comportamento que você quer, mas talvez não. Caso deseje que uma nova instância seja criada para cada chamada, utilize algo como:

```
def f( v, l=None ):
    if l is None:
        l = []
    l.append( v )
    return l
```

```
def f( v, l=None ):
    l = l or []
    l.append( v )
    return l
```

Funções: Número variável de argumentos

- **Argumentos sem nome:** os argumentos são passados para a função na forma de uma lista, na ordem em que foram digitados:

```
def arg_sem_nome( *args ):
    for arg in args:
        print "arg:", arg

arg_sem_nome( 'a', 'b', 123 )
```

- **Argumentos com nome:** os argumentos são passaos para a função na forma de um dicionário, o nome do argumento é a chave.

```
def arg_com_nome( **kargs ):
    for nome, valor in kargs.iteritems():
        print nome, "=", valor

arg_com_nome( a=1, b=2, teste=123 )
```

- **Usando Ambos:**

```
def f( a, b, *args, **kargs ):
    print "a:", a, ", b:", b, \
          "args:", args, ", kargs:", kargs

f(1, 2); f(1, 2, 3); f(1, 2, t=9); f(1, 2, 3, t=9)
```

Parada para Exercícios: Funções

Faça uma função que dado um número, retorne o próximo na seqüência de Robert Morris (`http://www.ocf.berkeley.edu/~stoll/number_sequence.html`): 1, 11, 21, 1211, 111221, ...

Parada para Exercícios: Funções (Resposta)

```
def next_morris( number ):
    number = str( number )
    r = []
    i = 0
    last = number[ 0 ]
    for c in number:
        if c == last:
            i += 1
        else:
            r.append( str(i) + last )
            last = c
            i = 1

    r.append( str(i) + last )
    return "".join( r )
```

Peculiaridades dos blocos de código

- Um bloco vazio é criado com o *keyword* `pass`

```
while True:
    pass # Bloco vazio
```

- Qualquer string "solta" (não atribuída a variáveis) é considerada uma "docstring" e contribui para a documentação do bloco, no atributo `__doc__`:

```
def f():
    '''Documentação da função.

    Esta função faz bla bla bla...
    '''
    corpo()
    return valor
```

Visualize com `help(f)` ou com o pydoc (vide próximo slide).

Panorama

Visualizando a documentação com PyDOC

- **No terminal:** utilize o comando `pydoc`[2]:

  `pydoc ./arquivo.py`

  ou

  `/usr/lib/python2.3/pydoc.py ./arquivo.py`

  para ver a documentação de `arquivo.py` no diretório atual.
- **No Navegador:** utilize o comando:

  `pydoc -p 8000`

  e acesse o endereço `http://localhost:8000/`
- **Utilitário Gráfico:** execute:

  `pydoc -g`

  para um utilitário gráfico que faz buscas e mostra a documentação utilizando um navegador.

[2] caso não existir, procure por `pydoc` ou `pydoc.py` na instalação do seu Python, por exemplo `/usr/lib/python2.3/pydoc.py`

Visualizando a documentação com PyDOC (2)

Atenção ao usar o PyDOC:

- O PyDOC utiliza os arquivos como módulos para gerar a documentação
- Implica em executar os códigos definidos no arquivo, um problema no caso de ler do teclado, usar rede, processamentos pesados, ...
- **Solução:** coloque o código a ser executado isolado pela condição `__name__ == '__main__'`:

```
def f1():
    print 'f1()'

def f2( a ):
    '''Documentacao da funcao...
    '''
    print 'f2(', a, ')'

if __name__ == '__main__':
    f1()
    f2( 'teste' )
```

Panorama

Panorama

Lista: Operações Interessantes

- Criação:

```
lista = [ 10, 2, 3, 'texto', 20 ]
```

- Acesso a elementos pelo índice:

```
print lista[ 2 ] # imprime '3'
```

- Mudar elementos já existentes[3]:

```
lista[ 0 ] = 123
```

- Acesso a pedaços da lista:

```
sub_lista = lista[ 2: 4 ] # pega de 2 a 4 (não incluso)
print sub_lista # imprime '[ 3, 'texto' ]'
sl1 = lista[ : 3 ] # pega elementos até posicão 3
sl2 = lista[ 3 : ] # pega elementos da posição 3 até o fim
```

- Acrescentando ao fim:

```
lista.append( 1 ) # Acrescenta 1 ao fim da lista
```

[3] Não pode criar uma nova posição, ela já deve existir!

Lista: Operações Interessantes (2)

- Estendendo a lista com outra lista:

```
lista.extend( [ 10, 20, 30 ] )
lista += [ 40, 50, 60 ]
```

- Ordenando a lista (altera a própria lista!):

```
lista.sort()
lista.sort( lambda x, y: cmp( y, x ) ) # reverso!
```

- Inverter lista (altera a própria lista!):

```
lista.reverse()
```

- Contar ocorrências de um elemento:

```
lista.count( 3 ) # retorna 1
```

- Retorna a posição do elemento na lista:

```
[ 1, 2, 3, 4 ].index( 3 ) # retorna 2
```

- Inserir um elemento na posição desejada:

```
lista.insert( 0, 'abc' ) # insere no começo da lista
```

Lista: Operações Interessantes (3)

- Apagando um elemento:

```
del lista[ 2 ] # apaga o 3o elemento
```

- Apagando um pedaço da lista:

```
del lista[ 3 : 5 ] # apaga da pos. 4 até 5 (não incluso)
```

- Mudando um pedaço da lista:

```
l = [ 1, 2, 3, 4, 5, 6 ]
l[ 2 : 4 ] = [ 10, 20 ]
print l # imprime '[1, 2, 10, 20, 5, 6]'
```

- Repetindo uma lista:

```
l = [ 1, 2 ] * 5 # l é [1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

- Tamanho da lista:

```
print len( range( 10 ) ) # imprime 10
```

- Para mais informações: `help( list )` e `http://www.python.org/doc/2.3/lib/typesseq-mutable.html`

Lista: Exemplos

```
>>> lst = [ 1, 2, 3, 4.0, "abc" ]
>>> print lst[ 0 ]
1
>>> print lst[ 0 : 2 ]
[1, 2]
>>> print lst[ : 2 ]
[1, 2]
>>> print lst[ -1 ]
abc
>>> print lst[ -2 : ]
[4.0, 'abc']
>>> del lst[ 1 ]
>>> print lst
[1, 3, 4.0, 'abc']
>>> print 3 in lst
True
>>> lst.append( 5 )
>>> print lst
[1, 3, 4.0, 'abc', 5]
>>> lst += [ 6, 7 ]
>>> lst.extend( [ 8, 9 ] )
>>> print lst
[1, 3, 4.0, 'abc', 5, 6, 7, 8, 9]
```

```
>>> lst.reverse()
>>> print lst
[9, 8, 7, 6, 5, 'abc', 4.0, 3, 1]
>>> lst.sort()
>>> print lst
[1, 3, 4.0, 5, 6, 7, 8, 9, 'abc']
>>> print [ 10 ] * 5
[10, 10, 10, 10, 10]
>>> print lst.pop()
abc
>>> print lst
[1, 3, 4.0, 5, 6, 7, 8, 9]
>>> len( lst )
8
```

Lista: Exercícios

1. Dado uma lista "`lista`", verifique se "`valor`" está dentro dela, caso verdade imprima "Sim", senão imprima "Não".
2. Dado uma lista "`lista`", itere sobre a lista, imprimindo cada um de seus elementos.
3. Dado uma lista "`lista`", crie uma nova lista "`rotaciona_3`" que cada posição está rotacionada em 3 posições, isto é, $indice' = indice + 3$. Existe o método "tosco" e o método fácil, ou Pythônico, de se fazer isto! ;-)

Lista: Exercícios (Respostas)

1.
```
# -*- coding: utf-8 -*-
valor = int( raw_input( 'Entre com o valor: ' ) )
lista = [ 1, 10, 2, 50 ]
if valor in lista:
   print 'Sim'
else:
   print 'Não'
```

2.
```
lista = [ 10, 1.0, 2.0, 'teste', ( 1, 2 ) ]
for elemento in lista:
   print 'Elemento:', elemento
```

3. Provavelmente você não usou esta solução, tente enteder o que ela faz e porque Python ajuda a sua vida!

```
lista = range( 10 )
rotaciona_3 = lista[ 3 : ] + lista[ : 3 ]
```

Lista: Colocando Pimenta!!!

- List Comprehensions:

```
>>> l1 = [ str( i ) for i in xrange( 10 ) ]
>>> print l1
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> l2 = [ i ** 2 for i in xrange( 5 ) if i % 2 == 0 ]
>>> print l2
[0, 4, 16]
```

Para aprender mais, vide
http://www.python.org/peps/pep-0202.html e
http://docs.python.org/tut/.

- Zip, Sum e Outros:

```
>>> l1 = range( 3 )
>>> l2 = range( 0, 30, 10 )
>>> print zip( l1, l2 )
[(0, 0), (1, 10), (2, 20)]
>>> print sum( l1 ), sum( l2 )
3 30
```

Vide também `map()`, `reduce()` e `filter()`.

Tuplas

- Parecido com as listas, porém é **imutável**: não se pode acrescentar, apagar ou modificar valores.
- Vantegem: eficiente!
- Para mais informações: `help( tuple )`
- Ou a documentação em: `http://www.python.org/doc/2.3/lib/typesseq.html`

```
>>> tupla = ( 1, 2, 'abc' )
>>> tupla[ 0 ]
1
>>> tupla[ : 2 ]
(1, 2)
>>> tupla[ 2 : ]
('abc',)
>>> len( tupla )
3
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

Panorama

Dicionário (*Hash Tables*): Operações Interessantes

- Criação:

```
d = { 'chave': 'valor', 'nome': 'Gustavo Barbieri' }
```

- Acesso a elementos:

```
print d[ 'chave' ] # imprime 'valor'
```

- Adicionando elementos (a chave precisa ter *hash* fixo, em geral os **imutáveis**):

```
d[ 'teste' ] = 'bla bla'
d[ 1 ] = 10
d[ ( 1, 2 ) ] = 'algum valor'
lista = [ 1, 2 ]
d[ lista ] = 'outro valor' # TypeError: list objects are unhashab
```

- Apagar elemento do dicionário:

```
del d[ 'teste' ]
```

Dicionário (*Hash Tables*): Operações Interessantes (2)

- Obtendo os ítens, chaves e valores:

```
itens   = d.items()  # Com 'M'! lista de tuplas (chave,valor)
chaves  = d.keys()   # lista de chaves
valores = d.values() # lista de valores
```

- Obtendo iteradores (otimizado para `for`):

```
for chave in d.iterkeys():
    print chave

for valor in d.itervalues():
    print valor

for chave, valor in d.iteritems():
    print chave, '=', valor
```

- Para mais informações: `help( dict )` e `http://www.python.org/doc/2.3/lib/typesmapping.html`

Dicionário (*Hash Tables*): Exemplos

```
>>> d = { 'a': 1, 'b': 2, 3: 'c' }
>>> d
{'a': 1, 3: 'c', 'b': 2}
>>> 'a' in d
True
>>> 'c' in d
False
>>> for k in d:
...     print k
a
3
b
>>> for k, v in d.iteritems():
...     print k, '=', v
a = 1
3 = c
b = 2
>>> d.get( 10, 'other value' )
'other value'
>>> d.setdefault( 10, 'some val' )
'some val'
>>> d
{'a': 1, 3: 'c', 10: 'some val', 'b': 2}
```

Dicionário (*Hash Tables*): Exercícios

1. Crie um dicionário `d` e coloque nele seus dados: nome, idade, telefone, endereço.
2. Usando o dicionário `d` criado anteriormente, imprima seu nome.
3. Também usando `d`, imprima todos os ítens do dicionário no formato `chave : valor`, ordenado pela chave.

Dicionário (*Hash Tables*): Exercícios (Respostas)

1.
```
# -*- coding: utf-8 -*-
d = { 'nome': 'Gustavo Sverzut Barbieri',
      'idade': 22,
      'telefone': '3242-9695',
      'endereco': 'Rua Votorantim' }
```

2.
```
print d[ 'nome']
```

3.
```
chaves = d.keys()
chaves.sort()
for c in chaves:
    print c, ':', d[ c ]
```

Panorama

String: Operações Interessantes

Comporta-se praticamente como uma tupla, mas é específica para texto.

Atenção

Strings são imutáveis, não é possível adicionar texto ou mesmo modificar algum caractere!

- Criação:

```
texto1 = 'abcdefghij'
texto2 = "outro texto"
texto3 = '''este texto
tem varias
linhas'''
texto4 = r'Digite \t para tabulação'
```

- Acesso a elementos pelo índice:

```
print texto1[ 2 ] # Imprime 'c'
print 'GSB'[ 1 ]  # Imprime 'S'
```

String: Operações Interessantes (2)

- Acesso a pedaços da string:

```
print texto1[   : 2 ]  # imprime 'ab'
print texto1[ 2 :   ]  # imprime 'cdefghij'
print texto1[ 2 : 4 ]  # imprime 'cd'
```

- Funções de procura por sub-strings:
  - No começo:

```
texto1.startswith( 'abc' ) # Verdadeiro
texto1.startswith( 'cba' ) # Falso
```

  - No fim:

```
texto1.endswith( 'ij' ) # Verdadeiro
texto1.endswith( 'xy' ) # Falso
```

  - Em qualquer posição:

```
texto1.find( 'cde' ) # retorna 2
texto1.find( 'XXX' ) # retorna -1, pois não existe
```

String: Operações Interessantes (3)

- Funções de verificação de conteúdo:
  - Se são somente letras:

```
'abc'.isalpha() # Verdadeiro
'123'.isalpha() # Falso
'a12'.isalpha() # Falso
```

  - Se são somente números:

```
'abc'.isdigit() # Falso
'123'.isdigit() # Verdadeiro
'a12'.isdigit() # Falso
```

  - Se são letras e números:

```
'abc'.isalnum() # Verdadeiro
'123'.isalnum() # Verdadeiro
'a12'.isalnum() # Verdadeiro
```

  - Se são somente espaços:

```
'   '.isspace()   # Verdadeiro
' \t\n'.isspace() # Veradeiro
'a    '.isspace() # Falso
```

String: Operações Interessantes (4)

- Transformar seqüências em texto e texto em seqüências:
  - Juntando textos de uma lista em um texto só:

```
print '-'.join( [ 'a', 'b', 'cde' ] ) # Imprime 'a-b-cde'
print 'AB'.join( [ '1', 'teste' ] )   # Imprime '1ABteste'
```

  - Quebrando um texto em uma lista:

```
print 'nome:senha'.split( ':' ) # Imprime ['nome', 'senha']
print 'a/|b/|c'.split( '/|' ) # Imprime ['a', 'b', 'c']
```

- Transformar a caixa do texto (cria-se novas instâncias!):
  - Para maiúscula:

```
print 'abC'.upper() # Imprime: 'ABC'
```

  - Para minúscula:

```
print 'aBC'.lower() # Imprime: 'abc'
```

  - Inverter maiúsculas e minúsculas:

```
print 'aBc'.swapcase() # Imprime: 'AbC'
```

String: Operações Interessantes (4)

- Funções de verificação de conteúdo (continuação):
  - Se está em maiúscula, minúscula, ...:

```
'abc'.islower() # Veradeiro
'AbC'.islower() # Falso
'ABC'.isupper() # Veradeiro
'Abc Teste'.istitle() # Verdadeiro
'Abc teste'.istitle() # Falso
```

- Retirar caracteres (cria-se novas instâncias!):
  - Da esquerda:

```
print '  abc  '.lstrip() # Imprime: 'abc  '
print '_*_abc_*_'.lstrip( '*_' ) # Imprime: 'abc_*_'
```

  - Da direita:

```
print '  abc  '.rstrip() # Imprime: '  abc'
print '_*_abc_*_'.rstrip( '*_' ) # Imprime: '_*_abc'
```

  - De ambos os lados:

```
print '  abc  '.strip() # Imprime: 'abc'
print '_*_abc_*_'.strip( '*_' ) # Imprime: 'abc'
```

String: Operações Interessantes (5)

- Troca de caracteres (cria-se novas instâncias!):
  - Troca de pedaços:

```
print 'este eh um teste'.replace( 'ste', 'ABC' )
# Imprime: 'eABC eh um teABC'
```

  - Caracteres individuais, baseados em uma tabela:

```
import string
tabela = string.maketrans( 'ael', '431' )
txt = 'lammers like to write like this'
print txt.translate( tabela )
# Imprime: '14mm3rs 1ik3 to writ3 1ik3 this'
```

- Vide também documentação do módulo string:
  `http://www.python.org/doc/lib/module-string.html`

String: Interpolação

- Operador **%** para fazer interpolação de string (o mesmo que `sprintf()` do C).
- O formato deve seguir a convenção do `printf()` do C.
- Pode-se usar valores nomeados, passando um dicionário.
- Operadores extra, como o "%r" para a representação do objeto, mesmo a usar "%s" com valor `repr(obj)`.
- Mais informações em `http://docs.python.org/lib/typesseq-strings.html`

```
>>> formato = "nome: %-30s, idade: %2d, peso: %3.1fkg"
>>> formato % ( "Gustavo Barbieri", 22, 73.46 )
'nome: Gustavo Barbieri                , idade: 22, peso: 73.5kg'
>>> formato2 = "nome: %(nome)-30s, idade: %(idade)2d, " \
...             "peso: %(peso)3.1fkg"
>>> formato2 % { "nome": "Gustavo Barbieri", "idade": 22,
...               "peso": 73.46 }
'nome: Gustavo Barbieri                , idade: 22, peso: 73.5kg'
```

String: Exemplos

```
>>> a = "texto"
>>> "a" in a
False
>>> "o" in a
True
>>> a[ 0 ]
't'
>>> a[ : 2 ]
'te'
>>> a[ 2 : ]
'xto'
>>> a.center( 20 )
'       texto        '
>>> a.capitalize()
'Texto'
>>> a.upper()
'TEXTO'
>>> a.startswith( "te" )
True
>>> a.endswith( "bla" )
False
```

```
>>> a.islower()
True
>>> a.isspace()
False
>>> a.isalpha()
True
>>> a.isdigit()
False
>>> a.replace( 'xto', 'ste' )
'teste'
>>> b = 'a:b:c'
>>> b.split( ':' )
['a', 'b', 'c']
>>> '*'.join( [ 'A', 'B', 'C' ] )
'A*B*C'
>>> 'nome: %s, idade: %2d' % \
... ( 'Gustavo', 22 )
'nome: Gustavo, idade: 22'
```

String: Exercícios

1. Converta uma string para maiúscula e imprima.
2. Dado o texto `'abacate'` troque as letras `'a'` por `'4'` e imprima.
3. Dado o texto `'bin:x:1:1:bin:/bin:/bin/false'`, quebre-o na ocorrência de `':'`.
4. Dado uma tupla `('a', 'b', 'c')`, transforme-a em uma string, separada por `'*'`.
5. Uma mensagem está "criptografada" usando o rot13[4]: *"fr ibpr rfgn yraqb rfgr grkgb, cnenoraf. pnfb anb graun hgvyvmnqb b genafyngr(), gragr qrabib!"*. Decodifique-a!

[4] ROT13: `http://en.wikipedia.org/wiki/ROT13`, considere somente letras minúsculas.

String: Exercícios (Respostas)

1. ```
   print 'texto'.upper()
   ```

2. ```
   print 'abacate'.replace( 'a', '4' )
   ```

3. ```
   valores = 'bin:x:1:1:bin:/bin:/bin/false'.split( ':' )
   ```

4. ```
   print '*'.join( ( 'a', 'b', 'c' ) )
   ```

String: Exercícios (Respostas, 2)

5.

```
import string

def rot13( texto ):
    p   = string.lowercase
    rot = p[ 13 : ] + p[ : 13 ]
    tabela = string.maketrans( p, rot )
    return texto.translate( tabela )
# rot13()

print rot13('fr ibpr rfgn yraqb rfgr grkgb, cnenoraf. pnfb ' \
            'anb graun hgvyvmnqb b genafyngr(), gragr qrabib!')
```

Panorama

Panorama

Referências utilizadas nesta aula

- Python Tutorial `http://docs.python.org/tut/tut.html`
- Python Library Reference `http://docs.python.org/lib/lib.html`
- Python Language Reference `http://docs.python.org/ref/ref.html`
- Python para já Programadores `http://www.gustavobarbieri.com.br/python/aulas_python/aula-01.pdf`

Panorama

Gustavo Sverzut Barbieri

Email:	barbieri@gmail.com
Website:	`http://www.gustavobarbieri.com.br`
ICQ:	17249123
MSN:	barbieri@gmail.com
Jabber:	gsbarbieri@jabber.org

Obtenha esta palestra em:
`http://palestras.gustavobarbieri.com.br/python-5hs/`