Tutorial Python

Eliane Rocha dos Santos

Sumário

1 Introdução

Este tutorial foi elaborado visando esclarecer, aos leigos em Python, mas com conhecimento em linguagens de programação, conceitos básicos da linguagem, onde o leitor vai entender a sua sintaxe e suas particularidades.
O tutorial está voltado a explicações objetivas sobre os componentes Python, bem como a simples exemplos com base na versão Python 2.5.

2 Python

Talvez você associe o nome ao réptil Píton, então estará enganado, pois a linguagem foi nomeada com base no britânico grupo humorístico Monty Python, o qual criou o programa *Monty Python's Flying Circus*.
Python é uma linguagem de programação free, moderna, simples, dinâmica e orientada a objetos, além de ser voltada a um estilo de linguagem informal, portanto de fácil aprendizado.
Assim, você deve concluir que Python tem diversas características.
A linguagem Python é marca registrada do Stichting Mathematisch Centrum de Amsterdam.

2.1 O interpretador

Python é uma linguagem interpretada, o que significa que o código não precisa ser compilado para que seja executado. Assim, o interpretador lê e executa o código diretamente. Linguagens interpretadas, normalmente, funcionam através de 'Compilação Just-In-Time' ou 'Interpretação pura ou em Bytecode'.
Você pode criar seu arquivo Python e salvá-lo com a extensão ".py" ou ainda pode executar no modo shell, ou seja, você digita o código diretamente no interpretador.
Python é multi plataforma, roda em Windows, Linux, Unix, Macintosh, etc. Pode ser utilizado um ambiente para edição e execução como o IDLE.

2.2 Orientada a objetos

Programação orientada a objetos é um modo baseado na colaboração entre objetos. Assim, em Python, tudo é objeto, que possui atributos e métodos e pode ser atribuído a uma variável ou passado como argumento de uma função.
Por exemplo, uma variável que contem uma string é um objeto, pois qualquer string possui um método chamado upper que converte a string para maiúsculo.

```
>>> a = "Hello"
>>> a.upper()
'HELLO'
```

2.3 Tipagem dinâmica

Python é uma linguagem dinâmica, ou seja, suporta uma variável que pode ter diferentes tipos durante a execução do programa. Embora não seja explícita, ela assume um único tipo no momento em que atribui um valor a ela.

```
>>> var = 3
>>> type(var)
<type 'int'>
>>> var = "3"
>>> type(var)
<type 'str'>
>>> var = 3.0
>>> type(var)
<type 'float'>
```

Para descobrir o tipo da variável, basta usar a função *type()*. Com linguagem dinâmica, você garante a simplicidade e flexibilidade da função.

3 Blocos controlados por indentação

Em Python, ao contrário da maioria das linguagens, você não digita tags de abertura e fechamento, você deve, somente, indentar os blocos. Indicando, assim, onde eles iniciam e encerram. Desse modo, você, terá seu código organizado e legível.

```
>>> if var == 0:
      print "Zero"
elif var>0:
      print "Positivo"
else:
      print "Negarivo"
```

Python te obriga a ter clareza no código, através da indentação consistente.

4 Operadores

4.1 Operadores aritméticos

+ = adição
- = subtração
*** = multiplicação
/ = divisão
% = módulo
*** = exponenciação

Obs.: Os operadores *+* e * fazem concatenação de strings, listas e tuplas.

4.2 Operadores de atribuição

+= = incrementação
-+ = decrementação
/= = divide pelo valor atribuído
***=* = multiplica pelo valor atribuído

4.3 Operadores de condição

== = igualdade
!= = diferente
> = maior
< = menor
>= = maior ou igual
<= = menor ou igual
in = este operador é utilizado em seqüências ou dicionários para verificar a presença de um elemento.

```
>>> 5 in (2,3,5)
True
>>> "m" in "Jorge"
False
```

4.4 Operadores lógicos

and = e
or = ou
not = não

Obs.: Em Python, a sintaxe a seguir é válida:

```
>>> if 0<x<10:
      print x
```

5 Condição e laço

Assim como nas outras linguagens, Python possui estruturas de controle de fluxo (condição e laço) também, porém com algumas variações.

5.1 Instrução *If*

Em Python, não é permitido fazer atribuições em um *if*, então é necessário usar o operador "==" quando fizer comparações. A palavra *elif* é a abreviação de "else if".

```
>>> if x < 0:
     print "Negativo"
elif x == 0:
     print "Zero"
elif x <= 10:
     print "Entre 1 e 10"
elif x <= 20:
     print "Entre 11 e 20"
elif x <= 30:
     print "Entre 21 e 30"
else:
     print "Acima de 30"
```

5.2 Instrução *while*

A instrução *while* segue a lógica de qualquer outro *while* de qualquer linguagem e serve para testar um bloco de comandos à cada repetição do laço.

```
>>> i = 1
>>> while i < 10:
     print i
     i+=1


1
2
3
4
5
```

```
6
7
8
9
```

5.3 Instrução *for*

A instrução *for* interage com uma seqüência, ou seja, requer um objeto lista ou qualquer outro de seqüência.

```
>>> cores = ("rosa", "azul", "amarelo", "verde")
>>> x = 1
>>> for i in cores:
     print x, "\t", i
     x+=1
```

```
1     rosa
2     azul
3     amarelo
4     verde
```

Tanto na instrução *while* quanto na instrução *for* existem dois comandos: *continue* e *break*. O comando *continue* leva para a próxima iteração (repetição) do laço, interrompendo a iteração atual. O comando *break* interrompe o laço imediatamente e o programa prossegue a partir do próximo passo depois daquele laço.

```
>>> cores = ("rosa", "verde", "amarelo", "azul")
>>> for i in cores:
      if len(i) > 5:
                  break
      else:
             print i
```

```
rosa
verde
```

Nesse exemplo, imprime até o elemento que não tem tamanho maior que 5, quando é detectado um, não imprime ele nem os próximos elementos. Isso porque o *break* usado encerrou o laço ao detectar esse elemento.

6 Tipos de dados

6.1 Strings

Strings são seqüências de caracteres entre aspas que não podem ser modificadas, a menos que você reatribua.

```
>>> "Isto é uma string"
'Isto \xe9 uma string'
```

6.2 Listas

Listas são conjuntos de elementos de qualquer tipo separados por vírgula e que estão entre colchetes, e que podem ser modificados.

```
>>> lista = [1,"um",1.0]
>>> lista.insert(3,"hum")
>>> print lista
[1, 'um', 1.0, 'hum']
```

Uma variável chamada lista retorna uma lista com um valor inteiro, uma string e uma valor float. Em seguida, foi inserida no final da lista (o número '3' indica a posição três começando do zero) uma outra string.

Listas possuem alguns métodos:

append() - adiciona um elemento ao fim da lista.
count() - retorna o número de vezes que um determinado elemento aparece na lista.
extend() - estende a lista com novos elementos passados.
index() - retorna o índice (ou posição) de um determinado elemento da lista.
insert() - insere um determinado elemento numa especificada posição.
pop() - remove o elemento da posição indicada e o retorna.
remove() - remove o primeiro elemento da lista.
reverse() - inverte a ordem dos elementos da lista.
sort() - ordena os elementos da lista.

```
>>> l = [10,56,32,89,25,14]
```

```
>>> l.append(90)
>>> l
[10, 56, 32, 89, 25, 14, 90]
>>> l.insert(3,50)
>>> l
[10, 56, 32, 50, 89, 25, 14, 90]
>>> l.pop(7)
90
>>> l
[10, 56, 32, 50, 89, 25, 14]
>>> l.index(89)
4
>>> a = [15,90,68]
>>> a.extend(l)
>>> a
[15, 90, 68, 10, 56, 32, 50, 89, 25, 14, '5', '5']
>>> a.sort()
>>> a
[10, 14, 15, 25, 32, 50, 56, 68, 89, 90, '5', '5']
```

6.3 Tuplas

Tuplas são bem parecidas com listas, com duas diferenças: usa-se ou não () e **NÃO** podem ser modificadas.

```
>>> tupla = ("Maria", "Pedro", "José")
>>> tupla[0]
'Maria'
>>> tupla[0]="Fátima"

Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    tupla[0]="Fátima"
TypeError: 'tuple' object does not support item assignment
```

Nesse caso, foi criada uma tupla, depois retorna o primeiro elemento da tupla. Em seguida, tentamos modificar um elemento dessa tupla, o que resultou em erro.

6.4 Dicionários

Dicionários são conjuntos não ordenados de pares, onde o primeiro elemento do par, o índice, é chamado de chave e o outro de valor.

Um dicionário, em Python, suporta qualquer tipo de objeto, seja ele uma lista ou até mesmo outros dicionários e pode ser modificado. Para criar um dicionário basta declarar pares "chave:valor" separados por vírgula e delimitados por chaves.

```
>>> dic = {1:'um', 2:'dois', 3:'três'}
>>> dic[1]
'um'
>>> dic[4]='quatro'
>>> dic
{1: 'um', 2: 'dois', 3: 'tr\xeas', 4: 'quatro'}
```

Os dicionários possuem alguns métodos, entre eles:

items() - esse método retorna uma lista de tuplas, ou seja, todos os pares chave:valor na forma de tuplas;

```
>>> dic.items()
[(1, 'um'), (2, 'dois'), (3, 'tr\xeas'), (4, 'quatro')]
```

keys() - esse método retorna somente as chaves;

```
>>> dic.keys()
[1, 2, 3, 4]
```

values() - esse método retorna somente os valores;

```
>>> dic.values()
['um', 'dois', 'tr\xeas', 'quatro']
```

*get(*chave*)* - retorna o conteúdo da chave passada como parâmetro;

```
>>> dic.get(2)
'dois'
>>> print dic.get(5)
None
```

*has_key(*chave*)* - verifica se existe a chave passada como parâmetro, retornando true ou false;

```
>>> dic.has_key(5)
False
>>> dic.has_key(2)
True
```

*update(*dicionário*)* - atualiza um dicionário com base em outro passado como parâmetro. Caso elementos do primeiro dicionário existam também no segundo, esse sobrescreve o primeiro, ou se o segundo conter elementos exclusivos, serão adicionados ao primeiro dicionário.

```
>>> dic = {1:'um', 2:'dois', 3:'três'}
>>> dic2 = {1:'one', 2:'two', 5:'five'}
>>> dic.update(dic2)
>>> dic
{1: 'one', 2: 'two', 3: 'tr\xeas', 4: 'quatro', 5: 'five'}
```

7 Docstrings

Como o nome sugere, são strings que servem para documentar código sem o uso de ferramentas externas. Basta criar a string entre seqüências de três aspas.
Porém, é possível usar ferramentas externas também.

```
>>> def my_function():

        """
        Author : Marcelo Braga
        Date   : 15/06/2001
        Version: 0.1
        """

        print "Minha função"
```

8 O operador %

Esse operador é muito útil para processamento de texto.
Existem três:
%s – substitui strings
%d – substitui inteiros
%f – substitui floats

```
>>> nome = "Eliane"
>>> print "Meu nome é %s" % nome
Meu nome é Eliane

>>> numInt = 19
>>> print "Eu tenho %d anos" % numInt
Eu tenho 19 anos

>>> numFloat = 1.60
>>> print "Altura: %.2f m" % numFloat
Altura: 1.60 m

>>> numFloat = 54.80
>>> print "Peso: %10.1f kg" % numFloat
Peso:       54.8 kg
```

Obs.: "%.2f" corresponde a duas casas decimais e "%10.1", a dez espaços, uma casa decimal.

9 Funções

Funções são blocos de códigos que recebem parâmetros e retornam um resultado.
Além de funções, existem os procedimentos, que em Python, a única diferença entre os dois é: funções retornam valores e procedimentos não. Procedimentos, portanto, podem ser considerados tipos especiais de funções.
Existem dois tipos de funções: definidas pelo usuário e pré-definidas.
Para definir uma função basta usar o comando *def* seguido de um nome, parênteses e dois pontos.

```
>>> def meunome(nome):
        print "Meu nome é %s" % nome

>>> meunome('Daniela')
Meu nome é Daniela
>>>
```

Primeiro, criamos a função que se chamou "meunome", ela recebeu um parâmetro, "nome". Depois invocamos a função e passamos um valor (argumento), "Daniela". E por fim, a função nos retornou a mensagem dizendo o nome.

Você também pode definir mais de um parâmetro, para isso, basta usar a declaração *return*.

```
>>> def operacoes(a,b):
        soma = a+b
        multiplica = a*b
        return soma,multiplica

>>> operacoes(4,5)
(9, 20)
```

9.1 Argumentos default

Ainda existe um terceiro caso, são os chamados argumentos default. Nesse caso, você associa um valor ao argumento, ou parâmetro, que será usado caso o usuário não passe um valor.

```
>>> def estadocivil(estado='solteiro'):
```

```
print "Eu sou %s" % estado

>>> estadocivil()
Eu sou solteiro

>>> estadocivil('casado')
Eu sou casado
>>>
```

Você deve ter percebido o que aconteceu aqui. Ao invocar a função, o usuário não passou um valor, então o interpretador assumiu o valor passado na função. Depois o usuário corrigiu passando a informação correta.

9.2 Funções pré-definidas

Quanto às funções pré-definidas, Python possui uma série delas. Aqui estão alguns:

*range(*a,b*)* - recebe dois inteiros, retorna uma lista com todos os números inteiros entre a e b, sem incluir o b.

```
>>> range(1,3)
[1, 2]
```

*len(*a*)* - retorna o comprimento da variável a seja ela uma string, uma lista e assim por diante.

```
>>> a=['maçã', 'banana', 'uva', 'morango']
>>> len(a)
4
```

*list(*l*)* - converte uma seqüência em uma lista.

```
>>> l = "terra"
>>> list(l)
['t', 'e', 'r', 'r', 'a']
```

*tuple(*l*)* - converte uma seqüência em uma tupla.

```
>>> l = "amor"
>>> tuple(l)
('a', 'm', 'o', 'r')
```

*int(*numFloat*)* – converte um float em um inteiro.

```
>>> int(1.0)
1
```

*float(*numInt*)* – converte um inteiro em um float.

```
>>> float(5)
5.0
```

*str(*num*)* – converte qualquer tipo em uma string.

```
>>> str(50)
'50'
>>> str([1,2,3])
'[1, 2, 3]'
>>> str(("one","true", "three"))
"('one', 'true', 'three')"
```

*chr(*num*)* – recebe um inteiro, entre 0 e 255, e retorna o caracter correspondente da tabela ASCII.

```
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(98)
'b'
>>> chr(35)
'#'
```

*ord(*c*)* – recebe um caracter e retorna o seu código ASCII.

```
>>> ord('#')
35
>>> ord('d')
100
>>> ord('ç')
231
```

unichr – recebe um inteiro, entre 0 e 65535, e retorna o caracter Unicode correspondente.

*Round(*numfloat,num*)* – recebe um float e um número, e retorna um float arredondado com este número de casas decimais.

*pow(*num,exp*)* – recebe dois inteiros e retorna o resultado da exponenciação de num à ordem exp.

```
>>> pow(5,2)
25
>>> pow(3,4)
 81
```

*min(*a,b*)* – recebe dois valores de qualquer tipo e retorna o menor entre eles.

```
>>> min('a',6)
6
>>> min('a','A')
'A'
>>> min(5.13,8)
5.1299999999999999
>>> min(0.5,'a')
0.5
```

*max(*a,b*)* – recebe dois valores de qualquer tipo e retorna o maior entre eles.

```
>>> max('a','B')
'a'
>>> max('f',56)
'f'
>>> max(72.5,'H')
'H'
```

*abs(*num*)* – recebe um número e retorna seu valor absoluto.

```
>>> abs(32.5322656)
32.532265600000002
>>> abs(13)
13
```

*hex(*numInt*)* – recebe um inteiro e retorna uma string contendo a representação em hexadecimal.

```
>>> hex(12)
'0xc'
>>> hex(10)
'0xa'
>>> hex(0)
'0x0'
```

*oct(*numInt*)* – recebe um inteiro e retorna uma string contendo a representação em octal.

```
>>> oct(7)
'07'
>>> oct(8)
'010'
>>> oct(15)
'017'
>>> oct(16)
'020'
```

10 Módulos

Módulos são coleções de funções, uma forma inteligente de organizar e acessar código (funções) a ser reutilizado. Para usá-los basta importá-los, assim, o *import* localiza o módulo desejado, carrega suas funções e, finalmente, são executados os comandos de inicialização do módulo.
Módulos podem ser importados por outros módulos.
Python tem um grande número de módulos. Por exemplo, o módulo *calendar* que retorna o calendário do ano e mês desejados.

```
>>> import calendar
>>> calendar.prmonth(2003,2)
   February 2003
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28
```

Outros módulos: *sys*, *math*, *random*, *getopt*, *os.path*, *time*, são alguns dos módulos que Python oferece.
Além desses módulos instalados com o interpretador Python, existem, também, módulos independentes, tais como: *win32pipe*, *HTMLgen*, *mx*, *PIL*, etc.

11 Classes e instâncias

Como em Python tudo é objeto, não seria muito apropriado (embora não seja errado) chamarmos classes de objetos, mas sim de instâncias que, por sua vez, não deixam de ser objetos criados a partir de uma classe definida. Assim, classes são objetos (instâncias) diferentes dos demais. Quando cria uma instância, você está encapsulando nela os elementos de uma classe.
Para entendermos melhor o que é uma classe em programação, compare-a a uma receita de bolo ou de qualquer outra coisa. A receita te ensina a fazer o bolo, pois nela você encontra os ingredientes que irá precisar, as medidas e o modo de preparo. Porém, ter a receita sem fazer o bolo não é o suficiente. Então, você precisa arregaçar as mangas e mãos à obra!
Com a criação de uma instância (objeto), você atribui a ela propriedades e métodos de uma classe, assim como atribui ao bolo tudo que contem a receita, ingredientes e modos de prepará-lo. Para concluirmos, a classe seria a receita e a instância, o bolo.

```
>>> class Notas:

     def Media( self,lista ):

          soma= 0
          for t in lista:
               soma += t
          print soma / len( lista )


a =Notas()
>>> a.Media((10.0,7.5,6.5))
8.0
```

Criamos uma classe chamada “Notas” que contém uma função. Em seguida, invocamos a classe por meio da instância criada, “a”, e ainda invocamos a única função “Media()” passando os argumentos requeridos e, por fim, nos retornou um valor.

Existem alguns métodos em classes, entre eles:

__init__ - é o método opcional construtor, é invocado quando uma instância é criada;

__del__ - é invocado quando uma classe será destruída;

__*call*__ - é invocado quando uma instância é chamada como uma função.

12 Herança

Herança é um mecanismo de reutilização de todos os elementos de uma classe já existente em uma nova. Nesse caso, a classe herdada pode ser chamada de classe pai.
Você, ainda, pode modificar a classe pai na classe filho sem interferir no lugar onde ela (a classe pai) foi criada, além disso ainda pode estender a classe filha com propriedades e métodos exclusivos, ou seja, que não existem na classe pai.

13 Erros de sintaxe e exceções

São os dois tipos de erro mais comuns. Um erro de sintaxe ocorre quando você não respeita a sintaxe, digitando errado ou esquecendo de algo, como ":".

```
>>> if x < 10
SyntaxError: invalid syntax
```

Nesse caso, está faltando ":" no final da condição.

Também pode vir a ocorrer um erro mesmo que seu código esteja, sintaticamente, correto. Nesse caso trata-se de exceções.
Quando exceções ocorrem, o programa para e o interpretador gera uma mensagem de erro. No entanto, é possível resolver esse problema por tratar as exceções.
Para este fim, basta usar duas instruções: *try* e *except*.
Enquanto a instrução *try* contém comandos que serão executados, a instrução *except* executa o tratamento nas possíveis exceções desses comandos, ou seja, na instrução *except* você diz o que é para ser feito caso um erro ocorra durante a execução dos comandos.

```
>>> n = int(raw_input("Escolha uma chave: "))
Escolha uma chave: 6
>>> dic = {1:'um',2:'dois',3:'tres'}
>>> try:
     print dic[n]
except:
     print "Chave não existe"


Chave não existe
```

Se não houver exceções nos comandos, a instrução *except* é ignorada. É possível numa única instrução *try* criar diversas *except* com diferentes tipos de exceções, entre eles: *ValueError*, *IndexError*, *TypeError*.

Existem, ainda, duas outras instruções: *raise* e *finally*. No primeiro caso, força a ocorrência de um determinado tipo de exceção.
Quando existir a instrução *finally*, seus comandos serão executados independentemente de existir ou não exceções, e serão executados antes das exceções.
A instrução *try* nunca permite que o usuário use *except* e *finally*, simultaneamente.

Além de existir diversos tipos pré-definidos de exceções no Python, o usuário tem a liberdade de criar seus próprios tipos de exceções através de classes.

14 Manipulação de arquivos

Em Python, assim como em outras linguagens, é possível manipular arquivos. Desse modo, o usuário consegue abrir, ler, alterar, gravar e fechar um arquivo.
Tudo o que você precisa para manipular, corretamente, um arquivo é conhecer os métodos responsáveis por cada coisa.
A função *open()* retorna um objeto arquivo, em outras palavras, abre o arquivo que você especificar.

```
>>> arq = open('C:\hoje.txt')
```

Você, ainda, tem a opção de passar como parâmetro o modo como vai abrir o arquivo. Caso passe um "r", seu arquivo será somente de leitura; se passar "a", poderá adicionar conteúdo ao final do arquivo; caso passe "w", o arquivo será apagado e aberto para escrita, no entanto, se ele não existir será automaticamente criado, etc.
Caso o usuário não passe o segundo parâmetro, Python assume o "r".

```
>>> t = open('C:\meuarquivo.txt','w')
>>> print a
<open file 'C:\meuarquivo.txt', mode 'w' at 0x00A85A40>
```

14.1 Métodos do objeto arquivo

Podemos a partir de um arquivo aberto, utilizar métodos:

write(string) – grava a string no arquivo aberto.

```
>>> t.write('Meu novo conteúdo')
```

read() - retorna todo o conteúdo do arquivo.

```
>>> var = arq.read()
>>> print var
Hoje é dia de alegria
```

readline() - retorna apenas uma linha do arquivo.

seek() - posiciona o cursor na posição indicada.

```
>>> arq.seek(0)
```

tell() - retorna a posição, atual, do cursor no arquivo.

```
>>> arq.tell()
```

Close() - fecha o arquivo.

```
>>> t.close()
```