PerguntasFrequentes / SobreProgramacao

Conteúdo

O Python não gostou do meu editor de textos

Um cuidado extra é necessário quanto ao editor de texto que você escolheu para editar arquivos em Python. Mas antes, uma breve explicação sobre a linguagem é necessária.

Python usa endentação como delimitação de bloco (ao contrário das chaves usadas pela linguagem C ou as palavras-chaves `Begin`/`end` do Pascal). Instruções são consideradas pertencentes ao mesmo bloco quando usam o mesmo distanciamento em relação à coluna zero. Aconselha-se trabalhar apenas com espaços, sem misturá-los com TABs (mais detalhes sobre isso na página GuiaDeEstilo), pois fica difícil a detecção de onde um ou outro foi utilizado, ocasionando erros de sintaxe crípticos. Dito isso, a regra mais comum de endentação é que cada bloco seja deslocado quatro espaços em relação ao anterior. Mas, qualquer que seja a regra, é mais importante a adesão a uma única regra por todo o arquivo.

Voltando ao editor de textos, verifique como ele trabalha com TABs e se são convertidos em espaços. Caso contrário, considere usar um editor mais apropriado para programadores. Muitos editores, como o `vim`, permitem mudar as configurações de fábrica. É uma boa idéia checar a documentação do seu editor favorito.

Os meus programas recebem um aviso "DeprecationWarning: Non-ASCII character in file". O que é isso?

É provavel que o seu programa esteja usando caracteres acentuados sem uma declaração de codificação no começo do programa. Até a versão 2.2 do Python, isso não representava um problema. A partir da versão 2.3, o Python passou a tratar de forma diferente os caracteres acentuados, visando uma melhor integração com a codificação Unicode. Neste sentido, é uma das poucas iniciativas válidas do ponto de vista do suporte nativo em uma linguagem de programação.

O motivo para isso é que os caracteres acentuados não fazem parte da codificação padrão ASCII. Existem várias formas de codificá-los usando caracteres adicionais (chamados erroneamente por alguns de "ASCII extendido"). Isso geralmente não causa problemas porque a codificação dentro de um mesmo país é razoavelmente padronizada. Porém, se você quiser trocar arquivos de código com um usuário em outro país, é possível que o seu código fonte seja lido com os caracteres incorretos. Já o Unicode é um padrão internacional de codificação superior ao ASCII, e que pode ser codificado dentro de arquivos normais. Para isso, é preciso que seja definida a codificação correta -- e é exatamente esta informação que o Python está pedindo. No caso do Brasil, a codificação ISO-8859-1 identifica um mapeamento padrão dos acentos.

Assim, é necessário informar ao Python qual a codificação em que o arquivo está escrito. Isso permitirá, por exemplo, que ele traduza as strings nativas (escritas em ASCII no código fonte) sejam traduzidas pelo programa para uma representação Unicode de forma automática. Para fazer isso, basta acrescentar uma linha de comentário contendo "`-*- coding: XXX -*-`" informando a codificação desejada. Esta linha deve ser obrigatoriamente a primeira ou segunda linha do arquivo. Por exemplo:

```
# -*- coding: ISO-8859-1 -*-
```

Ativa a codificação `ISO-8859-1`. Para o caso de um editor que utilize `UTF-8`:

```
# -*- coding: UTF-8 -*-
```

Observe que a codificação não é sensível ao uso de letras maísculas e minúsculas.

Como usar uma variável global em uma função?

Praticamente todo programador de Python já tentou fazer algo do tipo:

Esconder número das linhas

```
1 v = 1
2 def f():
3     print v
```

E funciona! Porém uma pequena variação...

Esconder número das linhas

```
1 v = 1
2 def f():
3     print v
4     v = 10
```

E recebeu o erro `UnboundLocalError: local variable 'v' referenced before assignment`. Isso ocorre porque qualquer variável definida dentro de uma função é considerada local àquela função, se houver uma atribuição a esta variável no escopo da função. Como redefinimos o valor de 'v' dentro do corpo da função, o interpretador considera esta variável como local. Sendo local, a variável 'v' no escopo global é inexistente para a função no momento do`print`, gerando o erro. A solução é explicitamente declarar a variável como global no início da função.

Esconder número das linhas

```
1 v = 1
2 def f():
3     global v
4     print v
5     v = 10
```

Em Python, variáveis que são apenas chamadas dentro de uma função são automaticamente globais. Se uma variável for definida em qualquer ponto da função, é considerada local, e você precisa declará-la explicitamente como global.

Como atualizo vários valores de um dicionário?

A classe de dicionários possui um método chamado `update` que recebe como parâmetro um dicionário com os novos ítens de chave e valor, como em`<instância>.update(<dicionário>)`. Por exemplo:

Esconder número das linhas

```
1 >>> d = { 'foo': 1, 'bar': 2 }
2 >>> d2 = { 'foo': 123, 'qux': 456, 'blah': 1337 }
3 >>> d.update(d2)
4 >>> d
5 {'qux': 456, 'foo': 123, 'bar': 2, 'blah': 1337}
```

Instalei um módulo no /usr/local/lib/python e agora como faço para o Python importar esse módulo?

É necessário adicionar o novo caminho à variável `path`, que sabe onde estão todas as bibliotecas do Python, para que o comando `import` possa abrir esse módulo. Existem duas soluções para isso, a primeira é manipular diretamente a variável `sys.path`. Essa variável é uma lista de caminhos para bibliotecas.

Esconder número das linhas

```
1 import sys
2 sys.path.append('/usr/local/lib/python')
```

Uma outra solução é definir a variável de ambiente `$PYTHONPATH` com um caminho ou vários caminhos delimitados por `:` (igual a variável `$PATH`). Essa solução é mais genérica e transparente. Basta adicionar no seu arquivo de inicialização do *shell* a seguinte linha:

```
PYTHONPATH=/usr/local/lib/python; export PYTHONPATH
```

Agora o comando `import` vai encontrar corretamente a sua biblioteca.

Gosto de usar uma IDE pra desenvolver, qual a melhor IDE para Python?

Existem diversas IDEs que podem ser usadas no desenvolvimento de aplicações em Python. Na página IdesPython você encontra informações sobre elas.

Como eu configuro o vim para que utilize o padrão de alinhamento sugerido pelo python.org?

Ponha em um comentário no seu arquivo .py

```
# vim:ts=4:sw=4:et
```

Ou seja, a tecla TAB pula 4 caracteres (ts=4), 4 caracteres para alinhamento (sw=4), expande TAB para espaços (et).

Você pode também editar seu arquivo ".vimrc" colocando as seguintes linhas nele:

```
et
ts=4
sw=4
```

É possível criar um arquivo executável de um programa em Python?

É possível, mas só é recomendável se você tiver um bom motivo pra fazer isso. Se a intenção for melhorar o desempenho do código, não irá fazer diferença. Se você quer facilitar a distribuição do seu programa para aqueles que não têm o interpretador Python instalado, pode ser uma vantagem, mas também pode ser um incômodo para usuários que já o tenham. Se, mesmo assim, você tem uma necessidade real, existem algumas ferramentas capazes de fazer isso. Duas delas são o 'freeze', incluído na distribuição (no diretório Tools/freeze), que exige um compilador de C no momento de criação do executável, e oPy2Exe, disponível somente para Windows e que pode ser encontrado em http://starship.python.net/crew/theller/py2exe.

Além do Py2Exe há também o cx_Freeze, que funciona tanto pra Linux quanto para Windows. Mais informações no site: http://starship.python.net/crew/atuining/cx_Freeze/

Como eu compilo um programa em Python?

Python não é uma linguagem compilada, e sim interpretada. Desta forma, você nunca vai 'compilar', 'linkar' e 'gerar um executável' de um programa feito em Python; pelo menos não da maneira que você faz com um programa em C. O fonte estará quase sempre disponível e ele é sempre verificado no momento da execução. Tendo dito isso, aqui vão as diferenças da visão clássica de compiladores e interpretadores para o que acontece em Python.

A interpretação clássica diz que, ao ser executado, o programa seria analisado e interpretado linha por linha, os comandos correspondendo a instruções do que a máquina deve fazer. A compilação clássica diz que o programa é convertido para instruções em linguagem de máquina e executados diretamente pelo processador.

O que Python faz é um meio termo - de vez em quando chamado de 'interpilação'. Quando você roda o seu programa em Python, o interpretador não faz a análise linha por linha, mas sim converte o seu programa para um conjunto de instruções intermediárias, que são códigos de uma máquina virtual projetada especificamente para o Python. Esses códigos são, eles sim, interpretados. O motivo da decisão por esse modelo é que ele é mais fácil de ser implementa-do, permite um dinamismo muito maior e torna muito mais fácil a portabilidade de programas entre sistemas. Para agilizar o carregamento do

programa na execução, o interpretador grava os códigos em um arquivo .pyc que é lido e utilizado caso não esteja defasado.

É o mesmo esquema utilizado pelo Java, exceto que, com o Python, o processo de compilação é transparente, e não são necessários arquivos de construção, configuração e etc.

Há, no entanto, formas de gerar 'executáveis' Python. Programas como o freeze para os Unices e Py2Exe para o Windows fazem o seguinte: compilam o seu código fonte gerando os arquivos .pyc, e linkam o interpretador e as bibliotecas básicas juntos com esses dados em um arquivo 'executável'. Note que o programa não foi compilado (não mais do que antes), e portanto não vai haver ganho de velocidade. Mas pode ser uma opção se você quiser 'esconder' o seu código-fonte, ou garantir que a distribuição do seu programa vai ter todas as bibliotecas necessárias, sem dependências externas.

Ouvi muitos boatos de que as Threads do Python não funcionam muito bem

Funcionam bem sim.

Existe a limitação do Global Interpreter Lock ou GIL, o que em termos práticos significa que threads não farão máximo uso do paralelismo quando existirmais de um processador. Ainda que eu tenha dois processadores e duas threads, apenas uma thread poderá estar executando código dentro das regiões protegidas pelo GIL. Isso significa que para máximo paralelismo é melhor modelar a sua aplicação como dois processo cooperantes, do que um processo com duas threads cooperantes.

Em máquinas monoprocessadas, dá na mesma.

Duas ou mais threads compartilham a mesma memória global de um processo. Para evitar que duas ou mais threads causem uma "race condition",recursos são protegidos por regiões críticas (existem outras construções para sincronização entre threads). O interpretador Python tem diversas regiões críticas controladas pela mesma tranca, que é o GIL. Assim, quando uma thread está dentro de uma dessas regiões protegidas, as demais threads tem que esperar a primeira liberar a tranca. Mesmo que outras CPUs estejam de bobeira.

Essa decisão de construção da PVM (Python Virtual Machine) facilita a depuração e diminui o risco de deadlocks (o que é bom), porém diminui oparalelismo (o que é ruim).

É possível contornar essa limitação criando dois processos (duas PVMs) que se comunicam explicitamente (sockets, RPC, banco de dados, etc) sem usar memória compartilhada.

Aplicações que precisam de grande escalabilidade usam uma abordagem híbrida (como o servidor Apache), onde o aplicativo é composto por váriosprocessos multi-threaded.

Outra forma de melhorar o paralelismo (de I/O) seria usar um framework assíncrono como o Twisted. A economia advém da ausência das "custosas" trocas de contextos entre threads.

Se não for preciso paralelismo com I/O (leitura de arquivos ou comunicação com a rede), uma opção seria usar co-rotinas (através de geradores) ou até mesmo apelar para o Stackless Python. Simuladores e jogos tem obtido sucesso com essa abordagem.

Como faço para definir um atributo como privado ou público?

A resposta curta é: não faz. Não é da práxis pythônica "privatizar" o que quer que seja. Então, para complementar, eu recomendo veementemente que você não se preocupe muito com isto. Por outro lado, você pode "embaralhar" o nome de atributos e métodos com o prefixo ___. Neste caso, os nomes ficarão diretamente inacessíveis a código exterior á classe. Veja o exemplo:

Esconder número das linhas

```
>>> class Foo(object):
...     def __init__(self, v):
...         self.__v = v
...     def __bar(self):
...         return self.__v
...
>>> f = Foo(3)
>>> f.__v
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Foo' object has no attribute '__v'
>>> f.__bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Foo' object has no attribute '__bar'
```

Entretanto, os métodos não foram feitos propriamente privados. O que acontece é que, agora, os nomes dos métodos foram alterados. Eles podem ser acessados sem grandes problemas acrescentando-se um underscore e o nome da classe antes do método:

Esconder número das linhas

```
>>> f._Foo__v
3
>>> f._Foo__bar()
3
```

O que se costuma fazer, porém, é o seguinte: se um atributo ou método não precisa ser acessado ou chamado fora dos métodos da classe, acrescenta-se um underscore antes dele:

Esconder número das linhas

```
>>> class Foo(object):
...     def __init__(self, v):
...         self._v = v
...     def _bar(self):
...         return self._v
...
>>> f = Foo(3)
>>> f._v
3
>>> f._bar()
3
```

Isto não impede que eles sejam acessados, mas serve como padrão de codificação para dizer que talvez não seja uma boa idéia mexer no atributo.

A primeira edição do The Python Papers traz um artigo sobre isto, e a PEP 8 toca no assunto.

Resumindo: não se preocupe em tornar as coisas privadas. Só vai dar dor de cabeça, em Python... Considere, porém, seguir os padrões de estilo que dizem onde é bom mexer e onde não é.

É preciso criar um método main que é o ponto de partida do programa?

Em Python, você não precisa definir a função/método main. Todo código Python é, sempre, um conjunto de instruções, incluindo as cláusulas para criar classes e funções.

Considere o código abaixo:

Esconder número das linhas

```
 1 class Foo(object):
 2     def __init__(self, v):
 3         self.v = v
 4
 5 print "As coisas são executadas conforme aparecem!"
 6
 7 def foo(v):
 8     return v
 9
10 print "Viu só?"
11 print Foo(3).v
12 print foo(4)
```

Executar este script resultará na saída

```
As coisas são executadas conforme aparecem!
Viu só?
3
4
```

Claro que fazer um programa assim é feio. O mais adequado é definir as funções e classes em módulos e importar um script que efetivamente funcione como o "main" do programa:

Esconder número das linhas

```
1 from meu_modulo import Foo, foo
2 print "As coisas são executadas conforme aparecem!"
3
4 print "Viu só?"
5 print Foo(3).v
6 print foo(4)
```

Se você quiser que um determinado módulo possua um "main", outra tradição pythônica é conferir se o módulo foi importado e, se não foi, executar o que se queira. Acontece que, por exemplo, o

módulo "meu_modulo" que citei acima pode tanto ser executado quanto chamado em linha de comando - ele é um script afinal.

Como saber se ele foi importado ou foi chamado como script? Simples: quando um módulo é executado (seja por importação, seja por invocação), a variável interna `__name__` é inicializada com o nome do módulo. Se ele foi importado, `__name__` é o nome do módulo; se ele foi invocado, `__name__` possui a string "`__main__`" como valor. Logo, queremos executar o código apenas se `__name__` é igual a "`__main__`". Como no exemplo abaixo:

Esconder número das linhas

```
 1 # começa módulo meu_modulo (meu_modulo.py)
 2 class Foo(object):
 3     def __init__(self, v):
 4         self.v = v
 5
 6 def foo(v):
 7     return v
 8
 9 if __name__== "__main__":
10     print "As coisas são executadas conforme aparecem!"
11     print "Viu só?"
12     print Foo(3).v
13     print foo(4)
14 # termina
```

Se o módulo for importado, o código não será executado.

Como "escondo" a janela do prompt do MS-DOS que aparece quando executo um script?

Em um sistema operacional Windows, ao executar um *script* Python comum com um duplo clique ou análogos, um *prompt* de comando aparece invariavelmente, mesmo que o script possua uma interface gráfica. Para evitar que isto aconteça, masta nomear o arquivo do script com a extensão ".pyw", ao invés de ".py". Se o arquivo se chama "program.py", por exemplo, renomeie-o para "program.pyw" e a janela do DOS não mais aparecerá.

Como posso alterar strings em Python?

As *strings* em Python são imutáveis, isto é, você não pode alterar a composição de uma *string*. A principal razão disso é permitir que *strings* possam ser utilizadas como chaves de dicionários.

Esconder número das linhas

```
1 >>> s = "abc"
2 >>> s[1] = 'd'
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in ?
5 TypeError: object does not support item assignment
6 >>> s
7 'abc'
8 >>>
```

Se você quiser alterar uma *string*, você terá de gerar uma nova. Por exemplo, se você quiser concatenar duas *strings*, terá de gerar uma nova *string*, como abaixo:

Esconder número das linhas

```
>>> a = 'abc'
>>> b = a+'d'
>>> a
'abc'
>>> b
'abcd'
>>>
```

Obviamente, nada impede que você referencie a nova *string* através da variável anterior, como abaixo.

Esconder número das linhas

```
>>> a = 'abc'
>>> id(a)
-1210723904
>>> a += 'd'
>>> a
'abcd'
>>> id(a)
-1210723488
>>>
```

Note como os `id`s são diferentes. É que o objeto que a variável `a` referencia agora é diferente da *string* anterior - em outras palavras, uma *string* foi criada, substituindo a anterior. O código abaixo deixa isso mais claro:

Esconder número das linhas

```
>>> a = 'abc'
>>> b = a
>>> a
'abc'
>>> id(a)
-1210723904
>>> b
'abc'
>>> id(b)
-1210723904
>>> a += 'd'
>>> a
'abcd'
>>> b
'abc'
>>> id(a)
-1210723552
>>> id(b)
-1210723904
>>>
```

Note como `a` e `b` ambos referenciam a mesma *string*, a princípio. Note como o `id` de `a` é o mesmo `id` de `b`. Quando, porém, fazemos `a += 'd'`, estamos criando uma nova *string* e fazendo com que `a` aponte para ela. Note como `b` continua com o `id` anterior, mas `a` aponta para um `id` diferente. Se você conhece o conceito de referências, certamente apreendeu a idéia.

Todos os métodos e operações que transformam *strings* inevitavelmente retornam *strings* novas, assim como qualquer função que aja sobre *strings*, transformando-as.