Python

Uma introdução Rápida

Python

Uma introdução Rápida

Conceitos Básicos

- Tudo em Python é um objeto
- Espaços em brancos são significativos em Python
- Python é sensível ao case.

Comentários

Comentários em Python são delimitados pelo caracter '#', e se extendem até o final da linha.

```
# primeiro comentário
SPAM = 1 # e esse é o segundo comentário
                # ... e ainda um terceiro !
STRING = "# Este não é um comentário."
```

Usando Python como calculadora

```
>>> 2+2
4
>>> # Isso é um comentário
... 2+2
4
>>> 2+2  # e um comentário na mesma linha de um comando
4
>>> (50-5*6)/4
5
>>> # Divisão inteira retorna com aredondamento para base
... 7/3
2
>>> 7/-3
-3
```

Atribuição

O sinal de igual (‘=’) é utilizado para atribuição de um valor a uma variável. Nenhum resultado é exibido até o próximo prompt interativo:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Atribuição simultânea

Um valor pode ser atribuído a diversas variáveis simultaneamente:

```
>>> x = y = z = 0  # Zero x, y e z
>>> x
0
>>> y
0
>>> z
0
```

Atribuição manipula referências

x = y #Não faz uma cópia de y
x = y #faz x referenciar o mesmo objeto que y referencia

Exemplo:

```
>>>a=[1,2,3]
>>>b=a
>>>a.append(4)
>>>print b
[1,2,3,4]
```

Variáveis

Não podemos mudar o valor de um tipo, mas podemos mudar o que ele referencia

```
>>>x=3
>>>x=x+1
>>>print x
4
```

Ponto flutuante

Há total suporte para ponto-flutuante; operadores com operandos de diferentes tipos convertem o inteiro para ponto-flutuante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Números Complexos

Números complexos também são suportados; números imaginários são escritos com o sufixo ‘j’ ou ‘J’. Números complexos com parte real não nula são escritos como ‘(real+imagj)’, ou podem ser criados pela chamada de função ‘complex(real, imag)’.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complexos

Números complexos são sempre representados por dois números ponto-flutuante, a parte real e a parte imaginária.
Para extrair as partes de um número z, utilize z.real e z.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Strings

Podem ser delimitadas pos aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Strings em várias linhas

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Terminadores de linha podem ser embutidos na string com barras invertidas, ex.:

```
oi = "Esta eh uma string longa contendo\n\
diversas linhas de texto assim como voce faria em C.\n\
    Observe que os espaços em branco no inicio da linha são \
 significativos."
print oi
```

```
Esta eh uma string longa contendo
diversas linhas de texto assim como voce faria em C.
    Observe que os espaços em branco no inicio da linha são significativos.
```

Strings raw

No entanto, se a tornarmos uma string “crua” (raw), as sequências de \n não são convertidas para quebras de linha.
Tanto a barra invertida quanto a quebra de linha no código-fonte são incluídos na string como dados.

```
oi = r"Esta eh uma string longa contendo\n\
diversas linhas de texto assim como voce faria em C.\n\
    Observe que os espaços em branco no inicio da linha são \
 significativos."

print oi
```

```
Esta eh uma string longa contendo\n\
diversas linhas de texto assim como voce faria em C.\n\
    Observe que os espaços em branco no inicio da linha são \
 significativos
```

Strings Verbatim

strings podem ser delimitadas por pares de aspas tríplices: " ou '''. Neste caso não é necessário embutir terminadores de linha, pois o texto da string será tratado verbatim.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

Produz

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Concatenar Strings

Strings podem ser concatenadas (coladas) com o operador +, e repetidas com *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Justaposição de strings literais

Duas strings literais justapostas são automaticamente concatenadas

```
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> str.strip('str') + 'ing'      #  <-  This is ok
'string'
>>> str.strip('str') 'ing'        #  <-  This is invalid
  File "<stdin>", line 1
    str.strip('str') 'ing'
                     ^
SyntaxError: invalid syntax
```

Indexação de strings

Strings podem ser indexadas; como em C, o primeiro índice da string é o 0. Não existe um tipo separado para caracteres; um caracter é simplesmente uma string unitária. Substrings podem ser especificadas através da notação slice (N.d.T: fatiar): dois índices separados por dois pontos.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Alteração de Strings

Diferentemente de C, strings não podem ser alteradas em Python.
Atribuir para uma posição (índice) dentro de
uma string resultará em erro:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Criação de Strings

Entretanto, criar uma nova string com o conteúdo combinado é fácil e eficiente:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Listas

Python possui diversas estruturas de dados nativas, utilizadas para agrupar outros valores. A mais versátil delas é a lista (list), que pode ser escrita como uma lista de valores separados por vírgula e entre colchetes. Mais importante, os valores contidos na lista não precisam ser do mesmo tipo.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Slice em listas

Da mesma forma que índices de string, índices de lista começam do 0, listas também podem ser concatenadas e sofrer o operador de slice.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]

>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Listas

Diferentemente de strings, que são imutáveis, é possível mudar elementos individuais da lista:

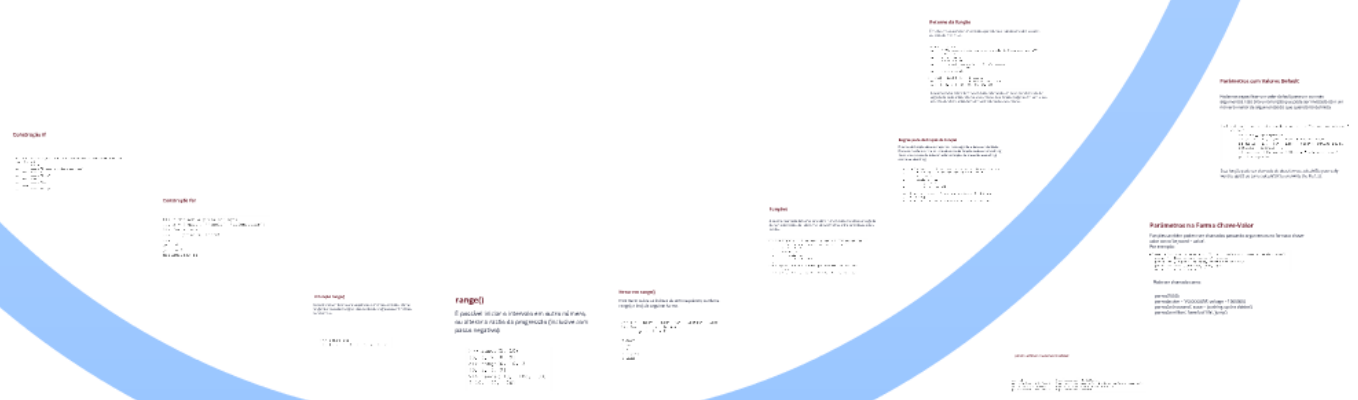
```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Aninhamento de Listas

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Programação

```
>>> # Serie de Fibonacci :
... # A soma de dois elementos define o proximo
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```



Construção if

```
>>> x = int(raw_input("Por favor entre com um numero inteiro: "))
>>> if x < 0:
...       x = 0
...       print 'Negativo mudou para zero'
... elif x == 0:
...       print 'Zero'
... elif x == 1:
...       print 'Um'
... else:
...       print 'Mais'
...
```

Construção for

```
>>> # Medindo algumas strings:
... a = ['gato', 'janela', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
janela 6
defenestrar 11
```

A Função range()

Se você precisar iterar sobre sequências numéricas, a função interna range() é a resposta. Ela gera listas contendo progressões aritiméticas, por exemplo:

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

range()

É possível iniciar o intervalo em outro número, ou alterar a razão da progressão (inclusive com passo negativo):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Iterar em range()

Para iterar sobre os índices de uma sequência, combine range() e len() da seguinte forma:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Funções

A palavra-reservada def serve para definir uma função. Ela deve ser seguida do nome da função, da lista formal de parâmetros entre parênteses e dois pontos.

```
>>> def fib(n):    # escreve  a serie de Fibonacci ate n
...     """Print a Fibonacci series up to n"""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Agora invoca a funçao que acabamos de definir
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Regras para definição de função

O corpo da função deve começar na linha seguinte e deve ser identado. Opcionalmente, a primeira linha do corpo da função pode ser uma string literal, cujo propósito é documentar a função. Se presente, essa string chama-se docstring.

```
>>> def fib(n):    # escreve  a serie de Fibonacci ate n
...     """Print a Fibonacci series up to n"""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Agora invoca a funçao que acabamos de definir
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Retorno da função

É muito simples escrever uma função que retorna a lista da série de Fibonacci, ao invés de imprimi-la:

```
>>> def fib2(n):
...     """Retorna a lista contendo a serie de Fibonacci ate n"""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # veja abaixo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)     # invoca
>>> f100                 # escreve resultado
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

A palavra-chave return termina a função retornando um valor. Se return não for seguido de nada, então retorna o valor None. Se a função chegar ao fim sem o uso explícito do return, então também será retornado o valor None.

Parâmetros com Valores Default

Podemos especificar um valor default para um ou mais argumentos. Isso cria uma função que pode ser invocada com um número menor de argumentos do que quando foi definida.

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Essa função pode ser chamada de duas formas: ask_ok('Do you really want to quit?') ou como ask_ok('OK to overwrite the file?', 2).

Parâmetros na Forma Chave-Valor

Funções também podem ser chamadas passando argumentos no formato chave-valor como 'keyword = value'.
Por exemplo:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

Pode ser chamado como:

parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')

porém, existem maneiras inválidas:

```
parrot()                     # parâmetro exigido faltando
parrot(voltage=5.0, 'dead')  # parâmetro não-chave-valor depois de parâmetro chave-valor
parrot(110, voltage=220)     # valor duplicado para mesmo parâmetro
parrot(actor='John Cleese')  # parâmetro desconhecido
```

Fim