Série Desenvolvimento *Eficaz* de Software

Scott Meyers, Editor

PYTHON *Eficaz*

59 maneiras de programar melhor em Python

novatec

Brett Slatkin

Elogios a Python Eficaz

“Cada item descrito em Python Eficaz ensina uma lição completa, com seu próprio código-fonte. Isso faz com que o livro possa ser lido de forma completamente aleatória: os itens são fáceis de procurar e estudar em qualquer ordem que o leitor deseje. Eu recomento Python Eficaz a meus alunos como uma fonte inestimável de orientação para uma gama bastante abrangente de assuntos de interesse do programador Python de nível intermediário.”

— Brandon Rhodes, engenheiro de software no Dropbox e membro da mesa na PyCon 2016-2017

“Programo em Python há anos e pensei que o conhecesse muito bem. Graças a esse pequeno tesouro de técnicas e dicas, percebi que posso fazer muito mais pelo meu código para torná-lo mais rápido (por exemplo, empregando as estruturas de dados nativas), mais fácil de ler (por exemplo, assegurando que os argumentos contenham apenas palavras-chave) e muito mais pythônico (por exemplo, usando zip para iterar sobre listas em paralelo).”

— Pamela Fox, educadora, Khan Academy

“Se eu tivesse este livro quando migrei do Java para o Python, teria me poupado muitos meses de código repetido, o que acontecia a cada vez que eu me dava conta de que estava fazendo as coisas de modo ‘não pythônico’. Este livro reúne a vasta maioria daquelas dicas de Python que todo programador deve, obrigatoriamente, conhecer, eliminando a necessidade de tropeçar em cada uma delas, uma por uma, ao longo dos anos. A abrangência do livro é impressionante, começando com a importância do PEP8, passando por todas as expressões mais importantes do Python, pelo projeto eficiente de funções, métodos e classes, uso eficaz das bibliotecas nativas, projeto de API de qualidade, testes e medição de desempenho – não há assunto importante que não seja abordado aqui. É uma introdução fantástica a respeito do que seja, realmente, ser um programador em Python, tanto para o iniciante como para o desenvolvedor experiente.”

— Mike Bayer, criador da SQLAlchemy

“Python Eficaz levará suas habilidades em Python a um novo patamar, com orientações para aprimorar o estilo e a funcionalidade do seu código.”

— Leah Culver, evangelizadora de desenvolvimento, Dropbox

“Este livro é um recurso excepcionalmente maravilhoso para desenvolvedores experientes em outras linguagens, que assim podem adotar o Python de forma rápida sem precisar usar a sintaxe mais básica da linguagem, podendo já adotar um estilo de código mais pythônico. A organização do livro é clara, concisa e fácil de digerir, cada item e capítulo se sustentam sozinhos como uma meditação a respeito de determinado tópico. O livro cobre toda a gama de sintaxes em Python puro sem confundir o leitor com a complexidade do ecossistema completo do Python. Para desenvolvedores mais experientes, o livro oferece exemplos aprofundados de estruturas da linguagem com os quais possivelmente ainda não se depararam, bem como exemplos de recursos menos conhecidos da linguagem. Fica claro que o autor é excepcionalmente fluente em Python e usa sua experiência profissional para alertar o leitor para os erros mais comuns, sempre sutis, e os modos corriqueiros de falhas. Além disso, o livro faz um excelente trabalho ao apontar as sutilezas entre Python 2.X e Python 3.X e pode servir como um curso de reciclagem para os que estão migrando de uma variante de Python para outra.”

— Katherine Scott, chefe de desenvolvimento de software, Tempo Automation

“Este é um grande livro, tanto para iniciantes como para o programador experiente. Os exemplos de código e explicações são muito bem pensados e explicados de forma concisa sem deixar de serem completos.”

— C. Titus Brown, professor, Davis University of California

“Este é um recurso imensamente útil para aprender Python avançado e construir software mais claro e fácil de manter. Qualquer um que esteja querendo levar seus conhecimentos de Python a um novo patamar serão altamente beneficiados se puserem em prática o conteúdo deste livro.”

— Wes McKinney, criador de pandas e autor de *Python for Data Analysis*; além de engenheiro de software na Cloudera

PYTHON *Eficaz*

59 maneiras de programar melhor em Python

Brett Slatkin

Novatec

Editor: Rubens Prates
Tradução: Henrique Cesar Ulbrich / Design for Context
Revisão gramatical: Smirna Cavalheiro
Editoração eletrônica: Carolina Kuwabata
Assistente editorial: Priscila A. Yoshimatsu

ISBN: 978-85-7522-600-1

Histórico de edições impressas:

Julho/2016 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: *novatec@novatec.com.br*
Site: *www.novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *linkedin.com/in/novatec*

À nossa família, aos que amamos e aos que perdemos.

Sumário

Elogios a Python Eficaz

Prefácio

O que este livro aborda
Convenções usadas neste livro
Onde obter o código e a errata
Como entrar em contato conosco

Agradecimentos

Sobre o autor

Capítulo 1 ▪ Raciocínio pythônico

Item 1: Saiba qual versão de Python está em uso
Item 2: Siga o Guia de Estilo PEP 8
Item 3: Saiba as diferenças entre bytes, str e unicode
Item 4: Escreva funções auxiliares em vez de expressões complexas
Item 5: Saiba como fatiar sequências
Item 6: Evite usar start, end e stride em uma mesma fatia
Item 7: Use abrangências de lista em vez de map e filter
Item 8: Evite mais de duas expressões em abrangências de lista
Item 9: Considere usar expressões geradoras em abrangências muito grandes
Item 10: Prefira enumerate em vez de range
Item 11: Use zip para processar iteradores em paralelo
Item 12: Evite usar blocos else depois de laços for e while
Item 13: Use todo o potencial dos blocos try/except/else/finally

Capítulo 2 ▪ Funções

Item 14: Prefira exceções em vez de devolver None
Item 15: Saiba como closures interagem com os escopos das variáveis
Item 16: Prefira geradores em vez de retornar listas
Item 17: Seja conservador quando iterar sobre argumentos

Item 18: Reduza a poluição visual com argumentos opcionais
Item 19: Implemente comportamento opcional usando palavras-chave como argumentos
Item 20: Use None e docstrings para especificar argumentos default dinâmicos e específicos
Item 21: Garanta a legibilidade com argumentos por palavras-chave

Capítulo 3 ▪ Classes e herança

Item 22: Prefira classes auxiliares em vez de administrar registros complexos com dicionários e tuplas
Item 23: Aceite funções para interfaces simples em vez de classes
Item 24: Use o polimorfismo de @classmethod para construir objetos genericamente
Item 25: Inicialize classes ancestrais com super
Item 26: Use heranças múltiplas apenas para classes utilitárias mix-in
Item 27: Prefira atributos públicos em vez de privativos
Item 28: Herde da classe collections.abc para obter tipos de contêiner personalizados

Capítulo 4 ▪ Metaclasses e atributos

Item 29: Use atributos comuns em vez dos métodos get e set
Item 30: Considere usar @property em vez de refatorar atributos
Item 31: Use descritores para implementar métodos reutilizáveis de @property
Item 32: Use __getattr__, __getattribute__ e __setattr__ para atributos preguiçosos
Item 33: Valide subclasses com metaclasses
Item 34: Registre a existência de uma classe com metaclasses
Item 35: Crie anotações de atributos de classe com metaclasses

Capítulo 5 ▪ Simultaneidade e paralelismo

Item 36: Use subprocess para gerenciar processos-filho
Item 37: Use threads para bloquear I/O e evitar paralelismo
Item 38: Use Lock para evitar que as threads iniciem condições de corrida nos dados
Item 39: Use Queue para coordenar o trabalho entre as threads
Item 40: Considere usar corrotinas para rodar muitas funções simultaneamente
Item 41: Considere usar concurrent.futures para obter paralelismo real

Capítulo 6 ▪ Módulos nativos

Item 42: Defina decoradores de função com functools.wraps
Item 43: Considere os comandos contextlib e with para um comportamento reutilizável de try/finally
Item 44: Aumente a confiabilidade de pickle com copyreg
Item 45: Use datetime em vez de time para relógios locais
Item 46: Use algoritmos e estruturas de dados nativos
Item 47: Use decimal quando a precisão for de importância vital
Item 48: Saiba onde encontrar os módulos desenvolvidos pela comunidade

Capítulo 7 ▪ Colaboração

Item 49: Escreva docstrings para toda e qualquer função, classe e módulo
Item 50: Use pacotes para organizar módulos e criar APIs estáveis
Item 51: Defina uma Exception-raiz para isolar chamadores e APIs
Item 52: Saiba como romper dependências circulares
Item 53: Use ambientes virtuais para criar dependências isoladas e reprodutíveis

Capítulo 8 ▪ Produção

Item 54: Crie código com escopo no módulo para configurar os ambientes de implementação
Item 55: Use strings com a função repr para depuração
Item 56: Teste absolutamente tudo com unittest
Item 57: Prefira usar depuradores interativos como o pdb
Item 58: Meça os perfis de desempenho antes de otimizar o código
Item 59: Use tracemalloc para entender o uso e os vazamentos de memória

Prefácio

A linguagem de programação Python possui nuances e pontos fortes únicos que podem ser difíceis de dominar. Programadores familiarizados com outras linguagens muitas vezes escrevem código em Python com uma mentalidade limitada, em vez de tirar partido de toda a sua expressividade. Outros desenvolvedores vão longe demais na direção oposta, exagerando no emprego de recursos do Python que podem causar muita confusão mais tarde.

Este livro oferece um insight do modo *pythônico* de escrever programas: a melhor maneira de usar Python. Consideramos que o leitor já conhece os fundamentos da linguagem. Os programadores novatos aprenderão as melhores práticas para usar os recursos do Python, enquanto os desenvolvedores experientes aprenderão como adotar e absorver a estranheza de uma ferramenta nova com confiança.

Meu objetivo é preparar o leitor para causar grande impacto com o Python.

O que este livro aborda

Cada capítulo deste livro contém um conjunto numeroso de itens, mas todos estão relacionados. Sinta-se convidado a saltar itens ou lê-los em qualquer ordem, de acordo com seu interesse. Cada item contém orientação concisa e específica, explicando como escrever programas em Python de forma eficaz. Em cada item também há conselhos sobre o que fazer, o que evitar, como chegar ao equilíbrio correto e por que essa é a melhor escolha.

Os itens no livro servem tanto para Python 3 como Python 2 (consulte o Item 1: “Saiba qual versão de Python está em uso”). Quem estiver usando runtimes alternativos como Jython, IronPython ou PyPy também vai conseguir aplicar a maioria dos itens.

Capítulo 1: Raciocínio pythônico

A comunidade Python costuma usar o adjetivo *pythônico* (em inglês, *Pythonic*) para descrever qualquer código que siga um estilo em particular. O dialeto

preferencial do Python emergiu ao longo do tempo através da experiência no seu uso no dia a dia e no trabalho colaborativo com outros programadores. Este capítulo descreve a melhor maneira de fazer as coisas mais comuns em Python.

Capítulo 2: Funções

Em Python, as funções têm um grande número de recursos extras que tornam a vida do programador muito mais fácil. Alguns deles são semelhantes aos encontrados em outras linguagens, mas a maioria é exclusiva do Python. Este capítulo mostra como usar funções para deixar clara a intenção, promover sua reutilização e reduzir o número de bugs.

Capítulo 3: Classes e herança

Sendo o Python uma linguagem orientada a objetos, para que seja possível realizar coisas nele é necessária a criação de novas classes e definir como elas interagem por meio de suas interfaces e hierarquias. Este capítulo mostra como usar classes e herança para expressar o comportamento que se pretende dar ao objeto.

Capítulo 4: Metaclasses e atributos

Metaclasses e atributos dinâmicos são recursos muito poderosos no Python. Contudo, eles também facilitam a implementação de comportamentos inesperados e absolutamente bizarros. Este capítulo mostra as estruturas de linguagem mais comuns para usar esses mecanismos, assegurando que obedeçam à *regra da menor surpresa possível*.

Capítulo 5: Simultaneidade e paralelismo

O Python facilita a criação de programas simultâneos que fazem muitas coisas diferentes ao mesmo tempo. O Python pode ser usado para fazer trabalho paralelo por meio de chamadas de sistema, subprocessos e extensões em linguagem C. Este capítulo mostra como usar o Python da melhor maneira possível nessas situações sutilmente diferentes.

Capítulo 6: Módulos nativos

O Python já vem com muitos dos módulos importantes e necessários para

escrever qualquer programa. Esses pacotes nativos estão tão intimamente entrelaçados com o Python-padrão que poderiam muito bem ser considerados parte da especificação da linguagem. Este capítulo cobre os módulos nativos essenciais.

Capítulo 7: Colaboração

Escrever em equipe um programa em Python requer que cada programador escreva, deliberadamente, o código seguindo um mesmo estilo. Mesmo se estiver trabalhando sozinho, o programador precisa entender como usar módulos escritos por outras pessoas. Este capítulo mostra as ferramentas nativas e melhores práticas para permitir que as pessoas trabalhem juntas em programas Python.

Capítulo 8: Produção

O Python tem recursos que facilitam sua adaptação a múltiplos ambientes de produção. Ele também possui módulos nativos que auxiliam no aprimoramento da segurança (hardening) de seus programas, tornando-os à prova de balas. Este capítulo mostra como usar o Python para depurar, otimizar e testar seus programas para maximizar a qualidade e o desempenho em tempo de execução.

Convenções usadas neste livro

Os trechos de código em Python mostrados neste livro estão em tipografia monoespaçada e com destaque de sintaxe. Empreguei alguma “licença poética” em relação ao Guia de Estilo do Python para fazer com que os exemplos de código pudessem caber nas páginas do livro ou para destacar algum ponto importante. Quando as linhas forem muito longas, usei o caractere ➥ para indicar onde ocorre a quebra. Alguns exemplos foram resumidos e os trechos suprimidos foram marcados com reticências em comentários (#. . .) para indicar que eles existem, mas não foram representados por não serem essenciais para explicar aquele ponto. Também deixei de fora a documentação embutida, para reduzir o tamanho dos exemplos de código. Eu sugiro com veemência que você não faça isso em seus próprios projetos. Pelo contrário, siga o Guia de Estilo à risca (consulte o Item 2: “Siga o Guia de Estilo PEP 8”) e escreva a documentação corretamente (consulte o Item 49: “Escreva docstrings para toda e

qualquer função, classe e módulo”).

Muitos dos trechos de código neste livro estão acompanhados do resultado apresentado na tela (ou, simplesmente, “saída”) que é produzido ao rodar o código. Quando eu digo “saída”, refiro-me ao console ou terminal: é o que se vê quando o programa em Python é executado no interpretador interativo. A representação da saída está em tipografia monoespaçada e precedida por uma linha composta pelos caracteres >>> (o prompt interativo do Python). A ideia é poder digitar os trechos de código diretamente no shell do Python e reproduzir a saída esperada.

Por fim, há outras seções em tipografia monoespaçada que não estão precedidas pelo >>>. Elas representam a saída de programas executados fora do interpretador Python. Esses exemplos normalmente iniciam com um caractere $, indicando que os programas estão sendo executados em um shell como o Bash.

Onde obter o código e a errata

É útil visualizar alguns dos exemplos deste livro como programas completos sem que haja texto explicativo separando cada trecho. Isso também permite que possamos fazer experiências alterando o código e entendendo por que o programa funciona como descrito. O código-fonte de todos os trechos de código deste livro está disponível em seu site (*http://www.effectivepython.com*). Quaisquer erros encontrados no texto ou em códigos terão correções publicadas nesse mesmo site.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

  http://www.novatec.com.br/catalogo/7522510-python-eficaz

- Página da edição original em inglês

  http://www.effectivepython.com

Para obter mais informações sobre os livros da Novatec, acesse nosso site em *http://www.novatec.com.br*.

Agradecimentos

Este livro não teria sido possível sem a orientação, apoio e encorajamento de muitas pessoas que fazem parte da minha vida.

Começo agradecendo a Scott Meyers, pela coleção de livros Desenvolvimento Eficaz de Software. Li *C++ Eficaz* quando tinha 15 anos e me apaixonei pela linguagem. Sem dúvida alguma foi o livro de Scott que me levou à vida acadêmica e ao meu primeiro emprego na Google. Sou imensamente grato pela oportunidade de escrever este livro.

Agradeço também aos meus revisores técnicos, pela profundidade e abrangência de seus comentários e correções: Brett Cannon, Tavis Rudd e Mike Taylor. Outro muito obrigado vai para Leah Culver e Adrian Holovaty, por me convencerem de que este livro seria uma boa ideia. A meus amigos, que leram com paciência as primeiras versões do texto, mando também abraços de gratidão a Michael Levine, Marzia Niccolai, Ade Oshineye e Katrina Sostek. E um obrigado a meus colegas na Google, por também ajudarem na revisão. Sem todo esse esforço de vocês este livro seria incompreensível.

Obrigado a todos que estiveram de alguma forma envolvidos em tornar este livro uma realidade. À minha editora, Trina MacDonald, por iniciar o processo e pelo apoio durante todo o seu desenvolvimento. À equipe editorial, crucial para que tudo tenha chegado a bom termo, formada pelos editores de desenvolvimento Tom Cirtin e Chris Zahn, a assistente editorial Olivia Basegio, a gerente de marketing Stephane Nakib, a copidesque Stephanie Geels e a editora de produção, Julie Nahil.

Agradecimentos especiais devem ser dados a todos os maravilhosos programadores Python que conheci ou com quem trabalhei: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein e Ka-Ping Yee. Obrigado por sua orientação e liderança. A comunidade formada ao redor do Python é da mais alta excelência e sinto-me honrado em fazer parte dela.

Obrigado a meus colegas de equipe ao longo dos anos, por permitirem que eu

fosse o pior músico da banda. Obrigado a Kevin Gibbs, por me ajudar a correr riscos. Obrigado a Ashcraft, Ryan Barrett e Jon McAlister, por me mostrarem como é que se faz, e a Brad Fitzpatrick, por sugerir o conceito de “elevar a um novo patamar”. Obrigado a Paul McDonald, por cofundar nosso projeto maluco, e a Jeremy Ginsberg e Jack Hebert, por torná-lo uma realidade.

Sou também grato aos excepcionais professores de programação que eu tive: Ben Chelf, Vince Hugo, Russ Lewin, Jon Stemmle, Derek Thomson e Daniel Wang. Sem suas aulas jamais teria tido sucesso em nossa profissão e muito menos a perspectiva necessária para ensinar a outros.

Obrigado à minha mãe, por ter me ensinado o senso de propósito e encorajado a me tornar um programador, a meu irmão, meus avós, demais familiares e amigos de infância por serem modelos de vida à medida que eu crescia e encontrava minha paixão.

Por fim, obrigado à minha esposa, Colleen, por seu amor, apoio e risadas pela jornada da vida.

Sobre o autor

Brett Slatkin é um engenheiro sênior de software na Google. É chefe de engenharia e cofundador dos Google Consumer Surveys. Anteriormente, trabalhou na infraestrutura em Python do Google App Engine e, há nove anos, ainda inexperiente, empregou Python na administração do imenso parque de servidores da Google. Slatkin também usou Python para implementar o sistema Google que gerencia o PubSubHubbub, um protocolo que ele cocriou.

Além de seu trabalho diurno, contribui com inúmeros projetos open source e escreve sobre software, ciclismo e outros tópicos em seu site pessoal (*http://onebigfluke.com*). Slatkin é graduado em Ciência da Computação na Columbia University, na cidade de Nova York, e hoje vive em San Francisco.

CAPÍTULO 1

Raciocínio pythônico

As expressões idiomáticas de uma linguagem de programação são definidas por seus usuários. Ao longo dos anos, a comunidade Python concebeu e vem usando o adjetivo *pythônico* (em inglês, *Pythonic*) para descrever qualquer código que siga um estilo em particular. O estilo pythônico não segue um regimento obrigatório nem é ditado pelo compilador. Ele foi destilado no decorrer do tempo pelo uso que os programadores têm feito da linguagem e pela necessidade de trabalho colaborativo com outras pessoas. Os desenvolvedores em Python preferem ser explícitos, escolhem o simples em vez do complicado e maximizam a legibilidade (digite import this).

Programadores familiarizados com outras linguagens podem tentar escrever código em Python como se fosse em C++, Java ou o que quer que conheçam melhor. Os novatos talvez ainda estejam tentando se encontrar em meio à vasta gama de conceitos expressáveis em Python. É importante que todos saibam a melhor maneira – a *pythônica* – de fazer a maioria das coisas em Python. Esses padrões afetarão todo e qualquer programa que você escreva.

Item 1: Saiba qual versão de Python está em uso

Ao longo deste livro, a maioria dos exemplos está na sintaxe do Python 3.4 (lançado em 17 de março de 2014). Este livro também mostra alguns exemplos na sintaxe antiga do Python 2.7 (lançado em 3 de julho de 2010) para mostrar as diferenças mais importantes. A maioria das sugestões vale também para runtimes populares de Python: CPython, Jython, IronPython, PyPy etc.

Alguns computadores vêm com mais de uma versão instalada do CPython padrão. Entretanto, o significado-padrão com comando python no terminal pode não ser muito claro. python é normalmente um alias para python2.7, mas pode ser também um atalho para versões mais antigas como python2.6 ou python2.5. Para saber qual a versão exata do Python em uso, usa-se a flag --version.

```
$ python --version
Python 2.7.8
```

O Python 3 está disponível, normalmente, sob o nome python3.

```
$ python3 --version
Python 3.4.2
```

Podemos também verificar a versão de Python em uso durante a execução do programa, inspecionando os valores no módulo nativo sys.

```
import sys
print(sys.version_info)
print(sys.version)
>>>
sys.version_info(major=3, minor=4, micro=2,
➥releaselevel='final', serial=0)
3.4.2 (default, Oct 19 2014, 17:52:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)]
```

Tanto o Python 2 como o Python 3 são ativamente mantidos pela comunidade Python. O desenvolvimento no Python 2 está congelado, só sendo permitidas correções de erros, falhas de segurança e backports para facilitar a transição do Python 2 para o Python 3. Existem ferramentas úteis, como 2to3 e six, que tornam a adoção do Python 3 ainda mais fácil.

O Python 3 está constantemente adquirindo novos recursos e aprimoramentos que jamais serão adicionados ao Python 2. Quando este livro estava em produção, a maioria das bibliotecas open source mais comuns do Python já estava disponível para o Python 3. Eu recomendo fortemente que se use Python 3 para qualquer projeto de agora em diante.

Lembre-se

- Há duas versões de Python ainda em uso: Python 2 e Python 3.
- Existem inúmeros runtimes populares para o Python: CPython, Jython, IronPython, PyPy etc.
- Assegure-se de que a linha de comando usada para rodar programas em Python no seu sistema é a versão que você espera que seja.

- Prefira sempre Python 3 para quaisquer projetos de hoje em diante, pois ele é o objetivo principal da comunidade Python.

Item 2: Siga o Guia de Estilo PEP 8

A Python Enhancement Proposal #8 (Proposta de Aprimoramento do Python número 8), conhecida simplesmente como PEP 8, é o Guia de Estilo para formatar código em Python. Você pode escrever código em Python da maneira que bem entender, desde que a sintaxe seja válida. Contudo, empregar um estilo consistente deixa seu código mais fácil de ler e distribuir. Compartilhar um estilo comum com outros programadores de Python na comunidade facilita a colaboração em projetos. Mesmo que você seja o único a ler seu código por séculos e séculos, seguir o guia de estilo fará com que qualquer modificação posterior seja bem mais fácil de fazer.

A PEP 8 é bastante detalhada no tocante a como escrever código legível em Python. Ela é continuamente atualizada à medida que a linguagem Python evolui. Vale a pena ler o guia completo, disponível online (*http://www.python.org/dev/peps/pep-0008/*). Algumas das regras que você deve sempre seguir são:

Espaços em branco: No Python, cada espaço em branco tem um significado sintático. Os programadores em Python são especialmente sensíveis aos efeitos dos espaços em branco na legibilidade do código.

- Use espaços em vez de tabulações para indentação.
- Use quatro espaços para cada nível de indentação sintaticamente significativa.
- As linhas devem ter 79 caracteres de comprimento, ou menos.
- As continuações de expressões longas em linhas adicionais devem ser indentadas com quatro espaços adicionais além de ser nível normal de indentação.
- Em um arquivo, as funções e classes devem ser separadas por duas linhas em branco.
- Em uma classe, os métodos devem ser separados por uma linha em branco.
- Não coloque espaços em volta de índices de listas, chamadas de funções ou atribuições de palavra-chave como argumento.

- Coloque um – e apenas um – espaço antes e depois de uma atribuição de variável.

Nomes: a PEP 8 sugere um estilo único para nomear as diferentes partes da linguagem. Isso torna muito fácil distinguir quais tipos correspondem a quais nomes ao ler o código.

- Funções, variáveis e atributos devem estar no formato caixa_baixa_underscore[1].
- Atributos protegidos de instâncias devem estar no formato _começa_com_um_underscore.
- Atributos privativos de instância devem estar no formato __começa_com_dois_underscores.
- Classes e exceções devem estar no formato PalavrasCapitalizadas (CamelCase[2]).
- Constantes dentro de módulos devem estar sempre no formato TUDO_EM_MAIÚSCULAS.
- Métodos de instância em classes devem usar self como o nome do primeiro parâmetro (referindo-se ao objeto).
- Métodos de classe devem usar cls como o nome do primeiro parâmetro (referindo-se à classe).

Expressões e comandos[3]: o texto *The Zen of Python* diz: “Deve existir um – e preferencialmente apenas um – modo óbvio de se fazer algo”. A PEP 8 tenta disciplinar esse estilo oferecendo uma orientação para expressões e comandos.

- Use negação em linha (if a is not b) em vez de negar uma expressão positiva (if not a is b).
- Não faça a verificação de valores vazios (como [] ou '') consultando seu comprimento (if len(somelist) == 0). Em vez disso use if not somelist e considere que os valores vazios serão implicitamente avaliados como False.
- A mesma coisa vale para valores não vazios (como [1] ou 'hi'). O comando if somelist é implicitamente True para valores não vazios.
- Evite comandos if, laços de repetição for e while e comandos compostos com except em uma única linha. Espalhe-os em múltiplas linhas para maior

clareza.

- Sempre coloque os comandos import no início do arquivo.
- Sempre use nomes absolutos para módulos quando os importar, e nunca o nome relativo ao caminho do módulo corrente. Por exemplo, para importar o módulo foo presente no pacote bar, use from bar import foo, jamais import foo, apenas.
- Mesmo que seja obrigatório, por algum motivo, importar módulos com caminhos relativos, use sempre a sintaxe explícita from . import foo.
- Os módulos importados devem estar em seções na seguinte ordem: módulos da biblioteca nativa, módulos de terceiros, seus próprios módulos. Os módulos em cada subseção devem ser importados em ordem alfabética.

Nota

A ferramenta de análise estática Pylint (*http://www.pylint.org/*) é bastante popular para verificar código-fonte em Python. O Pylint obriga-nos a usar o Guia de Estilo PEP 8 e detecta muitos outros tipos de erros comuns em programas Python.

Lembre-se

- Sempre obedeça ao Guia de Estilo PEP 8 quando escrever código em Python.
- Empregar o mesmo estilo que a grande comunidade de desenvolvedores Python usa facilita a colaboração com outros programadores.
- O uso de um estilo consistente facilita modificações futuras em seu próprio código.

Item 3: Saiba as diferenças entre bytes, str e unicode

No Python 3, existem dois tipos que representam caracteres: bytes e str. Instâncias de bytes contêm valores primários de 8 bits. Instâncias de str contêm caracteres Unicode.

No Python 2, existem dois tipos que representam sequências de caracteres: str e unicode. Em contraste com o Python 3, instâncias de str contêm valores primários de 8 bits. Instâncias de unicode contêm caracteres Unicode.

Existem muitas maneiras de representar caracteres Unicode como dados

puramente binários (valores primários de 8 bits). A codificação mais comum é *UTF-8*. É importante observar que as instâncias str em Python 3 e unicode em Python 2 não têm uma codificação binária associada. Para converter caracteres Unicode para dados binários, é preciso usar o método encode. Para converter dados binários para caracteres Unicode, é preciso usar o método decode.

Ao escrever programas em Python, é importante codificar e decodificar o Unicode no limite mais longínquo de suas interfaces. O núcleo de seu programa deve usar apenas caracteres Unicode (str no Python 3, unicode no Python 2) e não devem fazer nenhum julgamento quanto à codificação deles. Essa regrinha permite que o programa possa ser bastante assertivo a respeito de codificações alternativas de texto (como *Latin-1*, *Shift JIS* e *Big5*) enquanto mantém-se bastante rigoroso a respeito da codificação do texto de saída (idealmente, UTF-8).

A confusão causada pelos tipos de caractere causa duas situações muito comuns quando se codifica em Python:

- Queremos operar com valores brutos de 8 bits codificados em UTF-8 (ou qualquer outra codificação).
- Queremos operar com caracteres Unicode que não estão codificados em nenhum padrão específico.

Serão necessárias duas funções auxiliares para converter entre esses dois casos e assegurar que o tipo do valor de entrada é o que o código está esperando que seja.

No Python 3, precisaremos de um método que aceite um str ou bytes e sempre retorne str.

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instância de str
```

Será necessário outro método que aceite valores str ou bytes e sempre retorne bytes.

```
def to_bytes(bytes_or_str):
```

```
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instância de bytes
```

No Python 2, é necessário um método que aceite um valor str ou unicode e sempre retorne unicode.

```
# Python 2
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value  # Instância de unicode
```

Será necessário outro método que aceite str ou unicode e sempre retorne um str.

```
# Python 2
def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value  # Instância de str
```

Porém, há duas grandes pegadinhas quando se lida com valores puros em 8 bits e caracteres Unicode no Python.

O primeiro problema ocorre porque, no Python 2, as instâncias de unicode e str parecem ser do mesmo tipo quando, na realidade, str contém apenas caracteres ASCII de 7 bits. Entretanto:

- é possível combinar instâncias str e unicode usando o operador + ;
- é possível comparar instâncias str e unicode usando operadores de igualdade e desigualdade;
- é possível usar instâncias unicode em strings formatadas como %s'.

Todo esse comportamento indica que é possível passar uma instância str ou

unicode a uma função que espera apenas uma delas e tudo vai funcionar – desde que estejamos lidando apenas com caracteres ASCII de 7 bits. No Python 3, as instâncias bytes e str jamais serão equivalentes – nem mesmo a string vazia – portanto, é preciso ser cauteloso a respeito dos tipos de caracteres e strings sendo passados adiante.

O primeiro problema é que, em Python 3, as operações envolvendo “handlers”, os manipuladores de arquivos (devolvidos pela função nativa open), sempre esperam codificação UTF-8 por default. No Python 2, o default para as operações de arquivo é sempre o tipo binário. Isso causa falhas inesperadas, especialmente para os programadores acostumados com o Python 2.

Por exemplo, imaginemos que se queira escrever algum dado aleatório, de formato binário, em um arquivo. Em Python 2, o código a seguir funciona. Em Python 3, causa erro.

```
with open('/tmp/random.bin', 'w') as f:
    f.write(os.urandom(10))
>>>
TypeError: must be str, not bytes
```

A causa dessa exceção é o novo argumento encoding para a função open, adicionado no Python 3. Esse parâmetro tem como valor default 'utf-8'. Com isso, as operações read e write em manipuladores de arquivos esperam receber instâncias str contendo caracteres Unicode em vez de instâncias bytes contendo dados binários.

Para que isso funcione de forma apropriada, é preciso indicar que o arquivo será aberto em modo de escrita binária ('wb') em vez do modo de escrita de texto ('w'). O trecho de código a seguir emprega a função open de uma maneira que funciona tanto em Python 2 como em Python 3:

```
with open('/tmp/random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

Esse problema também existe para a leitura de dados em arquivos. A solução é a mesma: deixar expressa a indicação de leitura binária 'rb' em vez de 'r' ao abrir o arquivo.

Lembre-se

- No Python 3, bytes contêm sequências de valores puros em 8 bits e str contém sequências de caracteres Unicode. Instâncias bytes e str não podem ser usadas juntas com operadores (como > ou +).
- No Python 2, str contém sequências de valores de 8 bits e unicode contém sequências de caracteres Unicode. str e unicode *podem* ser usados juntos com operadores desde que str contenha apenas caracteres ASCII de 7 bits.
- Use funções auxiliares para assegurar que as entradas que estão sendo manipuladas são do tipo de caractere que o código está esperando (valores de 8 bits, caracteres codificados em UTF-8, caracteres Unicode etc.).
- Se desejar ler ou escrever dados binários em um arquivo, sempre o abra usando o modo binário (como 'rb' ou 'wb').

Item 4: Escreva funções auxiliares em vez de expressões complexas

A sintaxe concisa do Python torna fácil escrever em uma única linha uma expressão que implemente grande quantidade de lógica. Por exemplo, digamos que seja necessário decodificar a query string de um URL. Aqui, cada parâmetro da query string representa um valor inteiro:

```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))
>>>
{'red': ['5'], 'green': [''], 'blue': ['0']}
```

Alguns parâmetros de query string podem trazer múltiplos valores, outros apenas um único valor, alguns podem estar presentes, mas com o valor em branco e outros podem ter sido simplesmente suprimidos. Ao usar o método get no dicionário resultante, os valores devolvidos serão diferentes em cada circunstância.

```
print('Red:     ', my_values.get('red'))
print('Green:   ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))
```

```
>>>
Red:      ['5']
Green:    ['']
Opacity:  None
```

Seria ótimo se um valor default 0 fosse atribuído quando um parâmetro não for atribuído ou for suprimido. Alguns programadores usariam expressões booleanas porque a simplicidade da lógica requerida não parece ainda necessitar de todo um comando if ou função auxiliar.

A sintaxe do Python torna essa escolha fácil demais. O truque aqui é que strings vazias, listas vazias ou zeros são considerados implicitamente como False em termos booleanos. Portanto, as expressões a seguir vão utilizar a subexpressão após o operador or caso a primeira subexpressão seja False.

```
# Para a query string 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print('Red:     %r' % red)
print('Green:   %r' % green)
print('Opacity: %r' % opacity)
>>>
Red:     '5'
Green:   0
Opacity: 0
```

O caso red funciona porque a chave está presente no dicionário my_values. O valor é uma lista com um único número: a string '5'. Essa string implicitamente devolve True, portanto o valor red é atribuído à primeira parte da expressão or.

O caso green funciona porque o valor no dicionário my_values é uma lista com um único membro: uma string vazia. Como sabemos, uma string vazia implicitamente devolve o valor booleano False, fazendo com que a expressão or atribua o valor 0.

O caso opacity funciona porque não existe um valor no dicionário my_values. O comportamento do método get faz com que devolva seu segundo argumento se a chave não existir no dicionário. O valor default neste caso é uma lista com um

único membro, uma string vazia. Caso opacity não seja encontrado no dicionário, este código age exatamente como no caso green.

Entretanto, essa expressão é difícil de ler e ainda não faz tudo o que é necessário. Além disso, queremos assegurar que todos os valores dos parâmetros sejam inteiros para que possamos usá-los em expressões matemáticas. Para isso, é preciso envolver cada expressão em uma função int, nativa do Python, para converter a string em inteiro.

```
red = int(my_values.get('red', [''])[0] or 0)
```

Se já era um tanto difícil de ler, com o int a coisa piorou bastante. Agora, está extremamente difícil de entender. O ruído visual é considerável. O código não é mais acessível. Uma pessoa não familiarizada com o código terá que perder bastante tempo dividindo a expressão em pedacinhos menores para entender o que ela realmente faz. Mesmo sendo uma boa ideia manter o código o mais curto possível, não vale a pena incluir tudo isso em uma única linha.

O Python 2.5 introduziu expressões condicionais if/else – chamadas de ternárias – para tornar casos como esses mais claros e ainda assim manter o código enxuto.

```
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0
```

Muito melhor! Para situações menos complicadas, as expressões condicionais if/else podem tornar o código mais claro. No entanto, o exemplo anterior ainda não é mais claro do que se usássemos comandos if/else em múltiplas linhas. Comparar a lógica espalhada dessa forma com a versão condensada faz com que a resumida pareça ainda mais complicada.

```
green = my_values.get('green', [''])
if green[0]:
    green = int(green[0])
else:
    green = 0
```

Uma função auxiliar é sempre preferível, especialmente se essa lógica deve ser usada repetidamente:

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
```

```
    if found[0]:
        found = int(found[0])
    else:
        found = default
    return found
```

O código que chama a função é muito mais limpo que a expressão complexa usando or ou que a versão em duas linhas usando if/else.

```
green = get_first_int(my_values, 'green')
```

No exato momento em que suas expressões se tornem complicadas, é hora de considerar dividi-las em partes menores e mover a lógica para funções auxiliares. O que se ganha em legibilidade sempre supera qualquer enxugamento de código que se possa conseguir. Não deixe que a sintaxe concisa do Python em expressões complexas coloque-o em uma situação desagradável como essa.

Lembre-se

- A sintaxe do Python facilita a criação de expressões de uma única linha que são incrivelmente complexas e difíceis de ler.
- Transfira expressões complexas para uma função auxiliar, especialmente se for necessário usar a mesma lógica repetidamente.
- A expressão if/else oferece uma alternativa muito mais legível que o emprego de operadores booleanos como or e and em expressões.

Item 5: Saiba como fatiar sequências

O Python tem uma sintaxe específica para dividir sequências em pedaços menores. O fatiamento (slicing) permite acessar um subconjunto dos itens de uma sequência com muito menos esforço. Os candidatos mais simples para serem alvo dessa técnica são os valores do tipo list, str e bytes. Podemos dividir sequências também em qualquer classe de Python que implemente os métodos especiais __getitem__ e __setitem__ (consulte o Item 28: “Herde da classe collections.abc para obter tipos de contêiner personalizados”).

O formato básico da sintaxe de fatiamento é somelist[start:end], no qual start é inclusivo e end é excludente.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('First four:', a[:4])
print('Last four: ', a[-4:])
print('Middle two:', a[3:-3])
>>>
First four: ['a', 'b', 'c', 'd']
Last four:  ['e', 'f', 'g', 'h']
Middle two: ['d', 'e']
```

Ao fatiar uma lista incluindo na fatia seu início, você deve suprimir o índice zero para reduzir a poluição visual.

```
assert a[:5] == a[0:5]
```

Quando a fatia compreender um ponto intermediário até o final da lista, é necessário suprimir o índice final, por ser redundante.

```
assert a[5:] == a[5:len(a)]
```

Índices negativos de fatiamento são úteis para calcular deslocamentos relativos ao fim da lista. Todas essas formas de fatiamento são claras o bastante para que qualquer um que esteja lendo o código possa entender num relance. Não há surpresas e eu gostaria de recomendar ao leitor que sempre use essas variantes.

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #                  ['e', 'f', 'g', 'h']
a[-3:]    #                       ['f', 'g', 'h']
a[2:5]    #          ['c', 'd', 'e']
a[2:-1]   #          ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #                       ['f', 'g']
```

O fatiamento lida de forma apropriada com índices start e end que estejam além dos limites da lista, o que torna fácil estabelecer, pelo código, um tamanho máximo para uma sequência de entrada.

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:]
```

Em contraste, ao acessar o mesmo índice diretamente causa uma exceção.

```
a[20]
>>>
IndexError: list index out of range
```

Nota

Tenha em mente que usar em uma lista índices negativos obtidos de uma variável é uma das poucas situações em que podemos encontrar resultados imprevisíveis na operação de fatiamento. Por exemplo, a expressão `somelist[-n:]` funcionará bem se `n` for maior que 1 (por exemplo, `somelist[-3:]`). Entretanto, se `n` for zero, a expressão `somelist[-0:]` resultará em uma cópia da lista original.

O resultado do fatiamento de uma lista é sempre uma lista nova. Nenhuma referência aos itens da lista original é mantida. Modificar o resultado da nova lista (que é uma fatia da original) não modificará a lista original.

```
b = a[4:]
print('Before:   ', b)
b[1] = 99
print('After:    ', b)
print('No change:', a)
>>>
Before:    ['e', 'f', 'g', 'h']
After:     ['e', 99, 'g', 'h']
No change: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Quando usado em atribuições, as fatias substituem a porção especificada da lista original. Ao contrário das atribuições de tuplas (como a, b = c[:2]), o tamanho da fatia sendo atribuída não precisa ser o mesmo. Os valores que estiverem posicionados antes e depois da fatia serão preservados. A lista crescerá ou encolherá para acomodar os novos valores.

```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After  ', a)
>>>
Before  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After   ['a', 'b', 99, 22, 14, 'h']
```

Se não mencionarmos nem o índice de início (start) nem o final (end) quando fizermos uma atribuição de um fatiamento a uma variável, o resultado armazenado na variável será uma cópia em separado da lista original.

```
b = a[:]
assert b == a and b is not a
```

Por outro lado, ao atribuirmos uma fatia a uma variável sem usar índices de início e fim, o conteúdo da variável será substituído por uma referência à lista original (em vez de criar uma nova lista, ou seja, é a mesma lista, mas que pode ser chamada por dois nomes, tanto ‘a’ como ‘b’.).

```
b = a
print('Before', a)
a[:] = [101, 102, 103]
assert a is b          # a e b são dois nomes diferentes para a mesma lista
print('After ', a)     # A lista a teve seu conteúdo alterado
>>>
Before ['a', 'b', 99, 22, 14, 'h']
After  [101, 102, 103]
```

Lembre-se

- Evite ser prolixo: não digite 0 para o índice inicial (start) ou o comprimento da lista para o índice final (end).
- O fatiamento sabe como lidar com índices start ou end que extrapolaram os limites da lista existente, tornando bem fácil expressar fatias limitadas pelo início ou pelo fim da sequência (como a[:20] ou a[-20:]).
- Atribuir uma lista a uma fatia de outra lista substitui a fatia original pela lista completa que está sendo atribuída, mesmo que seus tamanhos sejam diferentes.

Item 6: Evite usar start, end e stride em uma mesma fatia

Além do fatiamento básico (consulte o Item 5: “Saiba como fatiar sequências”), o Python possui uma sintaxe especial para definir o salto (stride) na seleção dos

itens de uma fatia, na forma somelist[start:end:stride]. Isso permite que se possa selecionar os itens "*pulando de n em n*" no processo de fatiamento da sequência. Por exemplo, o parâmetro stride torna fácil agrupar em listas menores os itens de uma lista com índices pares e ímpares.

```
a = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = a[::2]
evens = a[1::2]
print(odds)
print(evens)
>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

O problema dessa técnica é que a sintaxe do stride às vezes se comporta de maneira inesperada, o que pode introduzir bugs. Por exemplo, um truque comum em Python para inverter a ordem em uma string do tipo byte é fatiá-la com um stride de -1.

```
x = b'mongoose'
y = x[::-1]
print(y)
>>>
b'esoognom'
```

Funciona muito bem para strings do tipo byte contendo caracteres ASCII puros, mas gera um erro em strings tipo byte contendo caracteres Unicode codificados em UTF-8.

```
w = "
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')
>>>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9d in
➥position 0: invalid start byte
```

Outra questão: há utilidade para strides negativos diferentes de -1? Considere os

seguintes exemplos.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[::2]   # ['a', 'c', 'e', 'g']
a[::-2]  # ['h', 'f', 'd', 'b']
```

Aqui, ::2 significa selecionar o primeiro item, e a partir daí o “segundo depois do próximo” (ou seja, “pula um”). O ::-2 , como se poderia imaginar, tem o mesmo comportamento mas começa do fim em direção ao início.

Mas o que significa 2::2? E qual a diferença entre -2::-2 , -2:2:-2 e 2:2:-2?

```
a[2::2]     # ['c', 'e', 'g']
a[-2::-2]   # ['g', 'e', 'c', 'a']
a[-2:2:-2]  # ['g', 'e']
a[2:2:-2]   # []
```

O problema é que a parte do stride na sintaxe de fatiamento é um tanto confusa. Já é difícil o bastante ler três números dentro dos colchetes por conta da densidade, e fica menos óbvio entender os índices start e end: se é antes ou depois de considerar o valor do stride. E se o stride for negativo, isso muda?

Para evitar esses problemas de legibilidade e entendimento, evite usar o stride em conjunto com os índices start e end. Se for necessário usar um stride, prefira valores positivos e omita os índices start e end. Se for absolutamente necessário combinar stride com índices start ou end, considere fazê-lo em duas atribuições separadas, usando uma variável a mais: uma atribuição para o stride, outra para realmente fazer o fatiamento.

```
b = a[::2]   # ['a', 'c', 'e', 'g']
c = b[1:-1]  # ['c', 'e']
```

Esse procedimento tem a desvantagem de criar uma cópia adicional dos dados (lembre-se: o fatiamento cria uma lista nova). A primeira operação deve tentar reduzir o tamanho do slice ao máximo. Se o seu programa por alguma razão não tiver condições de arcar com o uso adicional de memória necessário para esses dois passos, experimente o método islice, presente no módulo nativo itertools (consulte o Item 46: “Use algoritmos e estruturas de dados nativos”), que não permitem valores negativos para start, end ou stride.

Lembre-se

- Especificar simultaneamente start, end e stride em um slice pode ser extremamente confuso.
- Prefira usar valores positivos para o stride em fatias sem índices start ou end. Evite valores negativos de stride sempre que possível.
- Evite usar start, end e stride simultaneamente em uma mesma fatia. Se for realmente necessário usar todos os três parâmetros, prefira usar uma variável intermediária e fazer duas atribuições (uma para fatiar, outra para o stride) ou então use o método islice disponível no módulo nativo itertools.

Item 7: Use abrangências de lista em vez de map e filter

O Python oferece uma sintaxe compacta para derivar uma lista de outra. Essas expressões são chamadas de *abrangências de lista* ou *list comprehensions* ou, simplesmente, *listcomps*[4]. Por exemplo, digamos que se queira computar o quadrado de cada número em uma lista. Isso pode ser feito criando um laço (loop) na sequência de origem precedido pela expressão de cálculo.|

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x**2 for x in a]
print(squares)
```

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A não ser que a função sendo aplicada tenha apenas um argumento, para os casos mais simples as abrangências de listas são mais fáceis de ler que a função nativa map. O map precisa de uma função lambda para computar o resultado, e esta é visualmente confusa.

```
squares = map(lambda x: x ** 2, a)
```

Ao contrário do map, as abrangências de listas permitem filtrar itens com facilidade a partir da lista de entrada, removendo os resultados correspondentes na saída. Por exemplo, considere que precisamos calcular apenas os quadrados dos números divisíveis por 2. Nas abrangências de lista, basta adicionar uma expressão condicional (um simples if) depois do laço:

```
even_squares = [x**2 for x in a if x % 2 == 0]
```

```
print(even_squares)
>>>
[4, 16, 36, 64, 100]
```

A função nativa filter pode ser usada em conjunto com o map para obter o mesmo resultado, mas é muito mais indigesta:

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Os dicionários e conjuntos possuem seu próprio equivalente das abrangências de lista, o que torna muito fácil criar estruturas de dados derivadas em nossos algoritmos.

```
chile_ranks = {'ghost': 1, 'habanero': 2, 'cayenne': 3}
rank_dict = {rank: name for name, rank in chile_ranks.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rank_dict)
print(chile_len_set)
>>>
{1: 'ghost', 2: 'habanero', 3: 'cayenne'}
{8, 5, 7}
```

Lembre-se

- As abrangências de lista são mais fáceis de ler que as funções nativas map e filter porque não precisam de expressões lambda auxiliares.
- As abrangências de lista permitem saltar itens da lista de entrada facilmente, um comportamento que o map não suporta sem a ajuda da função filter.
- Os dicionários e conjuntos também suportam expressões abrangentes.

Item 8: Evite mais de duas expressões em abrangências de lista

Para além do básico (consulte o Item 7: “Use abrangências de lista em vez de map e filter”), as abrangências de lista também suportam laços em níveis múltiplos. Por exemplo, digamos que se queira simplificar uma matriz (uma lista

contendo outras listas) em uma lista unidimensional com todas as células. É possível fazê-lo com abrangências de lista, bastando incluir duas expressões for. Essas expressões rodam na ordem em que aparecem, da esquerda para a direita:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

```
>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

O exemplo acima é simples, legível e um uso razoável de laços múltiplos. Outro uso razoável de laços aninhados é replicar o leiaute em dois níveis da lista de entrada. Por exemplo, digamos que se queira elevar ao quadrado o valor de cada célula em uma matriz bidimensional. Essa expressão é menos legível por causa dos caracteres [] extras, mas ainda assim é fácil de ler.

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

Se a expressão incluir outro laço, a abrangência de lista seria tão grande que seria preciso dividi-la em mais de uma linha.

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    # ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

Neste ponto, as abrangências possuem várias linhas e, portanto, não são mais curtas que as alternativas. Entretanto, consigo produzir o mesmo resultado usando comandos de laço comuns. A indentação dessa versão faz com que os laços sejam mais claros que nas abrangências de lista.

```
flat = []
for sublist1 in my_lists:
```

```
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

As abrangências de lista também suportam mais de uma condição if. Comandos if múltiplos em um mesmo nível de laço são equivalentes a um único if com múltiplas condições ligadas logicamente por uma operação and implícita. Por exemplo, digamos que se queira filtrar uma lista de números para obter apenas os valores pares maiores que quatro. As duas abrangências de lista a seguir são equivalentes.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

As condições podem ser especificadas a cada nível de laço depois da expressão for. Por exemplo, imagine que se queira filtrar uma matriz de forma que as únicas células remanescentes sejam as divisíveis por 3 em linhas cuja soma seja 10 ou mais. Usar abrangências de lista para isso fica até curto, mas é extremamente difícil de ler.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)
```

```
>>>
[[6], [9]]
```

Embora esse exemplo seja um tanto rebuscado, na prática veremos situações em que essas expressões parecem ser apropriadas. Eu aconselho jamais usar abrangências de lista assim complexas. O código resultante é dificílimo de compreender, especialmente por outras pessoas. O que se economiza em número de linhas não justifica a dificuldade posterior.

Como regra, evite usar mais que duas expressões em uma abrangência de lista, podendo ser duas condições, dois laços ou uma condição e um laço. No momento em que a coisa fique mais complicada que isso, é melhor usar comandos if e for normais e escrever uma função auxiliar (consulte o Item 16: “Prefira geradores em vez de retornar listas”).

Lembre-se

- As abrangências de lista suportam múltiplos níveis de laço e múltiplas condições em cada nível.
- Abrangências de lista com mais de duas expressões são muito difíceis de ler e devem ser evitadas.

Item 9: Considere usar expressões geradoras em abrangências muito grandes

O problema com as abrangências de lista (consulte o Item 7: “Use abrangências de lista em vez de map e filter”) é que podem criar uma nova lista contendo um item para cada valor na sequência de entrada. Para sequências de entrada pequenas isso passa despercebido, mas sequências grandes podem consumir quantidades significativas de memória, causando uma falha grave que interrompe o programa.

Por exemplo, digamos que se queira ler um arquivo e retornar o número de caracteres em cada linha. Com uma abrangência de lista precisaríamos guardar na memória cada linha presente no arquivo. Se este for absurdamente gigantesco, ou se for (por exemplo) um socket de rede que é, por natureza, de tamanho infinito (o fim do arquivo nunca chega), teremos um grande problema se estivermos usando abrangências de lista. No exemplo a seguir, a abrangência de lista usada consegue lidar apenas com valores de entrada pequenos.

```
value = [len(x) for x in open('/tmp/my_file.txt')]
print(value)
>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

A solução, em Python, é empregar as chamadas *expressões geradoras*, uma generalização das abrangências de listas e dos geradores. As expressões geradoras não materializam a sequência de saída completa quando são executadas. Em vez disso, as expressões geradoras fazem uso de um elemento de iteração que captura um item por vez na expressão.

Uma expressão geradora pode ser criada colocando entre parênteses ( ) uma sintaxe parecida com a das abrangências de lista. No exemplo a seguir temos

uma expressão geradora equivalente ao código anterior. Entretanto, a expressão geradora imediatamente entrega o controle ao iterador e coloca o laço em espera.

```
it = (len(x) for x in open('/tmp/my_file.txt'))
print(it)
>>>
<generator object <genexpr> at 0x101b81480>
```

O iterador pode ser avançado um passo por vez para produzir o próximo resultado da expressão geradora quando necessário (usando a função nativa next). O código pode consumir o resultado da expressão geradora sempre que necessário sem esgotar a memória.

```
print(next(it))
print(next(it))
>>>
100
57
```

Outro ponto forte das expressões geradoras é que elas podem ser compostas. No exemplo a seguir, tomamos o iterador devolvido pela expressão geradora acima e o usamos como entrada para outra expressão geradora.

```
roots = ((x, x**0.5) for x in it)
```

Cada vez que o iterador avança, o iterador interno também avança, criando um efeito dominó no laço, testando as expressões condicionais e repassando entradas e saídas.

```
print(next(roots))
>>>
(15, 3.872983346207417)
```

Geradores aninhados como este executam de forma veloz no Python. Se estiver procurando uma maneira de criar alguma funcionalidade que precise manipular um fluxo muito intenso de dados de entrada, as expressões geradoras são a melhor ferramenta. A única pegadinha é que os iteradores devolvidos pelas expressões geradoras guardam seus estados (stateful), portanto é preciso ter cuidado para não usá-los mais de uma vez (consulte o Item 17: “Seja conservador quando iterar sobre argumentos”).

Lembre-se

- As abrangências de lista podem causar problemas quando o fluxo de dados na entrada for muito intenso, consumindo muita memória.
- As expressões geradoras evitam os problemas de memória porque devolve um iterador que produz apenas um valor por vez.
- As expressões geradoras podem ser compostas, bastando passar o iterador de uma expressão geradora para a subexpressão for de outra.
- Quando aninhadas juntas, duas ou mais expressões geradoras executam de forma veloz no Python.

Item 10: Prefira enumerate em vez de range

A função nativa range é útil para laços em iteração sobre um conjunto de inteiros.

```
random_bits = 0
for i in range(64):
    if randint(0, 1):
        random_bits |= 1 << i
```

Em uma estrutura de dados que se queira varrer iterativamente, como uma lista de strings, é possível usar um laço para varrer a sequência diretamente.

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print('%s is delicious' % flavor)
```

Muitas vezes, é necessário iterar sobre uma lista e também obter o índice do item atual. Por exemplo, imagine que se queira mostrar o ranking dos sabores favoritos de sorvete. Uma das maneiras de fazê-lo é com range.

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print('%d: %s' % (i + 1, flavor))
```

O código é bastante desajeitado, se comparado aos outros exemplos de iteração com flavor_list ou range. É preciso obter o tamanho da lista. É preciso considerar os índices do array. É difícil de ler.

O Python oferece a função nativa enumerate para esses casos. enumerate envolve qualquer iterador com um gerador simplificado. Esse gerador produz um par de valores, o índice do laço e o próximo valor do iterador. O código resultante é muito mais claro.

```
for i, flavor in enumerate(flavor_list):
    print('%d: %s' % (i + 1, flavor))
```

```
>>>
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

O código pode ficar ainda mais curto se for especificado o número pelo qual o enumerate deve iniciar a contagem (neste caso, 1).

```
for i, flavor in enumerate(flavor_list, 1):
    print('%d: %s' % (i, flavor))
```

Lembre-se

- enumerate oferece uma sintaxe concisa para laços em iteradores e para obter o índice de cada item do iterador durante o andamento da varredura.
- Prefira enumerate em vez de criar um laço baseado em range e índices de sequência.
- É possível fornecer um segundo parâmetro para o enumerate para especificar o número pelo qual a contagem será iniciada (o default é zero).

Item 11: Use zip para processar iteradores em paralelo

Muitas vezes em Python nos deparamos com múltiplas listas de objetos relacionados. As abrangências de lista facilitam tomar uma lista de origem e obter uma lista derivada aplicando uma expressão a ela (consulte o Item 7: “Use abrangências de lista em vez de map e filter”).

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]
```

Os itens na lista derivada estão relacionados aos itens da lista de origem pelos seus índices. Para iterar ambas as listas em paralelo, pode-se usar tamanho da lista de origem names como limite máximo de iteração.

```
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)\
>>>
Cecilia
```

O problema dessa implementação é que o laço inteiro é visualmente poluído. Os índices em names e letters deixam o código difícil de ler. Os arrays são indexados pelo índice i do laço em dois pontos diferentes. Se usarmos enumerate (consulte o Item 10: “Prefira enumerate em vez de range”) a situação melhora um pouco, mas ainda assim não é a ideal.

```
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count
```

Para que o código fique realmente claro, o Python oferece a função nativa zip. No Python 3, zip envolve dois ou mais iteradores com um gerador simplificado. O gerador zip produz tuplas contendo o próximo valor para cada iterador. O código resultante é muito mais claro que indexar múltiplas listas.

```
for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
```

```
        max_letters = count
```

Há dois problemas em se usar a função nativa zip.

O primeiro é que no Python 2 o zip não é um gerador. Em vez disso, vai varrer os iteradores fornecidos até o fim e devolver uma lista com todas as tuplas criadas. Esse comportamento pode potencialmente usar uma grande porção da memória e causar uma falha grave, interrompendo o programa. Para usar o zip em iteradores muito grandes no Python 2, o melhor é empregar o izip, disponível no módulo nativo itertools (consulte o Item 46: “Use algoritmos e estruturas de dados nativos”).

O segundo problema é que o comportamento do zip é errático se os iteradores de entrada têm tamanhos diferentes. Por exemplo, se adicionar mais um nome na lista anterior mas esquecer de atualizar a contagem de caracteres. Rodar zip nas duas listas de entrada trará resultados inesperados.

```
names.append('Rosalind')
for name, count in zip(names, letters):
    print(name)
>>>
Cecilia
Lise
Marie
```

O novo item 'Rosalind' não aparece. É uma característica de como o zip funciona, gerando tuplas até que um dos iteradores chegue ao fim da sequência. Essa técnica funciona quando se sabe que os iteradores sempre terão o mesmo tamanho, o que quase sempre é o caso para listas derivadas criadas por abrangência. Em muitos outros casos, o comportamento truncador de zip é surpreendente e ruim. Se não houver possibilidade de garantir que as listas entregues ao zip sejam de igual tamanho, considere usar a função zip_longest do módulo nativo itertools (também chamado de izip_longest no Python 2).

Lembre-se

- A função nativa zip pode ser usada para iterar em paralelo sobre múltiplos iteradores.
- No Python 3, zip é um gerador simplificado que produz tuplas. No Python 2,

zip devolve uma lista de tuplas contendo o resultado de todas as iterações.

- zip trunca sua saída silenciosamente caso dois iteradores de tamanhos diferentes sejam apresentados.
- A função zip_longest, presente no módulo nativo itertools, permite iterar sobre iteradores múltiplos em paralelo sem se preocupar com seus tamanhos (consulte o Item 46: “Use algoritmos e estruturas de dados nativos”).

Item 12: Evite usar blocos else depois de laços for e while

Os laços do Python possuem um recurso extra que não está disponível em nenhuma outra linguagem de programação: é possível colocar um bloco else imediatamente após o bloco de código a ser repetido pelo laço.

```
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')
>>>
Loop 0
Loop 1
Loop 2
Else block!
```

Para nossa surpresa, o bloco else é executado imediatamente após o fim do laço. Por que então essa cláusula é chamada de “else” (em português, “senão”)? Por que não “and” (“e”)? Em uma estrutura if/else, o else significa “faça isso somente se o bloco anterior não for executado”. Em uma estrutura try/except, o except tem a mesma definição: “faça isso somente se o bloco anterior falhar”.

De forma semelhante, o else empregado em uma estrutura try/except/else segue esse padrão (consulte o Item 13: “Use todo o potencial dos blocos try/except/else/finally”) porque significa “faça isso se o bloco anterior não falhar”. try/finally também é intuitivo porque significa, “sempre execute o bloco de finalização depois de tentar o bloco anterior”.

Com todos os usos de else, except e finally em Python, um programador novato

pode entender erradamente que o else presente num loop for/else significa, “faça isso se o loop não puder ser completado”. Na realidade, ele faz o oposto! Inserir um comando break em um laço faz com que o bloco else seja completamente ignorado.

```
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
else:
    print('Else block!')
>>>
Loop 0
Loop 1
```

Outra surpresa é que o bloco else roda imediatamente caso o laço tente varrer uma sequência vazia.

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')
>>>
For Else block!
```

O bloco else também roda mesmo que um laço while seja inicialmente falso.

```
while False:
    print('Never runs')
else:
    print('While Else block!')
>>>
While Else block!
```

A explicação dada pelos desenvolvedores do Python para esse comportamento é que os blocos else são úteis em laços que procuram por algo. Por exemplo, imagine que se queira determinar se dois números são primos entre si, os chamados coprimos (seu único divisor comum é 1). No trecho de código a

seguir, um laço itera todos os possíveis divisores comuns e testa os números. Depois que todas as possibilidades forem tentadas, o laço termina. O bloco else roda quando os números são coprimos porque o laço não encontra um break.

```
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')
>>>
Testing 2
Testing 3
Testing 4
Coprime
```

Na prática, ninguém escreve código assim. Em vez disso, costuma-se criar uma função auxiliar para o cálculo, normalmente escrita em dois estilos bastante comuns.

O primeiro estilo é a função retornar falso na primeira oportunidade em que a condição de teste for satisfeita. Caso o laço termine sem que a condição seja satisfeita, a função retorna um valor default, verdadeiro.

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

A segunda maneira é ter uma variável de resultado que indique se a condição foi encontrada no laço. O laço é interrompido com break no momento em que a condição é satisfeita.

```
def coprime2(a, b):
```

```
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

Ambos os estilos são muito mais claros para qualquer um que queira ler o código sem estar familiarizado com ele. A expressividade que se ganha com o bloco else não vale a pena por conta da extenuante dificuldade que as pessoas sofrerão (incluindo você mesmo) para interpretar o programa no futuro. Estruturas de código simples como os laços devem ser autoexplicativas em Python. Jamais use blocos else depois de laços. Jamais!

Lembre-se

- O Python possui uma sintaxe especial que permite blocos else imediatamente após os blocos internos de laços for e while.
- O bloco else depois do laço roda sempre, a não ser que haja um break dentro do laço.
- Jamais use blocos else depois dos laços porque seu comportamento não é intuitivo e pode ser confuso.

Item 13: Use todo o potencial dos blocos try/except/else/finally

Existem quatro momentos específicos nos quais queremos tomar alguma decisão ou executar uma ação durante o tratamento de exceções no Python. Esses momentos são capturados funcionalmente com o emprego dos blocos de decisão try, except, else e finally. Cada bloco serve um propósito único nesse comando composto, e suas várias combinações são muito úteis (consulte o Item 51: “Defina uma Exception-raiz para isolar chamadores e APIs” para outro exemplo).

Blocos Finally

Use try/finally (em português, tente/finalmente) quando quiser que as exceções

sejam propagadas para os níveis superiores, mas também quiser rodar um código de limpeza quando a exceção ocorrer. Um uso bastante comum de try/finally é para assegurar o fechamento de manipuladores de arquivos, os file handles (consulte o Item 43: “Considere os comandos contextlib e with para um comportamento reutilizável de try/finally” para outro modo de implementação).

```
handle = open('/tmp/random_data.txt')  # Pode gerar um erro de E/S (IOError)
try:
    data = handle.read()  # Pode gerar um erro de decodificação Unicode
                          # (UnicodeDecodeError)
finally:
    handle.close()        # Sempre é executado depois do try
```

Qualquer exceção que ocorra na execução do método read sempre será propagada para o código chamador, mas mesmo assim o método close do manipulador de arquivos de nome handle é garantidamente executado no bloco finally. É obrigatório chamar open antes de try porque as exceções que ocorram no momento da abertura do arquivo (como IOError se o arquivo não existir) não devem ser tratados pelo bloco finally e, portanto, precisam se desviar dele.

Blocos Else

Use try/except/else (em português, tente/exceto se/senão) para deixar bem claro quais exceções serão tratadas pelo seu código e quais serão propagadas para os níveis superiores. Quando o bloco try não gera uma exceção, o bloco else será executado. O bloco else ajuda a minimizar a quantidade de código no bloco try e melhora a legibilidade. Por exemplo, imagine que se queira carregar dados de um dicionário JSON a partir de uma string e devolver o valor de uma chave contida nele.

```
def load_json_key(data, key):
    try:
        result_dict = json.loads(data)  # Pode gerar o erro ValueError
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]         # Pode gerar o erro KeyError
```

Se os dados não estão em um formato JSON válido, decodificá-los com json.loads gera um erro ValueError. A exceção é capturada pelo bloco except e tratada. Se por outro lado a decodificação tiver sucesso (ou seja, não gera erros), a busca pela chave ocorrerá no bloco else. Se a busca pela chave retornar qualquer exceção, esta será propagada para o chamador porque está fora do bloco try. A cláusula else assegura que o que está depois do try/except é visualmente separado do bloco except, o que deixa muito claro qual é o comportamento da propagação da exceção.

Tudo ao mesmo tempo agora

Use try/except/else/finally (em português, tente/exceto se/senão/finalmente) quando se quer fazer tudo isso em uma única estrutura composta. Por exemplo, imagine que se queira ler de um arquivo a descrição do trabalho a ser feito, processá-lo de atualizar o arquivo automaticamente. O bloco try é usado aqui para ler o arquivo e processá-lo. O bloco except é usado para tratar exceções geradas pelo bloco try e que sejam esperadas. O bloco else é usado para atualizar o arquivo e para permitir que exceções relacionadas possam propagar-se para o nível superior. O bloco finally limpa o manipulador de arquivo.

```
UNDEFINED = object()


def divide_json(path):
    handle = open(path, 'r+')   # Pode gerar o erro IOError
    try:
        data = handle.read()    # Pode gerar o erro UnicodeDecodeError
        op = json.loads(data)   # Pode gerar o erro ValueError
        value = (
            op['numerator'] /
            op['denominator'])  # Pode gerar o erro ZeroDivisionError
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
```

```
        handle.write(result)    # Pode gerar o erro IOError
        return value
    finally:
        handle.close()          # Sempre é executado
```

Esse leiaute é especialmente útil porque todos os blocos trabalham juntos de uma maneira intuitiva. Por exemplo, se uma exceção for gerada no bloco else enquanto o resultado estiver sendo reescrito no arquivo, o bloco finally será mesmo assim executado e fechará o manipulador de arquivo.

Lembre-se

- O comando composto try/finally permite rodar código de limpeza existindo ou não exceções a serem tratadas no bloco try.
- O bloco else ajuda a minimizar a quantidade de código nos blocos try e visualmente separar o resultado verdadeiro dos erros gerados pelos blocos try/except.
- Um bloco else pode ser usado para executar ações adicionais depois que um bloco try com resultado positivo, mas antes da limpeza comum a todos os resultados feita pelo bloco finally.

---

1 N. do T.: O nome do caractere "_" em inglês é underscore. No Brasil, é popular o uso incorreto de outra expressão inglesa, underline, para nomeá-lo, um erro muito comum, embora grosseiro. A expressão em português "caractere de sublinhado", apesar de constar de grande número de glossários, praticamente não é usada na literatura nacional. A palavra underline, como substantivo, não existe na língua inglesa e, repetimos, é um erro grosseiro usá-la em português.

2 N. do T.: O formato de nomes com as palavras capitalizadas (por exemplo, EstaClasse) também é conhecido como Camel Case. Há dois tipos de Camel Case: um deles, chamado de UpperCamelCase, tem a inicial maiúscula em todas as palavras que compõem o nome. O outro é o lowerCamelCase, que tem a primeira palavra iniciando em caixa baixa e as demais em caixa alta. Algumas linguagens usam um só tipo de Camel Case, outras usam os dois. No Java e no .NET, por exemplo, usa-se UpperCamelCase para os nomes de classes, e lowerCamelCase para os atributos e métodos da classe. Em Python, usa-se apenas UpperCamelCase para dar nome às classes, e qualquer outro elemento deve ser grafado todo em minúsculas, com as palavras separadas por underscores (conhecido como snake_case).

3 N. do T.: Em português, Statement, Declaration e Assertion costumam ser traduzidos para a mesma palavra, declaração, mas em programação cada palavra tem um significado diferente. Neste livro, para evitar ambiguidades, traduzimos Statement como Comando, Declaration como Declaração e Assertion como Asseveração.

4 N. do T.: Segundo esta tradução independente feita a partir da documentação oficial do Python 2.7, disponível em *http://turing.com.br/pydoc/2.7/tutorial/datastructures.html*.

CAPÍTULO 2

Funções

A primeira ferramenta organizacional que os programadores usam no Python é a *função*. Como em outras linguagens de programação, as funções permitem que se desmembre programas grandes em componentes menores e mais simples. Eles melhoram a legibilidade do código, deixando-o mais acessível, além de permitir reutilização e refatoração.

As funções no Python têm um grande número de recursos extras para tornar bem mais fácil a vida do programador. Alguns são semelhantes ao que é oferecido por outras linguagens de programação, mas muitos desses recursos são exclusivos do Python. Esses extras podem tornar o propósito da função bem mais óbvio, eliminar ruído desnecessário, deixar clara a intenção dos chamadores e, principalmente, reduzir significativamente os erros de programação mais sutis e difíceis de encontrar.

Item 14: Prefira exceções em vez de devolver None

Ao criar uma função utilitária, existe um acordo tácito entre os programadores em Python para reservar um significado especial para None quando usado como valor de retorno. Em alguns casos, faz sentido. Por exemplo, digamos que se queira uma função auxiliar que divida um número por outro. No caso de divisão por zero, retornar None parece uma solução natural porque o resultado, do ponto de vista matemático, é realmente indefinido.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Qualquer código que use essa função pode interpretar o valor de forma apropriada.

```
result = divide(x, y)
if result is None:
    print('Invalid inputs')
```

O que acontece se o numerador for zero? O valor de retorno também será zero, desde que o denominador seja diferente de zero. Isso pode causar problemas, se o resultado for avaliado em uma condição como as usadas no if. O programador pode, por descuido ou acidente, testar a existência de qualquer valor equivalente a False em vez de procurar especificamente por None (consulte o Item 4: “Escreva funções auxiliares em vez de expressões complexas” para uma situação semelhante).

```
x, y = 0, 5
result = divide(x, y)
if not result:
    print('Invalid inputs')  # Isso está errado!
```

Esse é um erro muito comum em Python quando se atribui um significado especial ao valor None. Usar None como valor de retorno em uma função tende, invariavelmente, a atrair problemas. Há duas maneiras de reduzir a possibilidade de ocorrência desses erros.

A primeira é dividir o valor de retorno em uma tupla de dois elementos. A primeira parte da tupla indica se a operação foi um sucesso ou não. A segunda é o resultado real computado.

```
def divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Os chamadores dessa função precisam desempacotar a tupla, o que os força a pelo menos tomar conhecimento do status da operação, em vez de simplesmente olhar o resultado da divisão.

```
success, result = divide(x, y)
if not success:
    print('Invalid inputs')
```

O problema dessa solução é que os chamadores podem facilmente ignorar a

primeira parte da tupla (usando o underscore sozinho como nome de variável, uma convenção do Python para variáveis que jamais serão usadas). O código resultante não parece errado à primeira vista, mas isso é tão ruim quanto retornar apenas None.

```
_, result = divide(x, y)
if not result:
    print('Invalid inputs')
```

A segunda maneira, bem melhor, para reduzir esses erros é nunca retornar None. Em vez disso, repasse a exceção para o chamador e deixe que ele lide com o problema. Em nosso caso, a exceção ZeroDivisionError foi transformada em ValueError para indicar ao chamador que os valores de entrada são ruins:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs') from e
```

Agora, o chamador deve tratar a exceção para o caso de valores incorretos de entrada (e este comportamento deve obrigatoriamente ser documentado. Consulte o Item 49: “Escreva docstrings para toda e qualquer função, classe e módulo”). O chamador não precisa mais implementar um teste condicional no valor de retorno da função. Se a função não gerar uma exceção, o valor de retorno deve ser válido. O resultado do tratamento da exceção deve ser claro.

```
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)

>>>
Result is 2.5
```

Lembre-se

- Funções que retornam None para indicar significados especiais estão fadadas a erros porque o None, em conjunto com outros valores (por exemplo, o zero ou uma string vazia), são todos interpretados como False em expressões condicionais.
- Gere exceções para indicar situações especiais em vez de retornar None. Assuma que o código chamador só conseguirá tratar corretamente as exceções se você as documentar de forma apropriada.

Item 15: Saiba como closures interagem com os escopos das variáveis

Digamos que se queira ordenar uma lista de números, mas priorizar um grupo de números que venha primeiro. Esse padrão é útil ao renderizar uma interface de usuário quando existirem mensagens importantes ou eventos excepcionais que devem ser mostrados antes de qualquer coisa.

Uma maneira bastante popular de se fazer isso é passar uma função auxiliar como o argumento key do método sort de uma lista. O valor devolvido pela função auxiliar será usado como o valor para classificar em ordem cada item da lista. A função auxiliar pode verificar se o item que está sendo avaliado está no grupo preferencial e variar a chave de ordenação conforme o caso.

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key=helper)
```

A função funciona muito bem para dados de entrada mais simples:

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
```

```
>>>
[2, 3, 5, 7, 1, 4, 6, 8]
```

A função opera da forma esperada por três razões:

- O Python suporta os *closures (fechamentos)*: funções que fazem referência a variáveis a partir do escopo em que foram definidas. É por isso que a função auxiliar (em nosso exemplo chamada helper) é capaz de acessar o argumento group estando dentro de sort_priority.
- As funções são *objetos de primeira classe* em Python, ou seja, podemos nos referir a eles diretamente, atribuí-los a variáveis como se fossem valores discretos, passá-los como argumentos para outras funções, compará-las em expressões e comandos if e muitas outras coisas. Por essa razão, o método sort pôde aceitar um closure como valor para o argumento key.
- O Python possui regras específicas para comparar tuplas. Primeiro são comparados os itens de índice zero, depois no índice um, depois dois, e assim por diante. É por isso que o valor de retorno do closure chamado helper causa dois grupos distintos de números na ordem de classificação.

Seria bacana se esta função retornasse a existência de itens de alta prioridade, para que a interface de usuário possa agir de acordo. Implementar esse comportamento parece bastante direto. Já existe uma função de closure para decidir em qual grupo cada número está. Por que não usar o closure para também levantar uma flag quando existirem itens de alta prioridade? Dessa forma, a função pode retornar o valor da flag depois de ter sido modificado pelo closure.

No código a seguir, isso foi implementado de uma forma aparentemente óbvia:

```
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True  # Parece simples
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Posso rodar a função com as mesmas entradas de antes.

```
found = sort_priority2(numbers, group)
```

```
print('Found:', found)
print(numbers)
>>>
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]
```

Os resultados ordenados estão corretos, mas o resultado de found está errado. Os itens de group foram efetivamente encontrados no conjunto numbers, mas a função retornou False. Por que isso aconteceu?

Quando uma variável é referenciada em uma expressão, o interpretador Python irá cruzar os limites de escopo para tentar resolver a referência, na seguinte ordem:

1. O escopo da função atual.
2. Quaisquer escopos que contenham o atual (por exemplo, funções que declarem a atual).
3. O escopo do módulo que contém o código (também chamado de *global scope ou escopo global*).
4. O escopo nativo (que contém funções como len e str).

Se nenhum desses locais possuir uma variável definida com o nome referenciado, uma exceção NameError é gerada.

A atribuição de um valor a uma variável funciona de forma diferente. Se a variável já estiver definida no escopo atual, ela assumirá o novo valor. Se a variável não existir no escopo atual, o Python considera a atribuição como uma definição de variável. O escopo da nova variável é a função que contém a atribuição.

O comportamento de atribuição explica o valor de retorno incorreto na função sort_priority2. O valor True foi atribuído à variável found no closure chamado helper. A atribuição no closure é encarada como uma nova definição de variável dentro do escopo de helper, e não como uma atribuição no escopo de sort_priority2.

```
def sort_priority2(numbers, group):
    found = False         # Aqui o escopo é 'sort_priority2'
    def helper(x):
```

```
        if x in group:
            found = True  # Aqui o escopo é 'helper' -- e isso é
                          # catastrófico!
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Esse problema é às vezes chamado de *scoping bug* ou *falha de escopo* porque pode ser surpreendente para os programadores iniciantes. Porém, isso não é uma falha – pelo contrário, é o resultado pretendido. Esse comportamento impede que variáveis locais em uma função poluam o módulo que acondiciona tudo. Se não fosse assim, cada atribuição dentro de uma função produziria lixo de variáveis no escopo global do módulo. Além do ruído desnecessário, o efeito recíproco das variáveis globais resultantes poderia causar falhas bastante obscuras.

Entregando dados ao mundo externo

No Python 3, existe uma sintaxe especial para extrair dados de um closure. O comando nonlocal é usado para indicar que, em uma variável que tenha sido definida por atribuição, o cruzamento de limites de escopo tem permissão para acontecer em um nome de variável específico. O único fator limitante é que nonlocal não ousará cruzar a fronteira com o escopo do módulo, evitando poluí-lo com variáveis globais espúrias.

No exemplo a seguir, a mesma função é definida novamente usando nonlocal:

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

O comando nonlocal deixa claro que o dado está sendo atribuído em outro escopo, fora do closure. É complementar ao comando global, que força a definição da variável criada por atribuição diretamente no escopo global do módulo.

Entretanto, da mesma forma que no antiexemplo das variáveis globais, recomendamos usar nonlocal apenas para funções simples, nada mais! Os efeitos colaterais de nonlocal podem ser difíceis de rastrear. É especialmente difícil entendê-los em funções muito longas, porque o comando nonlocal, que reserva o nome da variável, está a uma distância de muitas linhas do local onde essas variáveis serão criadas por atribuição.

Quando o uso de nonlocal começa a complicar as coisas, é hora de envolver o estado em uma função auxiliar. No exemplo a seguir, definimos uma classe que produz o mesmo resultado que a solução com o nonlocal. É um pouquinho mais longa, mas muito mais fácil de ler (consulte o Item 23: “Aceite funções para interfaces simples em vez de classes” para mais detalhes sobre o método especial __call__).

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

Escopos no Python 2

Infelizmente, o Python 2 não suporta a palavra-chave nonlocal. Para conseguir o

mesmo comportamento, é preciso usar um artifício alternativo que tira proveito das regras de escopo do Python. Esta técnica não é lá muito elegante, mas é aceita como “politicamente correta” pelos programadores em Python.

```
# Python 2
def sort_priority(numbers, group):
    found = [False]
    def helper(x):
        if x in group:
            found[0] = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found[0]
```

Como explicado anteriormente, o Python sai dos limites de escopo onde a variável found é referenciada para resolver seu valor atual. O truque está no fato de que o valor de found é uma lista, que é um tipo de dados mutável. Isso quer dizer que, uma vez que seus valores tenham sido lidos, o closure pode modificar o estado de found para enviar dados para fora do escopo interno (desde que found[0] = True).

Essa técnica também funciona quando a variável usada para cruzar as fronteiras de escopo é um dicionário, um conjunto, ou uma instância de uma classe definida pelo programador.

Lembre-se

- As funções closure podem se referir a variáveis de qualquer um dos escopos nas quais elas foram definidas.
- Por default, os closures não conseguem afetar as variáveis de escopos hierarquicamente superiores ao seu.
- No Python 3, use o comando nonlocal para indicar quando um closure pode modificar uma variável definida em escopos superiores.
- Em Python 2, use uma variável mutável (por exemplo, uma lista de um único item) para contornar a ausência do comando nonlocal.
- Evite usar comandos nonlocal para funções complexas, use-as apenas em

casos muito simples.

Item 16: Prefira geradores em vez de retornar listas

A escolha mais simples para funções que produzem uma sequência de resultados é retornar uma lista de itens. Por exemplo, digamos que se queira encontrar o índice de cada palavra em uma string. No exemplo a seguir, os resultados são acumulados em uma lista usando o método append e devolvendo essa lista no final da função:

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

O exemplo funciona às mil maravilhas para dados de entrada quaisquer:

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:3])
```

```
>>>
[0, 5, 11]
```

Porém, há dois problemas na implementação da função index_words.

O primeiro é a densidade e “sujeira” do código. Cada vez que um novo resultado é encontrado, o método append é chamado. O elemento principal da chamada do método (result.append) não dá a devida ênfase ao valor que está sendo adicionado à lista (index + 1). Há uma linha para criar o resultado e outra para retorná-lo. Enquanto o corpo da função contém mais ou menos 130 caracteres (sem contar os espaços), apenas 75 deles são realmente importantes.

Uma maneira muito melhor de implementar essa função emprega o conceito de *geradores*. Os geradores são funções que usam expressões yield. Ao serem chamadas, as funções geradores não são, de verdade, executadas. Em vez disso,

elas retornam um iterador. Em cada chamada à função nativa next, o iterador avançará o gerador para a próxima expressão do yield. Cada valor passado ao yield pelo gerador será devolvido pelo iterador ao chamador.

No exemplo a seguir, a função geradora definida produz os mesmos resultados de antes:

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

O código é muito mais fácil de ler em comparação com o anterior, porque todas as interações com a lista resultante foram eliminadas. Os resultados, em vez disso, são passados a expressões yield. O iterador retornado pela chamada ao gerador pode facilmente ser convertido em uma lista, basta passá-lo como argumento da função nativa list (consulte o Item 9: “Considere usar expressões geradoras em abrangências muito grandes” para saber como isso funciona).

```
result = list(index_words_iter(address))
```

O segundo problema com index_words é que ele precisa ter todos os resultados de antemão para poder criar a lista e devolvê-la ao chamador. Para dados de entrada de grande monta, seu programa pode esgotar a memória e travar. Em contrapartida, a versão que emprega geradores pode facilmente ser adaptada para aceitar dados de entrada de tamanho arbitrário.

No exemplo a seguir, definimos um gerador que obtém dados de um arquivo de entrada, uma linha por vez, e devolve na saída uma palavra por vez. A memória de trabalho desta função está limitada ao tamanho máximo de uma linha de entrada.

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
```

```
            offset += 1
            if letter == ' ':
              yield offset
```

Rodar o gerador produz o mesmo resultado:

```
with open('/tmp/address.txt', 'r') as f:
    it = index_file(f)
    results = islice(it, 0, 3)
    print(list(results))

>>>
[0, 5, 11]
```

A única pegadinha na definição de geradores é que os chamadores devem estar cientes de que os iteradores devolvidos pelo gerador administram seu estado (stateful) e, portanto, não podem ser reutilizados (consulte o Item 17: “Seja conservador quando iterar sobre argumentos”).

Lembre-se

- Empregar geradores pode tornar seu código mais claro e legível que a alternativa de retornar listas de resultados acumulados.
- O iterador devolvido pelo gerador produz um conjunto de valores passados para as expressões yield, que estão dentro do corpo da função geradora.
- Os geradores podem produzir uma sequência de valores de saída a partir de valores de entrada arbitrariamente grandes porque a memória de trabalho não inclui o conjunto completo de todas as entradas e saídas.

Item 17: Seja conservador quando iterar sobre argumentos

Quando uma função toma uma lista de objetos como parâmetro, é importante iterar sobre essa lista várias vezes. Por exemplo, digamos que se queira analisar os números do turismo do estado americano do Texas. Imagine que o conjunto de dados é o número de visitantes em cada cidade (em milhões de visitantes por ano) e queremos descobrir a porcentagem de turismo que cada cidade recebe.

Para isso, é necessária uma função de normalização que soma, a partir dos dados

de entrada, o número de visitantes de cada cidade para obter o número total de visitantes no Texas. Depois, divide o número de cada cidade pelo número total do estado para encontrar a porcentagem que aquele município contribuiu para o turismo estadual.

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

A função funciona quando lhe é fornecida uma lista de quantidades de visitação.

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Para usar a função em um nível mais alto, é necessário ler os dados a partir de um arquivo que contém os dados de cada município do Texas. Para isso, é definido um gerador porque eu posso usar a mesma função mais tarde, quando quiser computar os números do turismo do mundo inteiro, um conjunto de dados muito maior. (Consulte o Item 16: “Prefira geradores em vez de retornar listas”.)

```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)
```

Para nosso espanto, chamar a função normalize usando como argumento o iterador, que carrega os valores produzidos pelo gerador, não produz nenhum resultado.

```
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize(it)
print(percentages)
```

```
>>>
[]
```

A causa desse comportamento bizarro é que o iterador produz seus resultados uma única vez. Quando insistimos em iterar sobre um iterador ou gerador que já levantou uma exceção StopIteration, não obteremos nenhum resultado na segunda vez.

```
it = read_visits('/tmp/my_numbers.txt')
print(list(it))
print(list(it))  # Aqui, o iterador já está exaurido
>>>
[15, 35, 80]
[]
```

A confusão fica maior porque o Python não mostra nenhum erro quando tentamos iterar sobre um iterador já exaurido. Os laços for, o construtor list e muitas outras funções da biblioteca-padrão do Python esperam que a exceção StopIteration seja gerada durante a operação normal. Essas funções não conseguem distinguir entre um iterador que desde o começo não possui nada para apresentar na saída e outro iterador que já está exaurido.

Para resolver esse problema, podemos exaurir explicitamente um iterador de entrada e manter uma cópia completa de seu conteúdo em uma lista. Podemos então iterar sobre a lista quantas vezes for necessário. O código a seguir mostra a mesma função de antes, mas copia de forma preventiva o iterador de entrada:

```
def normalize_copy(numbers):
    numbers = list(numbers)  # Copia o iterador
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Agora, a função devolve um resultado coerente quando lhe é fornecido um valor de retorno de um gerador:

```
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

O problema com essa técnica é que a cópia do iterador de entrada pode ser muito grande. Copiar o iterador pode causar uso excessivo de memória e o travamento do programa. Uma forma de contornar o erro é aceitar uma função que retorne um novo iterador a cada vez que for chamada.

```
def normalize_func(get_iter):
    total = sum(get_iter())   # Novo iterador
    result = []
    for value in get_iter():  # Novo iterador
        percent = 100 * value / total
        result.append(percent)
    return result
```

Para usar normalize_func, basta passar uma expressão lambda, que chama o gerador e produz um novo iterador a cada chamada.

```
percentages = normalize_func(lambda: read_visits(path))
```

Embora funcione, ter de usar uma função lambda dessa maneira é bastante desajeitado. A melhor maneira de obter o mesmo resultado é providenciar uma nova classe de contenção que implementa o *protocolo de iteração*.

No Python, o protocolo de iteração é a maneira como os laços for e expressões afins navegam pelo conteúdo de um tipo de dados contêiner. Quando o Python vê um comando do tipo for x in foo, o que ele fará na verdade é chamar iter(foo). A função nativa iter, por sua vez, chama o método especial foo.__iter__. O método __iter__ retorna um objeto iterador (que por sua vez implementa o método especial __next__). Depois, o laço for chama repetidamente a função nativa next no objeto do iterador até que esteja exaurido (ou seja, todos os itens foram lidos) e gera uma exceção StopIteration.

Parece complicado, mas em termos práticos podemos conseguir exatamente o mesmo comportamento para as classes implementando o método __iter__ como um gerador. No exemplo a seguir, definimos uma classe iterável do tipo

contêiner que lê os arquivos contendo os dados de turismo:

```
class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

O novo tipo contêiner funciona corretamente quando passado à função original sem qualquer modificação.

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Funciona porque o método sum em normalize chama ReadVisits.__iter__ para alocar um novo objeto iterador. O laço for que normaliza os números também chama __iter__ para alocar um segundo objeto iterador. Cada um desses iteradores será progressivamente varrido e exaurido de forma independente, assegurando que cada iteração única veja todos os valores de dados de entrada. A única desvantagem dessa técnica é que ela lê os dados de entrada mais de uma vez.

Agora que entendemos o funcionamento de contêiner como ReadVisits, podemos escrever funções para assegurar que os parâmetros não sejam simples iteradores. O protocolo rege que, sempre que um iterador é passado para a função nativa iter, esta devolve o mesmo iterador. Em contrapartida, quando um valor do tipo contêiner é passado ao iter, um novo objeto iterador será retornado a cada vez. Assim, podemos testar a existência desse comportamento para cada valor de entrada e gerar uma exceção TypeError para rejeitar iteradores.

```
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers):  # Um iterador foi encontrado.
                                        # Fora com ele!
```

```
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Essa solução evita ter que copiar o iterador de entrada por completo, como fizemos em normalize_copy, mas também precisamos iterar sobre os dados de entrada mais de uma vez. Essa função comporta-se como esperado para list e ReadVisits na entrada porque eles são contêineres. Funcionaria para qualquer tipo de contêiner que seguisse o protocolo de iteração.

```
visits = [15, 35, 80]
normalize_defensive(visits)  # Nenhum erro aqui!
visits = ReadVisits(path)
normalize_defensive(visits)  # Nenhum erro aqui!
```

A função gera uma exceção caso a entrada seja iterável, mas não seja um contêiner.

```
it = iter(visits)
normalize_defensive(it)
>>>
TypeError: Must supply a container
```

Lembre-se

- Tome cuidado com funções que façam iteração sobre argumentos de entrada mais de uma vez. Se esses argumentos forem, eles mesmos, iteradores também, é possível que o comportamento do código não seja o esperado, ou que alguns valores não sejam produzidos.
- O protocolo de iteração do Python define como os iteradores e contêineres conversam com as funções nativas iter e next, laços for e expressões afins.
- Podemos facilmente definir nosso próprio contêiner iterável implementando o método __iter__ como um gerador.

- É possível detectar que o valor de entrada é um iterador (em vez de um contêiner) verificando se uma dupla chamada a iter sobre o objeto produz o mesmo resultado, que pode ser varrido passo a passo com a função nativa next.

Item 18: Reduza a poluição visual com argumentos opcionais

Aceitar opcionalmente parâmetros posicionais (comumente chamados de *star args* ou *argumentos estrelinhas* em referência ao nome convencional do parâmetro, *args) pode tornar a chamada à função mais clara e remover uma grande quantidade de *poluição visual*.

Por exemplo, digamos que se queira armazenar em um log as informações de depuração (debugging) de determinado sistema. Com um número fixo de argumentos, seria preciso uma função que recebesse, na entrada, uma mensagem em uma lista de valores.

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', [1, 2])
log('Hi there', [])
>>>
My numbers are: 1, 2
Hi there
```

Ter a obrigação de passar uma lista vazia quando não tivermos valores a registrar é desajeitado e barulhento. Seria melhor simplesmente ignorar o segundo argumento. Felizmente, isso pode ser feito no Python se prefixarmos o último parâmetro posicional com *. O primeiro parâmetro, a mensagem de log, é obrigatório, enquanto quaisquer argumentos posicionais subsequentes são opcionais. O corpo da função não precisa ser modificado, apenas o chamador é

que precisa se adaptar ao novo formato.

```
def log(message, *values):  # A única diferença
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', 1, 2)
log('Hi there')  # Muito melhor!
```

```
>>>
My numbers are: 1, 2
Hi there
```

Veja que os argumentos não precisam mais estar numa lista, mas se você já tem uma lista e quiser chamar uma função com argumentos expansíveis como esta nossa log, pode-se usar o operador * também na chamada da função. O * instrui o Python para passar os itens na sequência como se fossem argumentos posicionais.

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)
```

```
>>>
Favorite colors: 7, 33, 99
```

Existem dois problemas a considerar quando decidimos aceitar uma quantidade variável de argumentos posicionais.

O primeiro problema é que os argumentos variáveis são sempre transformados em uma tupla antes de passados para sua função. Isso significa que, caso o chamador da função use o operador * em um gerador, este será iterado até a exaustão. A tupla resultante incluirá todos os valores produzidos pelo gerador, o que pode consumir uma grande quantidade de memória e causar o travamento do programa.

```
def my_generator():
    for i in range(10):
```

```
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)
>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Funções que aceitam *args são mais bem empregadas em situações nas quais conhecemos a quantidade de itens na lista de argumentos, e esta é razoavelmente pequena. É ideal para uma função que aceita muitos literais ou nomes de variável ao mesmo tempo. Esse arranjo existe unicamente para a conveniência do programador e para melhorar a legibilidade do código.

O segundo problema com *args é que não será possível adicionar novos argumentos posicionais (ou seja, de posição fixa) à função sem que todos os chamadores tenham de ser modificados também. Caso um novo argumento posicional seja inserido na frente da lista de argumentos de posição variável, os códigos chamadores dessa função poderão deixar de funcionar se não forem também atualizados.

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s: %s' % (sequence, message, values_str))

log(1, 'Favorites', 7, 33)      # Nova sintaxe de chamada funciona
log('Favorite numbers', 7, 33)  # A sintaxe antiga deixa de funcionar
                                # corretamente
>>>
1: Favorites: 7, 33
Favorite numbers: 7: 33
```

O problema nesse código está na segunda chamada à função log, que usou o 7 como parâmetro message porque um não foi fornecido um argumento para sequence. Bugs como esse são de difícil depuração porque o código funciona sem levantar nenhuma exceção. Para evitar a ocorrência desse tipo de erro, procure usar argumentos em palavras-chave quando quiser estender a funcionalidade de funções que aceitem *args (consulte o Item 21: “Garanta a legibilidade com argumentos por palavras-chave”).

Lembre-se

- As funções podem aceitar um número variável de argumentos posicionais usando *args na definição da função com o comando def.
- Os itens em uma sequência podem ser usados como argumentos posicionais em uma função com o operador *.
- Empregar o operador * com um gerador pode fazer com que o programa esgote a memória e trave.
- Adicionar novos parâmetros posicionais a funções que aceitem *args pode introduzir bugs difíceis de encontrar.

Item 19: Implemente comportamento opcional usando palavras-chave como argumentos

Como a maioria das linguagens de programação, chamar uma função em Python permite passar argumentos posicionais, ou seja, cuja atribuição à variável interna correta depende da posição do valor na lista de argumentos.

```
def remainder(number, divisor):
    return number % divisor


assert remainder(20, 7) == 6
```

Todos os argumentos posicionais em funções do Python podem ser também passados por palavra-chave, no qual o nome do argumento é usado em uma atribuição dentro dos parênteses da chamada à função. Os argumentos com palavras-chave podem ser passados em qualquer ordem, desde que todos os argumentos posicionais obrigatórios sejam especificados. Podemos misturar argumentos posicionais e por palavra-chave à vontade. As chamadas mostradas a

seguir são absolutamente equivalentes:

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Os argumentos posicionais devem ser especificados antes dos argumentos por palavra-chave.

```
remainder(number=20, 7)
>>>
SyntaxError: non-keyword arg after keyword arg
```

Cada argumento só pode ser especificado uma única vez.

```
remainder(20, number=7)
>>>
TypeError: remainder() got multiple values for argument 'number'
```

A flexibilidade dos argumentos por palavra-chave nos acarretam três benefícios significativos.

A primeira vantagem é que os argumentos por palavra-chave tornam a função mais clara para alguém que não esteja familiarizado com o código. Com a chamada remainder(20, 7), não fica evidente qual argumento é o número e qual é o divisor sem que o leitor precise consultar a implementação do método remainder. Na chamada com argumentos em palavra-chave, number=20 e divisor=7 deixa imediatamente óbvio qual parâmetro está sendo usado para cada propósito.

O segundo benefício de impacto dos argumentos por palavra-chave é que eles podem ter valores default especificados na definição da função. Isso permite que uma função ofereça funcionalidades opcionais quando necessário, mas deixa que o chamador aceite os valores-padrão na maior parte do tempo. Isso pode eliminar código repetitivo e reduzir ruído.

Por exemplo, digamos que se queira computar o fluxo de um fluido que jorre para o interior de um vasilhame. Se o vasilhame estiver sobre uma balança, podemos usar a diferença entre medições de peso em dois momentos específicos para determinar a vazão.

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff

weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print('%.3f kg per second' % flow)
>>>
0.167 kg per second
```

Tipicamente, é útil conhecer a vazão em quilogramas por segundo. Outras vezes, seria desejável usar as últimas medições para chegar a uma aproximação de escalas de tempo maiores, como horas ou dias. Podemos implementar o novo comportamento na mesma função adicionando um argumento para o fator de escala de tempo.

```
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

O problema é que agora precisamos especificar o argumento period toda vez que a função é chamada, mesmo no caso mais comum de vazão por segundo (em que o período é 1).

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

Para tornar a função um pouco menos poluída visualmente, podemos definir um valor default para o argumento period.

```
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period
```

A partir de agora, o argumento period é opcional.

```
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

Isso funciona bem para valores-padrão mais simples. Para valores default complexos, a coisa fica bem menos trivial – consulte o Item 20: “Use None e docstrings para especificar argumentos default dinâmicos e específicos”).

A terceira razão para usar argumentos com palavras-chave é que eles oferecem uma maneira poderosa de estender os parâmetros da função ao mesmo tempo em

que permanece compatível com os chamadores existentes. Isso permite oferecer funcionalidade adicional a uma função existente sem que seja necessário alterar o código chamador (que pode estar em muitos lugares), reduzindo a chance de introduzir bugs.

Por exemplo, digamos que se queira estender a função flow_rate anterior para calcular a vazão em outras unidades de massa além do quilograma. Isso pode ser feito adicionando um novo parâmetro que permita definir uma taxa de conversão para sua unidade preferida.

```
def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff / units_per_kg) / time_diff) * period
```

O valor-padrão para o argumento units_per_kg é 1, portanto o valor da massa devolvida continua sendo expresso em quilogramas. Isso significa que todos os chamadores atuais não verão nenhuma mudança no comportamento da função. Novos chamadores a flow_rate podem especificar um novo argumento por palavra-chave para obter o novo comportamento.

```
pounds_per_hour = flow_rate(weight_diff, time_diff,
                            period=3600, units_per_kg=2.2)
```

O único problema com essa técnica é que argumentos opcionais por palavra-chave como period e units_per_kg podem ainda ser especificados como argumentos posicionais.

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

Informar argumentos opcionais posicionalmente pode ser confuso porque não fica claro a que parâmetros os valores 3600 e 2.2 correspondem. A melhor prática neste caso é sempre especificar os parâmetros opcionais usando seus nomes (ou seja, por palavra-chave) e jamais passá-los como argumentos posicionais.

Nota

Compatibilidade com implementações anteriores usando argumentos opcionais por palavra-chave é crucial para funções que aceitam *args (consulte o Item 18: “Reduza a poluição visual com argumentos opcionais ”). No entanto, uma atitude ainda melhor seria usar apenas argumentos com palavras-chave (consulte o Item 21: “Garanta a legibilidade com argumentos

por palavras-chave”).

Lembre-se

- Os argumentos de uma função podem ser especificados por posição ou por palavra-chave.
- O uso de palavras-chave deixa mais clara a finalidade de cada argumento, enquanto o uso de argumentos posicionais pode ser bastante confuso.
- Argumentos em palavras-chave com valores default facilita a implementação de novos comportamentos a uma função existente, especialmente quando a função já é usada por código chamador antes da modificação.
- Parâmetros opcionais por palavra-chave devem ser sempre passados por palavra-chave, nunca por posição.

Item 20: Use None e docstrings para especificar argumentos default dinâmicos e específicos

Às vezes, precisamos usar um tipo não estático como valor-padrão de um argumento por palavra-chave. Por exemplo, digamos que se queira imprimir mensagens de log com a marcação da hora exata em que ocorreu o evento sendo registrado. No caso padrão, a mensagem deve incluir o horário em que a função foi chamada. O exemplo a seguir pode ser uma alternativa, desde que se saiba que os argumentos default são recalculados a cada chamada da função.

```
def log(message, when=datetime.now()):
    print('%s: %s' % (when, message))

log('Hi there!')
sleep(0.1)
log('Hi again!')
>>>
2014-11-15 21:10:10.371432: Hi there!
2014-11-15 21:10:10.371432: Hi again!
```

Contudo, não são! Os horários são os mesmos porque datetime.now é executado uma única vez: quando a função foi definida. Os valores default dos argumentos

são calculados apenas uma vez quando o módulo é carregado, o que normalmente acontece na carga inicial do programa. Depois que o módulo contendo o código é carregado, o argumento default datetime.now jamais será recalculado.

A forma convencional de conseguir o efeito desejado em Python é definir o valor default como None e documentar o comportamento completo na docstring (consulte o Item 49: “Escreva docstrings para toda e qualquer função, classe e módulo”). Quando seu código vir o valor None no argumento, o corpo da função calcula localmente o valor default.

```
def log(message, when=None):
    """Log a message with a timestamp.[1]

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    when = datetime.now() if when is None else when
    print('%s: %s' % (when, message))
```

Agora os horários de cada mensagem serão diferentes.

```
log('Hi there!')
sleep(0.1)
log('Hi again!')
>>>
2014-11-15 21:10:10.472303: Hi there!
2014-11-15 21:10:10.573395: Hi again!
```

Usar None para o argumento é especialmente importante quando os argumentos são mutáveis. Por exemplo, digamos que se queira carregar um valor codificado em formato JSON. Se a decodificação do dado falhar, queremos que um dicionário vazio seja devolvido por default. É uma técnica que alguns poderão querer experimentar.

```
def decode(data, default={}):
```

```
    try:
        return json.loads(data)
    except ValueError:
        return default
```

O problema com ela é o mesmo do exemplo anterior, datetime.now. O dicionário especificado para o parâmetro default será compartilhado por todas as chamadas a decode porque o valor default do argumento é calculado apenas uma vez (na carga do módulo). Isso pode resultar em um comportamento extremamente imprevisível.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

Pelo código acima, esperaríamos obter dois dicionários diferentes, cada um com um único par chave:valor. Todavia, ao modificar um, o outro é também alterado. O culpado é o fato de que foo e bar são iguais ao parâmetro default da função decode. São, portanto, todos o mesmo objeto de dicionário.

```
assert foo is bar
```

O truque para que tudo funcione é definir o valor default do argumento como None e documentar o comportamento esperado na docstring da função.

```
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
            Defaults to an empty dictionary.
```

```
    """
    if default is None:
        default = {}
    try:
        return json.loads(data)
    except ValueError:
        return default
```

Agora, ao rodar o mesmo código de teste de antes, temos os resultados esperados.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

Lembre-se

- Os argumentos default são calculados apenas uma vez: durante a definição da função no momento em que o módulo está sendo carregado na memória. Isso pode causar comportamento imprevisível quando são usados valores dinâmicos (como {} ou []).
- Use None como valor default para argumentos por palavra-chave que devam assumir um valor dinâmico. Documente o comportamento-padrão na docstring da função.

Item 21: Garanta a legibilidade com argumentos por palavras-chave

O recurso do Python de passar argumentos de função por palavra-chave é poderoso (consulte o Item 19: “Implemente comportamento opcional usando

palavras-chave como argumentos”). A flexibilidade dos argumentos por palavra-chave permite escrever código que se explica sozinho em qualquer caso de uso.

Por exemplo, digamos que se queira dividir um número por outro, mas com muito cuidado para reconhecer e tratar os casos especiais. Às vezes, queremos ignorar uma exceção ZeroDivisionError e simplesmente retornar infinito para o chamador. Outras vezes, queremos ignorar uma exceção de OverflowError e retornar o valor zero em seu lugar.

```
def safe_division(number, divisor, ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

O emprego dessa função é bastante direto. A chamada a seguir ignora o estouro em float simplesmente retorna zero.

```
result = safe_division(1, 10**500, True, False)
print(result)
>>>
0.0
```

A chamada a seguir ignora o erro de dividir por zero e retorna o objeto de infinito.

```
result = safe_division(1, 0, False, True)
print(result)
>>>
```

```
inf
```

O problema é que é fácil confundir a posição dos dois argumentos booleanos que controlam o comportamento que ignora as exceções. A obscuridade pode causar bugs difíceis de encontrar e depurar. Uma maneira de melhorar a legibilidade desse código é empregar argumentos por palavra-chave. Por default, a função deve ser cautelosa e sempre levantar as exceções.

```
def safe_division_b(number, divisor,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

Os chamadores podem, opcionalmente, usar os argumentos por palavra-chave para especificar qual dos casos especiais ativar, alterando o comportamento-padrão.

```
safe_division_b(1, 10**500, ignore_overflow=True)
safe_division_b(1, 0, ignore_zero_division=True)
```

Contudo, como o uso de argumentos por palavra-chave é opcional, não há nada que obrigue os chamadores a usar as tais palavras-chave para mais clareza. Mesmo com a nova definição de safe_division_b, ainda é possível empregar a velha maneira de especificar posicionalmente os argumentos.

```
safe_division_b(1, 10**500, True, False)
```

Em funções complexas como esta, é melhor obrigar os chamadores a serem claros em suas intenções. No Python 3, é possível exigir que o chamador explique as coisas com mais clareza. Para isso, defina suas funções com argumentos que funcionam apenas por palavra-chave. Argumentos definidos dessa maneira só podem ser passados por palavra-chave, nunca por posição.

No exemplo a seguir, redefinimos a função safe_division para aceitar apenas argumentos por palavra-chave. O símbolo * na lista de argumentos indica o fim dos argumentos posicionais e o começo dos argumentos exclusivamente por palavra-chave.

```
def safe_division_c(number, divisor, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

A partir de agora, chamar a função com argumentos posicionais onde é proibido não funcionará mais.

```
safe_division_c(1, 10**500, True, False)
>>>
TypeError: safe_division_c() takes 2 positional arguments but 4 were given
```

Argumentos por palavra-chave, aos quais é passado um valor diferente do definido como valor-padrão, funcionam sem problemas.

```
safe_division_c(1, 0, ignore_zero_division=True)  # Tudo bem aqui
try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    pass  # Como já era esperado
```

Argumentos por palavra-chave exclusivos em Python 2

Infelizmente, o Python 2 não possui uma sintaxe explícita para especificar argumentos exclusivamente por palavra-chave como no Python 3. Podemos conseguir o mesmo comportamento, a saber, levantar uma exceção TypeErrors para uma chamada inválida à função, usando o operador ** nas listas de argumentos. O operador ** é semelhante ao operador * (consulte o Item 18: “Reduza a poluição visual com argumentos opcionais”), mas em vez de aceitar um número variável de argumentos posicionais, aceitaremos um número variável de argumentos por palavra-chave, mesmo que eles não estejam definidos.

```
# Python 2
def print_args(*args, **kwargs):
    print 'Positional:', args
    print 'Keyword:   ', kwargs

print_args(1, 2, foo='bar', stuff='meep')
>>>
Positional: (1, 2)
Keyword:    {'foo': 'bar', 'stuff': 'meep'}
```

Para fazer com que safe_division receba argumentos exclusivamente por

palavra-chave no Python 2, faça a função aceitar **kwargs. Depois, é possível extrair os argumentos de palavra-chave do dicionário kwargs usando o método pop, como se fosse uma pilha. O segundo argumento do método pop deve especificar o valor default caso a chave não esteja presente. Por fim, certifique-se de que nenhum outro argumento por palavra-chave tenha sido deixado em kwargs para evitar que os chamadores passem argumentos inválidos.

```
# Python 2
def safe_division_d(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_div = kwargs.pop('ignore_zero_division', False)
    if kwargs:
        raise TypeError('Unexpected **kwargs: %r' % kwargs)
    # ...
```

Agora, podemos chamar a função com e sem argumentos por palavra-chave.

```
safe_division_d(1, 10)
safe_division_d(1, 0, ignore_zero_division=True)
safe_division_d(1, 10**500, ignore_overflow=True)
```

A tentativa de passar argumentos posicionalmente quando se espera que seja especificada a palavra-chave não funcionará, semelhante ao que acontece no Python 3.

```
safe_division_d(1, 0, False, True)
```

```
>>>
TypeError: safe_division_d() takes 2 positional arguments but 4 were given
```

A tentativa de passar uma palavra-chave não definida também causa erro.

```
safe_division_d(0, 0, unexpected=True)
```

```
>>>
TypeError: Unexpected **kwargs: {'unexpected': True}
```

Lembre-se

- Argumentos por palavra-chave tornam a intenção da função muito mais clara no momento da chamada.
- Use argumentos que funcionem exclusivamente por palavra-chave para forçar

os chamadores a usar as palavras-chave em funções potencialmente confusas, especialmente aquelas que aceitam um ou mais modificadores booleanos.

- O Python 3 suporta uma sintaxe explícita para argumentos exclusivamente por palavra-chave em funções.
- O Python 2 pode emular o comportamento de argumentos exclusivamente por palavra-chave em funções se for definida com **kwargs, tomando o cuidado de gerar manualmente as exceções TypeError.

---

1 N. do T.: O inglês é considerado o idioma universal dos programadores. Mesmo que seu sistema completo seja atualmente usado apenas por você ou pela sua empresa no Brasil, a boa prática manda escrever todas as docstrings em inglês para o caso de, no futuro, o código precisar ser revisto ou modificado por programadores de outros países. Também é importante usar a língua inglesa nas docstrings caso alguma função, módulo ou classe do sistema, ou mesmo o sistema todo, seja liberado em alguma licença de código aberto, que por definição fica disponível para pessoas do mundo todo.

CAPÍTULO 3

Classes e herança

Por ser uma linguagem de programação orientada a objetos, o Python suporta um vasto leque de recursos como herança, polimorfismo e encapsulamento. Para fazer as coisas acontecerem em Python precisamos, na maioria das vezes, escrever novas classes e definir como interagir com elas por suas interfaces e hierarquias.

As classes e a herança no Python simplificam expressar o comportamento pretendido de seu programa por meio de objetos, que ainda permitem aprimorar e expandir a funcionalidade do sistema ao longo de sua vida útil, e oferecem flexibilidade em um ambiente com requisitos sempre em mudança. Saber como usá-los corretamente permite escrever código simples de manter.

Item 22: Prefira classes auxiliares em vez de administrar registros complexos com dicionários e tuplas

O dicionário é um tipo de dado nativo do Python perfeito para manter de forma dinâmica os estados internos de um objeto durante toda a sua vida útil. Quando falamos em *manter de forma dinâmica*, queremos indicar situações em que é necessário administrar registros complexos de um conjunto inesperado de identificadores. Por exemplo, digamos que se queira gravar as notas de um conjunto de estudantes cujos nomes não são conhecidos de antemão. Podemos definir uma classe para guardar os nomes em um dicionário em vez de usar um atributo predefinido para cada estudante.

```
class SimpleGradebook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
```

```
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

Usar a classe é muito simples.

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
# ...
print(book.average_grade('Isaac Newton'))
>>>
90.0
```

Os dicionários são tão fáceis de usar que existe o risco de estendê-los demais e escrever um código frágil. Por exemplo, digamos que se queira estender a classe SimpleGradebook para manter uma lista de notas por matéria, e não apenas a média final do estudante. Podemos fazê-lo alterando o dicionário _grades para relacionar os nomes dos estudantes (as chaves) a outro dicionário (os valores). O dicionário mais interno relaciona cada matéria (as chaves) a uma nota (os valores).

```
class BySubjectGradebook(object):
    def __init__(self):
        self._grades = {}
    def add_student(self, name):
        self._grades[name] = {}
```

Parece bastante direto. Todavia, os métodos report_grade and average_grade ganharão uma pitada de complexidade para lidar com o dicionário tridimensional, mas, por enquanto, ainda é possível entendê-lo.

```
    def report_grade(self, name, subject, grade):
```

```
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
        total, count = 0, 0
        for grades in by_subject.values():
            total += sum(grades)
            count += len(grades)
        return total / count
```

O uso da classe ainda é simples.

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
```

Imagine que os requisitos mudem outra vez. Agora, as provas bimestrais e finais são mais importantes que os trabalhos entregáveis desenvolvidos em sala. Por isso, é preciso manter um registro do peso de cada nota porque a média final deve ser ponderada. Uma maneira de implementar esse recurso é alterar o dicionário interno. Em vez de associar as disciplinas (chaves) às notas (valores), poderíamos usar uma tupla no formato (score, weight) – em português (nota, peso) – como valor.

```
class WeightedGradebook(object):
    # ...
    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))
```

Embora as alterações em report_grade pareçam simples – o valor agora é uma tupla – o método average_grade precisa agora de um laço dentro de outro laço, o

que é absolutamente difícil de ler.

```
    def average_grade(self, name):
        by_subject = self._grades[name]
        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
            for score, weight in scores:
                # ...
        return score_sum / score_count
```

Usar a classe também fica mais difícil, pois não fica claro o que significam todos esses números nos argumentos posicionais.

```
book.report_grade('Albert Einstein', 'Math', 80, 0.10)
```

Quando esse tipo de complexidade começa a aparecer, é hora de abandonar os dicionários e as tuplas e migrar para uma hierarquia de classes.

De início, não sabíamos que seria necessário calcular médias ponderadas, portanto não valia a pena enfrentar a complexidade de desenvolver uma classe auxiliar a mais. Os dicionários e as tuplas, tipos nativos do Python, tornam fácil seguir adiante a cada novo requisito, adicionando camada sobre camada de registros interdependentes. Porém, devemos evitar esse caminho quando houver mais de um nível de aninhamento (ou seja, evite dicionários que contenham dicionários). Esse expediente torna a leitura do código muito difícil para outros programadores e é um pesadelo para manter no futuro – uma armadilha mortal!

Assim que perceber que os registros estão ficando complicados, desmembre-os em classes diferentes. Com isso, é possível definir interfaces bem-feitas que encapsulam melhor os dados, e também permite criar uma camada de abstração entre as interfaces e a implementação concreta.

Refatorando o código para incluir classes

Podemos começar migrando para o esquema de classes pela base da árvore de dependências: uma nota individual. Uma classe parece ser algo sofisticado demais para uma informação tão simples. Uma tupla, por outro lado, parece ser bastante apropriada porque as notas são imutáveis. No exemplo a seguir, uma tupla (score, weight) é empregada para manter o registro de notas em uma lista:

```
grades = []
grades.append((95, 0.45))
# ...
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

O problema com as tuplas comuns é que elas são posicionais. Quando associamos mais informação a uma nota, como, por exemplo, anotações do professor a respeito do trabalho a que aquela nota pertence, é preciso reescrever todos os pontos do sistema que usam aquela tupla para que estejam cientes de que agora há três itens presentes em vez de dois. No exemplo a seguir, usamos o caractere _ (o underscore como nome de variável, uma convenção do Python para variáveis que jamais serão usadas) para capturar a terceira entidade em uma tupla e simplesmente ignorá-la:

```
grades = []
grades.append((95, 0.45, 'Great job'))
# ...
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

Esse padrão de estender tuplas cada vez mais é semelhante a acrescentar camadas progressivamente mais profundas em dicionários. Assim que perceber que suas tuplas têm mais de dois elementos, é hora de considerar outro estilo de implementação.

O tipo namedtuple (tupla identificada) no módulo collections faz exatamente o que queremos: definir classes de dados minúsculas e imutáveis.

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight'))
```

Essas classes podem ser montadas tanto com argumentos posicionais como por palavra-chave. Os campos são acessíveis por atributos identificados. O fato de os atributos terem um identificador torna mais fácil migrar de uma namedtuple para uma classe mais complexa, criada por você no futuro, caso os requisitos mudem novamente e seja necessário adicionar comportamentos aos contêineres de dados

existentes.

Limitações da namedtuple

Embora seja útil em muitas circunstâncias, é importante identificar as situações em que uma namedtuple pode causar mais dano que benefício.

- Não é possível especificar valores default para argumentos em classes namedtuple. Isso as torna inadministráveis quando os dados têm muitas propriedades opcionais. Se perceber que está usando mais que um punhado de atributos, definir sua própria classe pode ser uma escolha melhor.
- Os valores de atributo das instâncias de namedtuple ainda estão acessíveis por índices numéricos e iteradores. Especialmente em uma API externa, isso pode levar a um uso não intencional que torna difícil a migração para uma classe de verdade no futuro. Se você não tem controle sobre quem vai usar suas instâncias de namedtuples, é melhor definir desde já sua própria classe.

A seguir, você pode escrever uma classe para representar uma única disciplina que contenha um conjunto de notas.

```
class Subject(object):
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

Depois, escreva uma classe para representar um conjunto de disciplinas sendo cursadas por um único aluno.

```
class Student(object):
```

```
    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

Por fim, escreva um contêiner para todos os estudantes dinamicamente escolhidos pelos seus nomes.

```
class Gradebook(object):
    def __init__(self):
        self._students = {}

    def student(self, name):
        if name not in self._students:
            self._students[name] = Student()
        return self._students[name]
```

O número de linhas de código dessas classes é quase o dobro da implementação anterior, mas é muito mais fácil de ler. O exemplo a seguir, que faz uso das classes, é ainda mais claro e com possibilidade de ser estendido.

```
book = Gradebook()
albert = book.student('Albert Einstein')
math = albert.subject('Math')
math.report_grade(80, 0.10)
# ...
```

```
print(albert.average_grade())
>>>
81.5
```

Se for necessário, você pode escrever métodos retrocompatíveis que auxiliem a migração do modelo anterior, no estilo API, para a nova hierarquia de objetos.

Lembre-se

- Evite criar dicionários com valores que sejam outros dicionários ou tuplas muito longas.
- Use namedtuple para criar contêineres de dados leves e imutáveis antes de precisar da flexibilidade de uma classe completa.
- Migre seu código de registro para uma hierarquia de classes auxiliares quando seus dicionários internos de estado começarem a ficar muito complicados.

Item 23: Aceite funções para interfaces simples em vez de classes

Muitas as APIs nativas do Python permitem personalizar seu comportamento pela passagem de uma função. Esses *ganchos* (*hooks*) são usados pelas APIs para chamar o código do usuário enquanto executam (operação conhecida como *callback*). Por exemplo, o método sort do tipo list aceita o argumento opcional key que é usado para determinar cada valor do índice para classificação. No exemplo a seguir, uma lista de nomes é ordenada com base em seus comprimentos pela passagem de uma expressão lambda como o gancho key:

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)
>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

Em outras linguagens, podemos esperar que os ganchos sejam definidos por uma classe abstrata. No Python, muitos ganchos são apenas unções sem estado (stateless) com argumentos e valores de retorno bem definidos. As funções são

ideais para os ganchos porque são mais fáceis de descrever e mais simples de definir que as classes. As funções trabalham como ganchos porque, em Python, as funções são objetos de *primeira-classe*: métodos e funções podem ser passados e referenciados a outros objetos como qualquer outro valor da linguagem.

Por exemplo, digamos que se queira personalizar o comportamento da classe defaultdict (consulte o Item 46: “Use algoritmos e estruturas de dados nativos” para mais informações). Essa estrutura de dados permite fornecer uma função que será chamada toda vez que uma chave faltante seja acessada. A função deve retornar o valor-padrão que a chave faltante deva ter no dicionário. No exemplo a seguir, é definido um gancho que registra no log sempre que uma chave não puder ser encontrada e retorna o valor default 0 para ela:

```
def log_missing():
    print('Key added')
    return 0
```

Dado um dicionário inicial e um conjunto de incrementos, podemos usar a função log_missing para rodar e imprimir duas vezes (para as chaves 'red' e 'orange').

```
current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]
result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After: ', dict(result))
```

```
>>>
Before: {'green': 12, 'blue': 3}
Key added
Key added
```

```
After:  {'orange': 9, 'green': 12, 'blue': 20, 'red': 5}
```

Disponibilizar funções como log_missing torna as APIs fáceis de construir e testar porque separam o comportamento determinístico dos efeitos colaterais. Por exemplo, digamos que agora se queira o gancho do valor default passado para a classe defaultdict para que se possa contar o número total de chaves não encontradas. Uma maneira de se fazer isso é usar um closure que armazene seu estado (stateful – consulte o Item 15: “Saiba como os closures interagem com os escopos das variáveis” para mais informações). No exemplo a seguir, definimos uma função auxiliar que usa um closure como esse como gancho de valor default:

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count  # Closure que guarda seu estado (stateful)
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount

    return result, added_count
```

Essa função, quando executada, produz o resultado esperado (2), mesmo que defaultdict não tenha a mínima ideia de que o gancho missing guarde seu estado. Esse é um dos grandes benefícios de aceitar funções simples como interfaces. É fácil adicionar funcionalidade mais tarde simplesmente escondendo algum estado dentro de um closure.

```
result, count = increment_with_report(current, increments)
assert count == 2
```

O problema ao definir um closure para ganchos stateful é que é mais difícil de ler que no exemplo da função sem estado (stateless). Outra maneira de abordar o problema poderia ser a definição de uma pequena classe que encapsule o estado

que queremos rastrear.

```
class CountMissing(object):
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

Em outras linguagens, seria de se esperar que a classe defaultdict agora tivesse que ser modificada para acomodar a interface de CountMissing. Em Python, todavia, graças às suas funções de primeira classe, é possível referenciar o método CountMissing.missing diretamente em um objeto e passá-lo à classe defaultdict como gancho de valor default. É trivial fazer com que o método satisfaça à função de interface.

```
counter = CountMissing()
result = defaultdict(counter.missing, current)  # Referencia o método

for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

Usar uma classe auxiliar como essa para oferecer o comportamento de um closure que guarda seu estado é mais limpo e organizado que a função increment_with_report descrita anteriormente. Entretanto, vista isoladamente, não fica muito óbvio qual a razão de ser da classe CountMissing. Quem constrói um objeto CountMissing? Quem chama o método missing? A classe vai precisar que outros métodos públicos sejam adicionados no futuro? Até que se veja como ela é usada pelo defaultdict, a classe é um mistério.

Para esclarecer a situação, o Python permite que as classes definam o método especial __call__. Esse método permite que um objeto seja chamado da mesma maneira que uma função. Com isso, a função nativa callable retorna True para qualquer instância dessa classe.

```
class BetterCountMissing(object):
    def __init__(self):
```

```
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
counter()
assert callable(counter)
```

No exemplo a seguir, usamos uma instância de BetterCountMissing como gancho de valor default para um defaultdict de modo a registrar a quantidade de chaves não encontradas que forem sendo adicionadas:

```
counter = BetterCountMissing()
result = defaultdict(counter, current)  # Depende de __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

Isso é muito mais claro que o exemplo anterior de CountMissing.missing. O método __call__ indica que as instâncias da classe podem ser usadas em qualquer lugar onde uma função seria apropriada (como ganchos de API) e redireciona as pessoas que examinarem o código no futuro para o ponto de entrada responsável pelo comportamento primário da classe. Além disso, é uma dica muito forte de que o objetivo da classe é agir como um closure que armazena seu estado (stateful).

O melhor de tudo é que defaultdict não tem como ver o que acontece em seu código quando se usa __call__. Tudo o que defaultdict requer é uma função para o gancho de valor default. O Python oferece muitas maneiras diferentes de satisfazer uma interface funcional simples, dependendo do que for necessário realizar.

Lembre-se

- Em vez de definir e instanciar classes, as funções são normalmente tudo o que precisamos para interfaces simples entre componentes do Python.

- Referências a funções e métodos em Python são de primeira classe, o que significa poderem ser usadas em expressões como qualquer outro tipo de dados.
- O método especial __call__ permite chamar instâncias de classes como se fossem funções comuns do Python.
- Quando precisar de uma função para armazenar estados, considere definir uma classe que ofereça o método __call__ em vez de definir um closure que guarde estado (consulte o Item 15: “Saiba como os closures interagem com os escopos das variáveis”).

Item 24: Use o polimorfismo de @classmethod para construir objetos genericamente

No Python, não são apenas os objetos que suportam polimorfismo: as próprias classes também podem sofrer mutação. O que isso significa, e para que isso serve?

O polimorfismo é uma maneira de múltiplas classes em uma hierarquia implementar suas próprias versões únicas de um mesmo método. Isso permite que muitas classes preencham a mesma interface ou classe abstrata básica ao mesmo tempo em que oferece funcionalidade diferente (consulte o Item 28: “Herde da classe collections.abc para obter tipos de contêiner personalizados” para ver um exemplo).

Por exemplo, digamos que se queira escrever uma implementação de MapReduce e se quer uma classe comum para representar os dados de entrada. No exemplo a seguir, definimos uma classe como essa com um método read que deve ser definido por subclasses:

```
class InputData(object):
    def read(self):
        raise NotImplementedError
```

No código seguinte, definimos uma subclasse concreta de InputData que lê os dados de um arquivo no disco:

```
class PathInputData(InputData):
    def __init__(self, path):
```

```
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

Podemos ter qualquer quantidade de subclasses InputData como a PathInputData mostrada, e cada uma pode implementar a interface-padrão para o método read que retorne os dados em bytes que devam ser processados. Outras subclasses de InputData podem ler da rede, descomprimir dados de forma transparente etc.

O trecho a seguir é uma interface abstrata similar para o MapReduce chamada Worker que consume os dados de entrada de forma padrão.

```
class Worker(object):
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

No exemplo a seguir, definimos uma subclasse concreta de Worker para implementar uma função específica de MapReduce que queremos aplicar: um simples contador de quebras de linha.

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

Pode parecer que essa implementação está indo às mil maravilhas, mas

chegamos ao grande problema nisso tudo. O que é que junta todas essas peças? Temos um belo conjunto de classes com abstrações e classes razoavelmente bem-feitas, mas que só são úteis quando os objetos são instanciados. Quem é o responsável por construir os objetos e orquestrar o MapReduce?

A maneira mais simples é construir manualmente o programa principal e conectar os objetos usando um punhado de funções auxiliares. No exemplo a seguir, listamos o conteúdo de uma pasta e instanciamos PathInputData para cada arquivo que ela contém:

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

Depois, criamos as instâncias de LineCountWorker usando as instâncias de InputData devolvidas por generate_inputs.

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

Executamos essas instâncias de Worker abrindo em leque o passo map em múltiplas threads (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”). Depois, chamamos reduce repetidamente para combinar os resultados em um único valor final.

```
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
    return first.result
```

Finalmente, conecto todas as peças juntas em uma função que executa cada um

dos passos.

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

Rodar essa função em um conjunto de arquivos de teste funciona às mil maravilhas.

```
from tempfile import TemporaryDirectory

def write_test_files(tmpdir):
    # ...

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

print('There are', result, 'lines')
>>>
There are 4360 lines
```

Se é tudo maravilhoso, onde está o problema? O grande senão é que a função mapreduce não é nem um pouco genérica. Se quisermos escrever outra subclasse de InputData ou Worker, precisaríamos reescrever as funções generate_inputs, create_workers e mapreduce para que acompanhassem a mudança.

Esse problema revela a necessidade mais básica de se ter uma maneira genérica de construir objetos. Em outras linguagens, resolveríamos o problema facilmente com um construtor de polimorfismo, bastando obrigar cada subclasse de InputData a providenciar um construtor especial que possa ser usado genericamente pelos métodos auxiliares que orquestram o funcionamento do programa principal MapReduce. Em Python, infelizmente, só temos um único método construtor, o __init__. É totalmente fora de propósito exigir que cada subclasse de InputData tenha um construtor compatível.

A melhor maneira de resolver o problema é empregar o polimorfismo de @classmethod. Funciona da mesma maneira que o polimorfismo de método de

instância que usamos em InputData.read, mas com a vantagem serem válidos para toda uma classe em vez de a cada um dos objetos individuais dela.

Implementando essa ideia nas classes de MapReduce, vemos no exemplo a seguir como estender a classe InputData com um método genérico de classe que será responsável pela criação das instâncias de InputData usando uma interface comum a todas:

```
class GenericInputData(object):
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

Fizemos generate_inputs aceitar um dicionário com um conjunto de parâmetros de configuração cuja interpretação é de responsabilidade da subclasse concreta InputData. No exemplo a seguir, usou-se config para encontrar a pasta da qual listaremos os arquivos de entrada:

```
class PathInputData(GenericInputData):
    # ...
    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

Da mesma forma, podemos agregar a função auxiliar create_workers como parte da classe GenericWorker. No exemplo a seguir, o parâmetro input_class, que deve ser uma subclasse de GenericInputData, gera os dados necessários de entrada. As instâncias da subclasse concreta GenericWorker são criadas usando cls() como um construtor genérico.

```
class GenericWorker(object):
```

```
    # ...
    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

Observe que a chamada a input_class.generate_inputs no código anterior é a demonstração do polimorfismo de classe. Podemos ver também como create_workers, ao chamar cls, oferece uma maneira alternativa de construir objetos GenericWorker sem usar o método __init__ diretamente.

O efeito em minha classe concreta GenericWorker não é nada complicado, apenas uma mudança em sua classe ancestral.

```
class LineCountWorker(GenericWorker):
    # ...
```

Por fim, podemos reescrever a função mapreduce de forma a torná-la completamente genérica.

```
def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

Executar o novo Worker em um conjunto de arquivos de teste produz os mesmos resultados da implementação antiga. A diferença é que a função mapreduce requer mais parâmetros para que possa operar genericamente.

```
with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
```

```
result = mapreduce(LineCountWorker, PathInputData, config)
```

Agora podemos escrever outras classes GenericInputData GenericWorker à vontade e não será mais preciso reescrever nenhum código do programa principal.

Lembre-se

- O Python suporta apenas um construtor por classe, o método __init__.
- Use @classmethod para definir construtores alternativos para suas classes.
- Use polimorfismo de método de classe para implementar uma maneira genérica de construir e conectar subclasses concretas.

Item 25: Inicialize classes ancestrais com super

A velha maneira de inicializar uma classe-mãe a partir de uma classe-filha era chamar diretamente o método __init__ da classe-mãe com a instância-filha.

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value
class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

Funciona para hierarquias simples, mas começa a apresentar problemas em muitos casos.

Se sua classe for afetada por múltiplas heranças (algo a ser evitado, aliás! Consulte o Item 26: “Use heranças múltiplas apenas para classes utilitárias mix-in”), chamar o método __init__ da superclasse diretamente pode causar comportamento imprevisível.

Um dos problemas com isso é que a ordem de chamada a __init__ não está especificada em todas as subclasses. Por exemplo, o código a seguir define duas classes-mãe que operam no campo value da instância:

```
class TimesTwo(object):
    def __init__(self):
        self.value *= 2
```

```
class PlusFive(object):
    def __init__(self):
        self.value += 5
```

Esta classe define suas classes-mãe em determinada ordem.

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

Ao construir um objeto com essa classe, o resultado produzido obedece à ordem definida de classes-mãe.

```
foo = OneWay(5)
print('First ordering is (5 * 2) + 5 =', foo.value)
>>>
First ordering is (5 * 2) + 5 = 15
```

Esta outra classe define as mesmas classes-mãe, mas em ordem diferente:

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

Entretanto, as chamadas aos construtores PlusFive.__init__ e TimesTwo.__init__ da classe mãe estão na mesma ordem que antes, fazendo com que o comportamento desta segunda classe não combine com a ordem da classe-mãe em sua definição.

```
bar = AnotherWay(5)
print('Second ordering still is', bar.value)
>>>
Second ordering still is 15
```

Outro problema ocorre com a chamada herança losangular (diamond

inheritance). Uma herança losangular acontece quando uma subclasse obtém herança a partir de duas classes separadas, mas essas duas classes têm a mesma superclasse em algum lugar da hierarquia. O caminho que liga hierarquicamente as quatro classes forma um losango (ou diamante, como é conhecido na língua inglesa). A herança losangular faz com que o método __init__ da superclasse que as classes compartilham seja executado mais de uma vez, causando comportamento inesperado. Por exemplo, o código a seguir define duas classes-filha herdeiras de MyBaseClass.

```
class TimesFive(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 5

class PlusTwo(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 2
```

Então, é definida uma classe-filha que herda das duas classes, deixando MyBaseClass no vértice superior do losango.

```
class ThisWay(TimesFive, PlusTwo):
    def __init__(self, value):
        TimesFive.__init__(self, value)
        PlusTwo.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 5) + 2 = 27 but is', foo.value)
>>>
Should be (5 * 5) + 2 = 27 but is 7
```

A saída é 27 porque (5 * 5) + 2 = 27. Porém, a chamada ao construtor da segunda classe-mãe, PlusTwo.__init__, causa a redefinição do valor armazenado em self.value para 5 no momento em que MyBaseClass.__init__ é chamada pela segunda vez.

Para resolver o problema, o Python 2.2 introduziu a função nativa super e

definiu a ordem de solução de métodos (Method Resolution Order – MRO). A MRO padroniza a ordem em que as superclasses são inicializadas (e.g., por exemplo, mais profundas primeiro, da esquerda para a direita). Ela também assegura que superclasses envolvidas em arquiteturas losangulares sejam executadas apenas uma vez.

No exemplo a seguir, outra hierarquia de classes losangular foi criada, mas desta vez usando super (no estilo do Python 2) para inicializar a classe ancestral:

```
# Python 2
class TimesFiveCorrect(MyBaseClass):
    def __init__(self, value):
        super(TimesFiveCorrect, self).__init__(value)
        self.value *= 5

class PlusTwoCorrect(MyBaseClass):
    def __init__(self, value):
        super(PlusTwoCorrect, self).__init__(value)
        self.value += 2
```

Agora a parte superior do losango, a função MyBaseClass.__init__, só é executada uma única vez. As outras classes-mãe são executadas na ordem especificada pelo comando class.

```
# Python 2
class GoodWay(TimesFiveCorrect, PlusTwoCorrect):
    def __init__(self, value):
        super(GoodWay, self).__init__(value)

foo = GoodWay(5)
print 'Should be 5 * (5 + 2) = 35 and is', foo.value
>>>
Should be 5 * (5 + 2) = 35 and is 35
```

A ordem pode parecer invertida à primeira vista. TimesFiveCorrect.__init__ não deveria ser a primeira a executar? O resultado não deveria ser (5 * 5) + 2 = 27? A resposta é não. Essa ordem obedece ao que a MRO definiu para esta classe. A ordenação pela MRO está disponível em um método da classe chamado mro.

```
from pprint import pprint
pprint(GoodWay.mro())
>>>
[<class '__main__.GoodWay'>,
<class '__main__.TimesFiveCorrect'>,
<class '__main__.PlusTwoCorrect'>,
<class '__main__.MyBaseClass'>,
<class 'object'>]
```

Quando eu chamo GoodWay(5), este por sua vez chama TimesFiveCorrect.__init__, que chama PlusTwoCorrect.__init__, que chama MyBaseClass.__init__. Uma vez que as chamadas atinjam o topo do losango, todos os métodos de inicialização executam o que foram programados para fazer na ordem inversa com que suas funções __init__ foram chamadas, MyBaseClass.__init__ atribui a value o valor 5. PlusTwoCorrect.__init__ soma 2 para fazer com que value seja igual a 7. TimesFiveCorrect.__init__ multiplica-o por 5 para que value assuma o valor 35.

A função nativa super funciona muito bem, mas tem dois problemas bastante incômodos no Python 2:

- Sua sintaxe é um tanto prolixa. É preciso especificar a classe em que se está, o objeto self, o nome do método (normalmente __init__) e todos os argumentos. Essa construção pode ser confusa para programadores iniciantes em Python.
- É preciso especificar a classe atual pelo nome ao chamar super. Se por algum motivo for necessário mudar o nome da classe – algo extremamente comum quando se está otimizando a hierarquia de classes – também se faz necessário atualizar todas as chamadas a super.

Felizmente, o Python 3 acaba com esses inconvenientes, pois as chamadas a super sem nenhum argumento equivalem a chamar super com __class__ e self automaticamente especificados. No Python 3, você deve sempre usar super porque é limpo, conciso e sempre faz a coisa certa.

```
class Explicit(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value * 2)
```

```
class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

assert Explicit(10).value == Implicit(10).value
```

O código anterior funciona porque o Python 3 permite referenciar de forma confiável a classe atual em métodos usando a variável __class__. Isso não funciona no Python 2 porque __class__ não é definida. Poderíamos pensar que o emprego de self.__class__ como argumento de super funcionaria, mas, na verdade, o artifício geraria um erro por conta da maneira com que super foi implementada em Python 2.

Lembre-se

- O método-padrão de solução de ordem no Python (MRO) resolve alguns problemas na inicialização de superclasses, seja em sua ordenação ou nos casos de herança losangular.
- Sempre use a função nativa super para inicializar classes ancestrais.

Item 26: Use heranças múltiplas apenas para classes utilitárias mix-in

O Python é uma linguagem orientada a objetos com recursos nativos para tornar palatáveis as situações de múltipla herança (consulte o Item 25: “Inicialize classes ancestrais com super”). Entretanto, o melhor a se fazer é evitar o emprego de herança múltipla.

Se você deseja a conveniência e o encapsulamento que advém da múltipla herança, considere em seu lugar o uso de um *mix-in*, uma minúscula classe que define unicamente um conjunto de métodos adicionais que uma classe deve oferecer. As classes mix-in não definem seus próprios atributos de instância nem precisam que seu construtor __init__ seja chamado.

Escrever um mix-in é fácil porque em Python é trivial inspecionar o estado atual de qualquer objeto, independentemente de seu tipo. A inspeção dinâmica permite escrever funcionalidade genérica uma única vez em um mix-in, que pode ser

aplicada em muitas outras classes. Os mix-ins são compostos e montados em camadas de forma a minimizar código repetitivo e maximizar sua reutilização.

Por exemplo, digamos que se queira implementar um recurso para converter um objeto qualquer do Python que tenha uma representação em memória em um dicionário pronto para ser serializado. Por que não escrever essa funcionalidade de forma genérica para que possa ser usada em todas as classes?

No exemplo a seguir, definimos um mix-in de exemplo que obtém esse resultado com um novo método público que pode ser inserido em qualquer classe, bastando que essa classe herde do mix-in esse método.

```
class ToDictMixin(object):
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

Os detalhes de implementação são simples e diretos e se apoiam em um acesso dinâmico a atributos usando hasattr, inspeção dinâmica de tipos com isinstance e acesso ao dicionário da instância, __dict__.

```
    def _traverse_dict(self, instance_dict):
        output = {}
        for key, value in instance_dict.items():
            output[key] = self._traverse(key, value)
        return output

    def _traverse(self, key, value):
        if isinstance(value, ToDictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._traverse_dict(value)
        elif isinstance(value, list):
            return [self._traverse(key, i) for i in value]
        elif hasattr(value, '__dict__'):
            return self._traverse_dict(value.__dict__)
        else:
            return value
```

No exemplo a seguir, definimos uma classe de exemplo que usa o mix-in para fazer uma representação em formato de dicionário de uma árvore binária:

```
class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

Com a classe fica muito fácil traduzir para o formato dicionário um grande número de objetos do Python.

```
tree = BinaryTree(10,
    left=BinaryTree(7, right=BinaryTree(9)),
    right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())
```

```
>>>
{'left': {'left': None,
          'right': {'left': None, 'right': None, 'value': 9},
          'value': 7},
 'right': {'left': {'left': None, 'right': None, 'value': 11},
           'right': None,
           'value': 13},
 'value': 10}
```

O aspecto mais bacana dos mix-ins é que podemos tornar plugável essa sua funcionalidade genérica, portanto os comportamentos podem ser modificados sempre que necessário. Por exemplo, o código a seguir define uma subclasse de BinaryTree que mantém uma referência à sua mãe. Essa referência circular pode fazer com que a implementação default de ToDictMixin.to_dict entre em laço infinito.

```
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent
```

A solução é sobrescrever o método ToDictMixin._traverse presente na classe BinaryTreeWithParent para que processe apenas valores relevantes, evitando ciclos encontrados pelo mix-in. No exemplo a seguir, o método _traverse é sobrescrito para que não cruze os limites da classe-mãe e simplesmente insira seu valor numérico:

```
    def _traverse(self, key, value):
        if (isinstance(value, BinaryTreeWithParent) and
                key == 'parent'):
            return value.value  # Evita laços circulares
        else:
            return super()._traverse(key, value)
```

Uma chamada a BinaryTreeWithParent.to_dict funciona sem problemas porque as propriedades que contêm a referência circular não são seguidas.

```
root = BinaryTreeWithParent(10)
root.left = BinaryTreeWithParent(7, parent=root)
root.left.right = BinaryTreeWithParent(9, parent=root.left)
print(root.to_dict())
```

```
>>>
{'left': {'left': None,
          'parent': 10,
          'right': {'left': None,
                    'parent': 7,
                    'right': None,
                    'value': 9},
          'value': 7},
 'parent': None,
 'right': None,
 'value': 10}
```

Definindo BinaryTreeWithParent._traverse, também permitimos que qualquer classe que tenha um atributo do tipo BinaryTreeWithParent possa trabalhar com ToDictMixin.

```
class NamedSubTree(ToDictMixin):
```

```
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
print(my_tree.to_dict())  # Impede laços infinitos
>>>
{'name': 'foobar',
'tree_with_parent': {'left': None,
                     'parent': 7,
                     'right': None,
                     'value': 9}}
```

Os mix-ins podem ser combinados. Por exemplo, digamos que se queira um mix-in que gere uma serialização genérica de objetos JSON para qualquer classe. Podemos fazê-lo se considerarmos que alguma classe ofereça o método to_dict (que pode ou não ser fornecido pela classe ToDictMixin).

```
class JsonMixin(object):
    @classmethod
    def from_json(cls, data):
        kwargs = json.loads(data)
        return cls(**kwargs)

    def to_json(self):
        return json.dumps(self.to_dict())
```

Observe como a classe JsonMixin define tanto métodos de instância como métodos de classe. Os mix-ins permitem introduzir ambos os tipos de comportamento. Neste exemplo, os únicos requisitos de JsonMixin são o de que a classe possua o método to_dict e que seu método __init__ aceite argumentos por palavra-chave (consulte o Item 19: “Implemente comportamento opcional usando palavras-chave como argumentos”).

Esse mix-in torna simples criar hierarquias de classes utilitárias que podem ser serializadas tanto para JSON como a partir de JSON com uma quantidade mínima de código de enchimento (clichê). Por exemplo, o código a seguir tem

uma hierarquia de classes de dados representando as partes de uma topologia de datacenter:

```
class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
    # ...

class Machine(ToDictMixin, JsonMixin):
    # ...
```

Serializar essas classes, indo e voltando do JSON, é muito simples. No exemplo a seguir, temos a confirmação de que o código pode ser serializado e desserializado a partir do formato JSON e novamente de volta ao formato JSON:

```
serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""

deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)
```

É permitido usar mix-ins como este nos casos em que a classe já havia herdado anteriormente de JsonMixin em um ponto mais alto da hierarquia. A classe resultante se comportará da mesma maneira.

Lembre-se

- Evite usar heranças múltiplas; em vez disso empregue classes mix-in, que produzem o mesmo resultado.
- Use comportamentos plugáveis no nível da instância para permitir personalização por classe sempre que as classes mix-in necessitarem.
- Combine dois ou mais mix-ins para criar funcionalidades complexas a partir de comportamentos simples.

Item 27: Prefira atributos públicos em vez de privativos

Em Python, há apenas dois tipos de visibilidade para atributos em uma classe: públicos e privativos, representados pelas palavras reservadas public e private.

```
class MyObject(object):
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

Os atributos públicos podem ser acessados por qualquer um usando o operador ponto no objeto.

```
foo = MyObject()
assert foo.public_field == 5
```

Os campos privativos são especificados precedendo o nome do atributo com dois caracteres underscore, sem espaços. Eles podem ser acessados diretamente por métodos na classe que os contém.

```
assert foo.get_private_field() == 10
```

Contudo, acessar de fora da classe esses campos privativos levanta uma exceção.

```
foo.__private_field
>>>
AttributeError: 'MyObject' object has no attribute '__private_field'
```

Os métodos de classe também têm acesso a atributos privativos porque são

declarados dentro do bloco class que os contém.

```
class MyOtherObject(object):
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field

bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

Como era de se esperar, uma subclasse não consegue acessar os campos privativos de sua classe-mãe.

```
class MyParentObject(object):
    def __init__(self):
        self.__private_field = 71

class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field

baz = MyChildObject()
baz.get_private_field()
>>>
AttributeError: 'MyChildObject' object has no attribute
➥'_MyChildObject__private_field'
```

O comportamento do atributo privativo foi implementado com uma simples transformação do nome do atributo. Quando o compilador do Python vê um acesso a um atributo privativo em métodos como MyChildObject.get_private_field, traduz __private_field para acessar _MyChildObject__private_field em seu lugar. Neste exemplo, __private_field foi definido apenas em MyParentObject.__init__, portanto o nome real do

atributo privativo é _MyParentObject__private_field. Acessar o atributo privativo da classe-mãe a partir da classe-filha falha. O motivo, bastante simples, é que o nome do atributo transformado não existe na outra classe.

Sabendo disso, o programador pode acessar facilmente os atributos privativos de qualquer classe, tanto de uma classe como externamente, sem pedir permissão.

```
assert baz._MyParentObject__private_field == 71
```

Se olharmos no dicionário de atributos do objeto, veremos que os atributos privativos são, na verdade, armazenados com os nomes da forma como aparecem depois da transformação.

```
print(baz.__dict__)
>>>
{'_MyParentObject__private_field': 71}
```

Por que então a sintaxe para atributos privativos não assegura as restrições de visibilidade como deveria? A resposta mais simples é um dos lemas mais citados do Python: “Somos todos adultos aqui e consentimos em dar liberdade uns aos outros”[1]. Os programadores de Python acreditam que os benefícios da liberdade são maiores que as desvantagens de ter a cabeça fechada.

Além disso, ter a possibilidade de criar ganchos para recursos da linguagem como o acesso a atributos (consulte o Item 32: “Use __getattr__, __getattribute__ e __setattr__ para atributos preguiçosos”) permite que qualquer um possa desfigurar bastante o funcionamento interno do objeto sempre que desejar. Todavia, se isso é permitido, qual a vantagem de o Python tentar bloquear o acesso a atributos privativos por outras vias?

Para minimizar os danos causados pelo acesso indevido às engrenagens internas de uma classe, os programadores de Python seguem à risca uma convenção de nomes definida no Guia de Estilo oficial (consulte o Item 2: “Siga o Guia de Estilo PEP 8”). Campos prefixados com um único underscore (como _protected_field) são *protected*, o que alerta os usuários externos à classe para proceder com a devida cautela.

Entretanto, muitos programadores novatos em Python usam campos provativos para indicar uma API interna que não deve ser acessada por subclasses ou externamente.

```
class MyClass(object):
```

```
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return str(self.__value)

foo = MyClass(5)
assert foo.get_value() == '5'
```

Essa não é a maneira correta de fazê-lo. Inevitavelmente alguém, incluindo você mesmo, precisará criar uma subclasse desta classe para adicionar um novo comportamento ou para contornar deficiências nos métodos existentes (como mostrado no trecho de código anterior, MyClass.get_value sempre retorna uma string). Escolher atributos privativos para isso torna desajeitadas e frágeis as modificações propostas pela subclasse e extensões. Os usuários em potencial das subclasses ainda conseguirão acessar os campos privativos quando eles realmente precisarem fazê-lo.

```
class MyIntegerSubclass(MyClass):
    def get_value(self):
        return int(self._MyClass__value)

foo = MyIntegerSubclass(5)
assert foo.get_value() == 5
```

Porém, se a hierarquia de classes mudar nos níveis mais baixos, essas classes deixarão de funcionar corretamente porque as referências privativas não são mais válidas. No exemplo a seguir, o ancestral imediato (ou seja, a mãe) da classe MyIntegerSubclass, chamada MyClass, tem uma nova classe-mãe adicionada, chamada de MyBaseClass:

```
class MyBaseClass(object):
    def __init__(self, value):
        self.__value = value
    # ...

class MyClass(MyBaseClass):
```

```
    # ...

class MyIntegerSubclass(MyClass):
    def get_value(self):
        return int(self._MyClass__value)
```

O atributo __value está associado agora à classe-mãe MyBaseClass, e não mais à classe MyClass. Isso causa uma quebra na referência à variável privativa self._MyClass__value na classe MyIntegerSubclass.

```
foo = MyIntegerSubclass(5)
foo.get_value()
>>>
AttributeError: 'MyIntegerSubclass' object has no attribute '_MyClass__value'
```

Na maioria das vezes, é melhor “errar para o lado certo” e permitir que as subclasses façam mais coisas empregando atributos protegidos. Documente muito bem cada campo protegido e explique quais são as APIs internas disponíveis para subclasses e quais devem ser deixadas em paz. Isso tanto é um alerta para outros programadores como orientação para você mesmo no futuro, quando for estender seu próprio código.

```
class MyClass(object):
    def __init__(self, value):
        # O atributo abaixo armazena o valor informado pelo usuário
        # para o objeto. Ele deve ser coercível para uma string. Uma
        # vez atribuído ao objeto o valor deve ser tratado como imutável
        self._value = value
```

O único momento em que devemos considerar seriamente usar atributos privativos é quando estamos preocupados com conflitos de nomes entre as subclasses. Esse problema ocorre quando uma classe-filha involuntariamente define um atributo que já foi definido na classe-mãe.

```
class ApiClass(object):
    def __init__(self):
        self._value = 5
```

```
    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello'  # Conflito!!!

a = Child()
print(a.get(), 'and', a._value, 'should be different')
>>>
hello and hello should be different
```

O problema é preocupante especialmente em classes que fazem parte de uma API pública. As subclasses estão fora do seu alcance, portanto não é possível refatorá-las para resolver o problema. Esse tipo de conflito é possível em especial naqueles nomes de atributos mais comumente usados (como, por exemplo, value). Para reduzir esse risco, podemos usar um atributo privativo na classe-mãe para assegurar que não haja sobreposição de nomes com as classes-filha.

```
class ApiClass(object):
    def __init__(self):
        self.__value = 5

    def get(self):
        return self.__value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello'  # OK!

a = Child()
print(a.get(), 'and', a._value, 'are different')
```

```
>>>
5 and hello are different
```

Lembre-se

- Atributos privativos não são verdadeiramente privativos, segundo o compilador do Python.
- Planeje desde o início permitir que as subclasses façam mais coisas usando suas APIs e atributos internos em vez de trancá-los para fora.
- Documente bem os campos protegidos para orientar os programadores que criarão subclasses baseadas em sua classe em vez de tentar forçar um controle de acesso com atributos privativos que jamais funcionará direito.
- O único uso verdadeiramente útil dos atributos privativos é para evitar conflitos de nome com subclasses que você não controla.

Item 28: Herde da classe collections.abc para obter tipos de contêiner personalizados

Grande parte do tempo os programadores de Python passam definindo classes que contenham dados e descrevendo como esses objetos se relacionam com os outros. Cada classe de Python é um contêiner de algum tipo, encapsulando em um mesmo objeto atributos e funcionalidade. O Python também oferece tipos nativos que funcionam como contêineres para administrar dados: listas, tuplas, conjuntos e dicionários.

Quando se está criando classes para casos de uso simples como sequências, é natural querer criar uma subclasse do tipo list do Python. Por exemplo, digamos que se queira criar seu próprio tipo de lista que inclua métodos adicionais para contar a frequência de seus membros.

```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
        counts = {}
```

```
        for item in self:
            counts.setdefault(item, 0)
            counts[item] += 1
        return counts
```

Ao criar uma subclasse list, herdamos toda a funcionalidade-padrão do objeto list e preservamos sua semântica, familiar a todo programador Python. Seus métodos adicionais implementam qualquer comportamento personalizado que se queira.

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Length is', len(foo))
foo.pop()
print('After pop:', repr(foo))
print('Frequency:', foo.frequency())
```

```
>>>
Length is 7
After pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'c': 1, 'b': 2}
```

Agora, imagine que se queira criar um objeto que se pareça e se comporte como uma list, permitindo que seja indexada, mas que não seja uma subclasse de list. Por exemplo, digamos que se queira providenciar semântica de sequência (como a de list ou tuple) para uma classe de árvore binária.

```
class BinaryNode(object):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

Como fazer uma árvore binária se comportar como uma sequência? O Python implementa seus comportamentos de contêiner com métodos de instância que têm nomes especiais. Quando acessamos um item pelo seu índice:

```
bar = [1, 2, 3]
bar[0]
```

o interpretador entende da seguinte maneira:

```
bar.__getitem__(0)
```

Para fazer a classe BinaryNode atuar como uma sequência, basta providenciar uma implementação de __getitem__ que navegue primeiro pela maior dimensão da árvore do objeto.

```
class IndexableNode(BinaryNode):
    def _search(self, count, index):
        # ...
        # Retorna (found, count)

    def __getitem__(self, index):
        found, _ = self._search(0, index)
        if not found:
            raise IndexError('Index out of range')
        return found.value
```

Depois, basta construir a árvore binária normalmente.

```
tree = IndexableNode(
    10,
    left=IndexableNode(
        5,
        left=IndexableNode(2),
        right=IndexableNode(
            6, right=IndexableNode(7))),
    right=IndexableNode(
        15, left=IndexableNode(11)))
```

Além do formato de árvore binária, também é possível acessá-la como uma list.

```
print('LRR =', tree.left.right.right.value)
print('Index 0 =', tree[0])
print('Index 1 =', tree[1])
print('11 in the tree?', 11 in tree)
print('17 in the tree?', 17 in tree)
print('Tree is', list(tree))
>>>
```

```
LRR = 7
Index 0 = 2
Index 1 = 5
11 in the tree? True
17 in the tree? False
Tree is [2, 5, 6, 7, 10, 11, 15]
```

O problema é que implementar __getitem__ não é o suficiente para obter toda a semântica que esperaríamos de uma sequência.

```
len(tree)
>>>
TypeError: object of type 'IndexableNode' has no len()
```

A função nativa len requer outro método especial chamado __len__, que necessariamente deve ser implementado em seu tipo personalizado de sequência.

```
class SequenceNode(IndexableNode):
    def __len__(self):
        _, count = self._search(0, None)
        return count

tree = SequenceNode(
    # ...
)

print('Tree has %d nodes' % len(tree))
>>>
Tree has 7 nodes
```

Infelizmente, ainda não é o bastante. Também é necessário implementar os métodos count e index, que são esperados em uma sequência que se comporte como list ou tuple. Percebemos que definir nossos próprios tipos contêineres é uma tarefa bem mais difícil do que parece.

Para evitar essa dificuldade no universo Python, o módulo nativo collections.abc define um conjunto de classes abstratas básicas que declaram interfaces para todos os métodos típicos de cada tipo contêiner. Caso uma subclasse herde as interfaces dessas classes abstratas básicas, mas não implementem os métodos

correspondentes, o módulo avisará que algo está errado.

```
from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()
>>>
TypeError: Can't instantiate abstract class BadType with
abstract methods __getitem__, __len__
```

Uma vez que todos os métodos requeridos pela classe abstrata básica sejam implementados, como fizemos anteriormente com SequenceNode, todos os métodos adicionais, como index e count, serão implementados automaticamente.

```
class BetterNode(SequenceNode, Sequence):
    pass

tree = BetterNode(
    # ...
)

print('Index of 7 is', tree.index(7))
print('Count of 10 is', tree.count(10))
>>>
Index of 7 is 3
Count of 10 is 1
```

O benefício de usar essas classes abstratas básicas é ainda maior para tipos mais complexos como Set e MutableMapping, que têm grande quantidade de métodos especiais que precisariam ser implementados manualmente para atender às convenções do Python.

Lembre-se

- Herde diretamente dos tipos contêiner do Python (como list ou dict) para

casos de uso simples.

- Tenha em mente a grande quantidade de métodos que obrigatoriamente devem existir para que se possa implementar corretamente os tipos contêiner obrigatórios.
- Faça com que seus tipos contêiner personalizados herdem das interfaces definidas em collections.abc para ter certeza de que suas classes implementam os comportamentos para todas as interfaces herdadas.

---

1 N. do T.: No original, “we are all consenting adults here”, em uma alusão à idade em que legalmente um cidadão pode tomar suas próprias decisões “para maiores”, como o consumo de bebidas ou tabaco ou o relacionamento interpessoal de qualquer tipo com outro adulto.

CAPÍTULO 4

Metaclasses e atributos

As metaclasses são frequentemente mencionadas quando se fala dos recursos do Python na mídia, mas poucos entendem o que elas realizam na prática. O nome *metaclasse* insinua um conceito acima e além das classes mundanas. Posto de forma simples, uma metaclasse permite interceptar um comando class do Python provisionar um comportamento especial cada vez que a classe for definida.

Os recursos nativos do Python para modificar dinamicamente os acessos a atributos são poderosos e envoltos em uma espessa névoa de mistério. Assim como as estruturas de código orientadas a objeto da linguagem Python, os recursos de alteração dinâmica de atributos oferecem ferramentas maravilhosas para facilitar a transição de casos simples para versões mais complicadas.

Entretanto, todo esse poder traz muitas armadilhas. Atributos dinâmicos permitem sobrepor objetos com novas versões e causar efeitos colaterais inesperados. As metaclasses podem criar comportamentos extremamente bizarros que são inacessíveis para programadores novatos. É importante seguir a *regra da menor surpresa* e somente usar esses mecanismos para implementar estruturas de código que o programador já domine completamente.

Item 29: Use atributos comuns em vez dos métodos get e set

Os programadores vindos de outras linguagens que se iniciam no Python podem querer implementar métodos explícitos de leitura (getter) ou definição (setter) em suas classes.

```
class OldResistor(object):
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
```

```
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```

Usar esses setters e getters é bastante simples e até certo ponto funciona, mas não é considerado código pythônico.

```
r0 = OldResistor(50e3)
print('Before: %5r' % r0.get_ohms())
r0.set_ohms(10e3)
print('After:  %5r' % r0.get_ohms())
```

```
>>>
Before: 50000.0
After:  10000.0
```

Esses métodos são especialmente desajeitados em operações de atualização imediata do atributo, como nesta adição em uma única linha.

```
r0.set_ohms(r0.get_ohms() + 5e3)
```

Esses métodos utilitários realmente simplificam a definição da interface para a classe, facilitando encapsular funcionalidade, forçar a validação de regras de uso e definir limites. São objetivos importantes ao projetar uma classe, pois asseguram que o código chamador não será afetado negativamente à medida que a classe evoluir com o tempo.

Em Python, todavia, e a não ser em raríssimos casos, uma classe jamais precisará da implementação explícita de métodos de setter ou getter. Em vez disso, as implementações iniciais de qualquer classe devem sempre começar com atributos públicos bastante simples.

```
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

r1 = Resistor(50e3)
```

```
r1.ohms = 10e3
```

Implementado dessa forma, uma atualização imediata do atributo (neste exemplo também uma adição) é muito mais fácil de ler por parecer algo mais natural e visualmente limpo.

```
r1.ohms += 5e3
```

Mais tarde, se for necessário implementar um comportamento especial sempre que um atributo for modificado, o código pode perfeitamente migrar para o decorador @property e seu atributo setter correspondente. No exemplo de código a seguir, definimos uma nova subclasse de Resistor, que permite variar a corrente elétrica (variável current) quando for atribuído um valor à propriedade voltage. Observe que, para que a classe funcione como esperado, os nomes do setter e do getter devem ser iguais ao da propriedade pretendida.

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0
    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

A partir de agora, quando a propriedade voltage executar o método setter voltage, a propriedade current também será modificada para que a equação (Lei de Ohm) esteja sempre equilibrada.

```
r2 = VoltageResistance(1e3)
print('Before: %5r amps' % r2.current)
r2.voltage = 10
print('After:  %5r amps' % r2.current)
>>>
```

```
Before:     0 amps
After:   0.01 amps
```

Especificar o setter em uma propriedade também permite que se faça verificação de tipos e validação dos valores passados à classe. No exemplo de código a seguir, definimos uma classe que assegura sempre valores positivos de resistência elétrica:

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError('%f ohms must be > 0' % ohms)
        self._ohms = ohms
```

Atribuir uma resistência inválida à propriedade levanta uma exceção.

```
r3 = BoundedResistance(1e3)
r3.ohms = 0
>>>
ValueError: 0.000000 ohms must be > 0
```

Uma exceção será levantada caso um valor inválido seja passado à função construtora.

```
BoundedResistance(-5)
>>>
ValueError: -5.000000 ohms must be > 0
```

Isso ocorre porque BoundedResistance.__init__ chama Resistor.__init__, que executa a atribuição self.ohms = -5. Essa atribuição provoca a chamada do método @ohms.setter da classe BoundedResistance, executando o código de

validação imediatamente antes de se iniciar a construção propriamente dita do objeto.

@property pode ser usada até mesmo para transformar em imutáveis os atributos da classe-mãe.

```
class FixedResistance(Resistor):
    # ...
    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Can't set attribute")
        self._ohms = ohms
```

Atribuir um valor à propriedade depois que o objeto tiver sido construído também levanta uma exceção.

```
r4 = FixedResistance(1e3)
r4.ohms = 2e3
>>>
AttributeError: Can't set attribute
```

A grande desvantagem de @property é que os métodos para um atributo só podem ser compartilhados por subclasses. Classes não relacionadas não podem usar a mesma implementação. Entretanto, o Python suporta também os chamados *descritores* ou *descriptors* (consulte o Item 31: “Use descritores para implementar métodos reutilizáveis de @property”) que permitem reutilização de lógica ligada a propriedades e muitos outros casos.

Por fim, quando usamos métodos de @property para implementar setters e getters, precisamos nos certificar de que o comportamento implementado não seja surpreendente. Por exemplo, jamais atribua valores com set a outros atributos em métodos que leem propriedades com get.

```
class MysteriousResistor(Resistor):
```

```
    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms
    # ...
```

O comportamento resultante é extremamente bizarro.

```
r7 = MysteriousResistor(10)
r7.current = 0.01
print('Before: %5r' % r7.voltage)
r7.ohms
print('After:  %5r' % r7.voltage)
>>>
Before:     0
After:    0.1
```

A melhor política aqui é apenas modificar estados relacionados em um objeto em métodos @property.setter. Evite a todo custo qualquer outro efeito colateral que pode ser uma surpresa para o chamador quando constrói o objeto, como a importação dinâmica de módulos, a execução de funções auxiliares muito lentas ou buscas muito pesadas no banco de dados. Os usuários da sua classe esperam que seus atributos funcionem como em qualquer outro objeto do Python: rapidamente e de forma fácil. Use métodos normais para desempenhar qualquer tarefa que seja mais complexa ou lenta.

Lembre-se

- Defina novas interfaces de classe usando atributos públicos simples, e fuja de métodos com funcionalidades de set e get.
- Use @property para definir qualquer comportamento especial disparado quando os atributos do objeto são acessados, se necessário.
- Seja a regra da menor surpresa e evite efeitos colaterais esquisitos em seus métodos de @property.
- Garanta que os métodos de @property sejam rápidos; para tarefas que sejam lentas ou complexas, use métodos normais.

Item 30: Considere usar @property em vez de refatorar atributos

O decorador @property facilita que os acessos simples aos atributos de uma instância possam agir de forma inteligente (consulte o Item 29: “Use atributos comuns em vez dos métodos get e set”). Um uso bastante comum, mas absolutamente avançado, para @property é a transformação do que antes era um simples atributo numérico para um uma entidade que faz cálculos imediatos. Uma implementação dessas é extremamente útil porque permite migrar todo o comportamento de uma classe e criar comportamentos inteiramente novos sem ter que reescrever nenhum dos trechos de código que chamam a classe. A implementação também é um importante tampão para a evolução das interfaces da classe durante sua vida útil.

Por exemplo, digamos que se queira implementar uma rotina de limitação de quota usando a metáfora do balde furado (leaky bucket[1]) empregando apenas objetos simples do Python. No código de exemplo a seguir, a classe Bucket representa quanto da quota ainda pode ser utilizada e a duração na qual a quota estará disponível:

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return 'Bucket(quota=%d)' % self.quota
```

O algoritmo leaky bucket garante que, sempre que o balde estiver completamente cheio, a quantidade de quota disponível não é rolada de um período para o próximo.

```
def fill(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
```

```
    bucket.quota += amount
```

Toda vez que o consumidor de quota quiser fazer alguma coisa, o algoritmo deve primeiro garantir que pode debitar do espaço restante na quota a quantidade necessária de armazenamento.

```
def deduct(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

Para usar esta classe, primeiro enchemos o balde até a boca.

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)
>>>
Bucket(quota=100)
```

Depois, debitamos a quota necessária.

```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print(bucket)
>>>
Had 99 quota
Bucket(quota=1)
```

Até que chega o momento em que não conseguimos mais fazer isso porque estamos tentando debitar mais quota do que o disponível. Neste caso, o nível de quota no balde permanece inalterado.

```
if deduct(bucket, 3):
```

```
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)
>>>
Not enough for 3 quota
Bucket(quota=1)
```

O problema com essa implementação é que eu nunca sei qual o nível de quota inicial do balde. A quota é debitada no decorrer do período até que atinja o valor zero. Nesse ponto, deduct sempre retornará False. Quando isso acontece, seria útil saber se os chamadores de deduct estão sendo bloqueados porque o Bucket esgotou sua quota ou se o Bucket nunca teve quota desde o início.

Para aprimorar a funcionalidade, podemos modificar a classe para que registre a quota máxima atribuída no período (max_quota) e também a quota consumida (quota_consumed) no mesmo período.

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return ('Bucket(max_quota=%d, quota_consumed=%d)' %
                (self.max_quota, self.quota_consumed))
```

Usamos um método de @property para computar o nível de quota atual em tempo real usando esses novos atributos.

```
    @property
    def quota(self):
        return self.max_quota - self.quota_consumed
```

Quando é atribuído um valor à propriedade quota, é preciso uma ação especial para a interface atual seja a mesma usada por fill e deduct.

```
    @quota.setter
    def quota(self, amount):
        delta = self.max_quota - amount
        if amount == 0:
            # Quota sendo reiniciada para um novo período
            self.quota_consumed = 0
            self.max_quota = 0
        elif delta < 0:
            # Quota sendo preenchida para um novo período
            assert self.quota_consumed == 0
            self.max_quota = amount
        else:
            # Quota sendo consumida durante o período
            assert self.max_quota >= self.quota_consumed
            self.quota_consumed += delta
```

Executando o programa de demonstração novamente, o mesmo dos exemplos anteriores, obtemos os mesmos resultados.

```
bucket = Bucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)

if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')

print('Now', bucket)

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
```

```
print('Still', bucket)
>>>
Initial Bucket(max_quota=0, quota_consumed=0)
Filled Bucket(max_quota=100, quota_consumed=0)
Had 99 quota
Now Bucket(max_quota=100, quota_consumed=99)
Not enough for 3 quota
Still Bucket(max_quota=100, quota_consumed=99)
```

A melhor parte é que o código usando Bucket.quota não precisa ser modificado ou saber que a classe foi modificada. O novo modo de uso de Bucket faz a coisa certa sozinho e acessa max_quota e quota_consumed diretamente.

Eu gosto especialmente de @property porque permite fazer progressos incrementais em direção a um modelo de dados melhor. Examinando o código do exemplo anterior, Bucket, possivelmente o leitor tenha pensado consigo mesmo: “fill e deduct deveriam ter sido implementados como método de instância, para começar!” Embora você provavelmente esteja certo (consulte o Item 22: “Prefira classes auxiliares em vez de administrar registros complexos com dicionários e tuplas”), na prática existem situações em que os objetos são inicialmente implementados com interfaces definidas de forma muito ruim ou funcionam como contêineres de dados sem nenhuma inteligência. Isso acontece à medida que o código cresce ao longo do tempo, a quantidade de níveis de escopo fica maior, outros programadores contribuem com código sem que ninguém pense na higiene a longo prazo, e assim por diante.

@property é uma ferramenta que ajuda a resolver problemas que encontramos no dia a dia do mundo real, mas não abuse dela! Se você perceber que está repetidamente estendendo métodos de @property, provavelmente chegou a hora de refatorar sua classe. Construir avenidas para pavimentar o projeto malfeito de seu código nunca é boa ideia.

Lembre-se

- Use @property para implementar novas funcionalidades e atributos de instâncias existentes.

- Faça alterações incrementais usando @property para obter progressivamente modelos de dados melhores.
- Considere refatorar uma classe e todos os códigos chamadores quando perceber que está usando @property demais.

Item 31: Use descritores para implementar métodos reutilizáveis de @property

O grande problema com o decorador nativo @property (consulte o Item 29: “Use atributos comuns em vez dos métodos get e set” e o Item 30: “Considere usar @property em vez de refatorar atributos”) é a reutilização de código. Os métodos que ele modifica não podem ser reutilizados por mais de um atributo dentro da mesma classe. Eles também não podem ser reutilizados por classes que não estejam na mesma “família”.

Por exemplo, digamos que se queira uma classe que valide como porcentagem uma nota recebida por um estudante em uma tarefa de casa.

```
class Homework(object):
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._grade = value
```

Por empregar @property, essa classe é fácil de usar.

```
galileo = Homework()
galileo.grade = 95
```

Digamos que também seja desejável dar ao estudante uma nota para uma mesma

prova abrangendo mais de um assunto, cada um com uma nota em separado.

```
class Exam(object):
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
```

Rapidamente, o problema começa a não ficar mais tão divertido. Cada seção da prova precisa de um novo @property e validação apropriada.

```
    @property
    def writing_grade(self):
        return self._writing_grade

    @writing_grade.setter
    def writing_grade(self, value):
        self._check_grade(value)
        self._writing_grade = value

    @property
    def math_grade(self):
        return self._math_grade

    @math_grade.setter
    def math_grade(self, value):
        self._check_grade(value)
        self._math_grade = value
```

Além disso, essa abordagem não é generalista. Se quisermos reutilizar esta validação de percentagem em outra coisa que não seja lição de casa ou prova, será necessário repetir inúmeras vezes, em pontos diferentes, os mesmos trechos

de código-padrão (os chamados boilerplates[2]) escritos para @property e _check_grade.

A melhor maneira de resolver esse tipo de problema em Python é usar um *descritor*, ou *descriptor*. O protocolo do descritor define como os acessos a atributos são interpretados pela linguagem. Uma classe descritora pode implementar métodos __get__ e __set__ que nos permitem reutilizar o comportamento da validação de notas sem precisar escrever nenhum clichê. Para esse propósito, os descritores são também melhores que os mix-ins (consulte o Item 26: “Use heranças múltiplas apenas para classes utilitárias mix-in”), porque elas permitem reutilizar a mesma lógica para vários atributos diferentes em uma mesma classe.

No exemplo de código a seguir, definimos uma nova classe chamada Exam com atributos de classe que são, na verdade, instâncias de Grade. A classe Grade implementa o protocolo do descritor. Antes de entender como a classe Grade funciona, é importante entender o que o Python fará quando o código acessa um desses atributos de descritor em uma instância de Exam.

```
class Grade(object):
    def __get__(*args, **kwargs):
        # ...

    def __set__(*args, **kwargs):
        # ...

class Exam(object):
    # Atributos de classe
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

Quando é atribuído um valor à propriedade...

```
exam = Exam()
exam.writing_grade = 40
```

...o Python na verdade interpretará a atribuição da seguinte maneira:

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

Quando uma propriedade é recuperada...

```
print(exam.writing_grade)
```

... o Python interpretará da seguinte maneira:

```
print(Exam.__dict__['writing_grade'].__get__(exam, Exam))
```

O que leva a esse comportamento é o método __getattribute__ do object (consulte o Item 32: “Use __getattr__, __getattribute__ e __setattr__ para atributos preguiçosos”). Resumidamente, quando uma instância de Exam não possuir um atributo chamado writing_grade, o Python usará o atributo da classe Exam em seu lugar. Se esse atributo de classe for um objeto que possua os métodos __get__ e __set__, o Python considerará que o programador deseja seguir o protocolo do descritor.

Agora que conhecemos esse comportamento, bem como a maneira como o decorador @property foi usado para implementar a validação da nota na classe Homework, o trecho de código a seguir mostra uma primeira tentativa bastante razoável de implementar o descritor Grade.

```
class Grade(object):
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._value = value
```

Infelizmente, o código está completamente errado e o comportamento resultante será errático e imprevisível. Se acessarmos mais de um atributo em uma mesma instância de Exam, tudo funciona como esperado.

```
first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
```

```
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)
>>>
Writing 82
Science 99
```

Contudo, se os mesmos atributos forem acessados ao mesmo tempo em instâncias diferentes de Exam, obteremos um comportamento inesperado.

```
second_exam = Exam()
second_exam.writing_grade = 75
print('Second', second_exam.writing_grade, 'is right')
print('First ', first_exam.writing_grade, 'is wrong')
>>>
Second 75 is right
First  75 is wrong
```

O problema é que uma única instância de Grade é compartilhada por todas as instâncias de Exam no tocante ao atributo de classe writing_grade. A instância Grade para esses atributos é construída apenas uma vez em todo o tempo de vida do programa, ou seja, no momento em que a classe Exam é definida e não a cada vez que uma instância de Exam é criada.

Para resolver o problema, é preciso que a classe Grade mantenha um registro de seu valor para cada uma das instâncias únicas de Exam. Isso pode ser obtido armazenando o estado de cada instância em um dicionário.

```
class Grade(object):
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
```

```
            raise ValueError('Grade must be between 0 and 100')
        self._values[instance] = value
```

Essa implementação é simples e funciona bem, mas há ainda uma pegadinha: vazamento de memória. O dicionário _values armazena uma referência a cada instância de Exam que for passada para o método __set__ em todo o tempo de vida do programa. Com isso, as instâncias nunca zeram suas referências, o que impede que o coletor de lixo (garbage collector) do Python faça seu trabalho.

Para remediar a situação, pode-se empregar o módulo nativo weakref do Python, que oferece uma classe especial chamada WeakKeyDictionary para ser usada em lugar do dicionário de _values. O comportamento de WeakKeyDictionary é diferente, ela remove as instâncias de Exam de sua lista de chaves quando o runtime percebe que está armazenando a última referência restante da instância no programa. O Python fará ele mesmo o controle dos registros para você e garantirá que o dicionário _values esteja vazio quando todas as instâncias de Exam já não estiverem em uso.

```
class Grade(object):
    def __init__(self):
        self._values = WeakKeyDictionary()
    # ...
```

Com essa implementação do descritor Grade, tudo funciona como esperado.

```
class Exam(object):
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print('First ', first_exam.writing_grade, 'is right')
print('Second', second_exam.writing_grade, 'is right')
>>>
First  82 is right
Second 75 is right
```

Lembre-se

- Reutilize o comportamento e a validação dos métodos de @property definindo suas próprias classes descritoras.
- Use WeakKeyDictionary para garantir que suas classes descritoras não causem vazamentos de memória.
- Não se demore muito tentando entender como o __getattribute__ usa o protocolo do descritor para definir (setter) ou recuperar (getter) os atributos.

Item 32: Use __getattr__, __getattribute__ e __setattr__ para atributos preguiçosos

Os ganchos da linguagem Python facilitam desenvolver código genérico para "colar" todas as partes de um sistema. Por exemplo, digamos que se queira representar as linhas de seu banco de dados como objetos do Python. O banco possui um schema rodando. O código em seu sistema que usa objetos correspondes àquelas linhas deve também conhecer como o banco está organizado. Entretanto, em Python, o código que conecta os objetos ao banco de dados não precisa conhecer o schema das linhas. Pelo contrário, ele pode ser genérico.

Como isso é possível? Atributos comuns de instância, métodos @property e descritores não podem ser usados dessa maneira porque precisam ser definidos de antemão. O Python torna possível esse comportamento dinâmico por meio do método especial __getattr__. Quando uma classe define __getattr__, esse método é chamado toda vez que um atributo não for encontrado no dicionário da instância do objeto.

```
class LazyDB(object):
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        value = 'Value for %s' % name
        setattr(self, name, value)
        return value
```

No exemplo de código a seguir, acessamos uma propriedade chamada foo que não existe no objeto. Isso faz com que o Python chame o método __getattr__ definido na classe LazyDB, que altera __dict__, o dicionário da instância:

```
data = LazyDB()
print('Before:', data.__dict__)
print('foo:   ', data.foo)
print('After: ', data.__dict__)
>>>
Before: {'exists': 5}
foo:    Value for foo
After:  {'exists': 5, 'foo': 'Value for foo'}
```

No código a seguir, adicionamos um registro de eventos (log) à classe LazyDB para saber em que momento __getattr__ é chamada. Observe que usamos super().__getattr__() para obter o valor real da propriedade e assim evitar recursão infinita.

```
class LoggingLazyDB(LazyDB):
    def __getattr__(self, name):
        print('Called __getattr__(%s)' % name)
        return super().__getattr__(name)

data = LoggingLazyDB()
print('exists:', data.exists)
print('foo:   ', data.foo)
print('foo:   ', data.foo)
>>>
exists: 5
Called __getattr__(foo)
foo:    Value for foo
foo:    Value for foo
```

O atributo exists está presente no dicionário da instância, portanto __getattr__ nunca é chamado. O atributo foo não está no dicionário, portanto __getattr__ é chamado pela primeira vez. Porém, a chamada a __getattr__ para foo chama

internamente o método setattr, que preenche o valor de foo no dicionário da instância. É por isso que, na segunda vez que foo é acessado, __getattr__ não é chamada.

Este comportamento é especialmente útil para casos de uso como o acesso preguiçoso a dados que não possuam schema. O método __getattr__ roda uma primeira vez para fazer todo o trabalho sujo de carregar a propriedade. Todos os acessos subsequentes recuperam o valor agora existente.

Digamos que também seja desejável implementar transações neste sistema de banco de dados. Na próxima vez que o usuário acessar uma propriedade, queremos saber se a linha correspondente no banco de dados anda é válida e se as transações ainda estão abertas. O gancho de __getattr__ não permite que essa operação seja consumada de forma confiável porque usa o dicionário da instância do objeto como um atalho para os atributos existentes.

Para que esse caso de uso possa ser satisfeito, o Python possui outro gancho chamado __getattribute__. Esse método especial é chamado toda vez que um atributo é acessado em um objeto, mesmo nos casos em que ele *já existe* no dicionário de atributos, o que permite fazer coisas como verificar o estado global da transação em cada acesso a uma propriedade. No exemplo de código a seguir, definimos ValidatingDB para incluir uma mensagem no log cada vez que __getattribute__ for chamado.

```
class ValidatingDB(object):
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        try:
            return super().__getattribute__(name)
        except AttributeError:
            value = 'Value for %s' % name
            setattr(self, name, value)
            return value
```

```
data = ValidatingDB()
print('exists:', data.exists)
print('foo:   ', data.foo)
print('foo:   ', data.foo)
```

```
>>>
Called __getattribute__(exists)
exists: 5
Called __getattribute__(foo)
foo:    Value for foo
Called __getattribute__(foo)
foo:    Value for foo
```

Na eventualidade de uma propriedade dinamicamente acessada não existir, podemos elevar uma exceção AttributeError para provocar o comportamento-padrão de atributo ausente do Python, tanto para __getattr__ como para __getattribute__.

```
class MissingPropertyDB(object):
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError('%s is missing' % name)
        # ...

data = MissingPropertyDB()
data.bad_name
```

```
>>>
AttributeError: bad_name is missing
```

Qualquer código em Python que implemente funcionalidade genérica normalmente depende da função nativa hasattr para determinar a existência ou não de uma propriedade e de getattr, outra função nativa, para ler os valores armazenados nas propriedades. Essas funções também consultam o dicionário da instância para saber o nome do atributo antes de chamar __getattr__.

```
data = LoggingLazyDB()
print('Before:     ', data.__dict__)
```

```
print('foo exists: ', hasattr(data, 'foo'))
print('After:      ', data.__dict__)
print('foo exists: ', hasattr(data, 'foo'))
>>>
Before:      {'exists': 5}
Called __getattr__(foo)
foo exists:  True
After:       {'exists': 5, 'foo': 'Value for foo'}
foo exists:  True
```

No exemplo acima, __getattr__ só é chamado uma vez. Em contraste, as classes que implementam __getattribute__ chamam esse método todas as vezes que hasattr ou getattr são executadas contra um objeto.

```
data = ValidatingDB()
print('foo exists: ', hasattr(data, 'foo'))
print('foo exists: ', hasattr(data, 'foo'))
>>>
Called __getattribute__(foo)
foo exists:  True
Called __getattribute__(foo)
foo exists:  True
```

Digamos que se queira agora forçar o envio de dados de forma preguiçosa para o banco de dados. O banco deve gravar esses dados sempre que algum valor for atribuído ao objeto em Python. Podemos fazê-lo com __setattr__, um gancho de linguagem semelhante aos anteriores, que permite interceptar atribuições arbitrárias a atributos. Em vez de ler um atributo com os métodos __getattr__ e __getattribute__, não há necessidade de dois métodos separados. O método __setattr__ é sempre chamado toda vez que é atribuído um valor a um atributo em uma instância (seja diretamente ou por meio da função nativa setattr).

```
class SavingDB(object):
    def __setattr__(self, name, value):
        # Grava alguns dados no banco de dados de log
        # ...
```

```
        super().__setattr__(name, value)
```

No exemplo de código a seguir, definimos uma subclasse de log para SavingDB. Seu método __setattr__ é sempre chamado a cada atribuição de valor a um atributo.

```
class LoggingSavingDB(SavingDB):
    def __setattr__(self, name, value):
        print('Called __setattr__(%s, %r)' % (name, value))
        super().__setattr__(name, value)

data = LoggingSavingDB()
print('Before: ', data.__dict__)
data.foo = 5
print('After:  ', data.__dict__)
data.foo = 7
print('Finally:', data.__dict__)
>>>
Before:  {}
Called __setattr__(foo, 5)
After:   {'foo': 5}
Called __setattr__(foo, 7)
Finally: {'foo': 7}
```

O problema com __getattribute__ e __setattr__ é que eles são chamados em todo e qualquer acesso a um atributo do objeto, mesmo quando não queremos que isso aconteça. Por exemplo, digamos que se queira que os acessos a atributos no objeto procurem por chaves em um dicionário.

```
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = {}

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        return self._data[name]
```

Para que seja possível, é necessário acessar self._data a partir do método __getattribute__. Contudo, se tentarmos fazer isso na vida real, o Python se recusará a fazê-lo recursivamente, até que atinja o limite de sua pilha. Nesse momento, o programa será interrompido com um erro.

```
data = BrokenDictionaryDB({'foo': 3})
data.foo
>>>
Called __getattribute__(foo)
Called __getattribute__(_data)
Called __getattribute__(_data)
...
Traceback ...
RuntimeError: maximum recursion depth exceeded
```

O problema é que __getattribute__ acessa self._data, o que causa uma nova execução de __getattribute__, que acessa self._data e assim por diante, numa recursão circular. A solução é usar o método super().__getattribute__ em sua instância para obter os valores do dicionário de atributos. Dessa forma, evitamos recursão.

```
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data


    def __getattribute__(self, name):
        data_dict = super().__getattribute__('_data')
        return data_dict[name]
```

De forma semelhante, os métodos __setattr__ que modifiquem atributos em um objeto devem usar super().__setattr__.

Lembre-se

- Use __getattr__ e __setattr__ para carregar e salvar atributos de forma preguiçosa (lazy) em um objeto.
- Entenda que __getattr__ só é chamado uma única vez quando um atributo inexistente é acessado, enquanto __getattribute__ é chamado toda vez que

um atributo for acessado.

- Evite recursão infinita em __getattribute__ e __setattr__ empregando métodos de super() (ou seja, da classe object) para acessar atributos de instância diretamente.

Item 33: Valide subclasses com metaclasses

Uma das aplicações mais simples das metaclasses é conferir se uma classe foi definida corretamente. Ao construir uma hierarquia complexa de classes, podemos querer obrigar o uso de um determinado estilo, ou tornar mandatória a sobreposição de métodos, ou ainda garantir relacionamentos rigorosos entre os atributos de classe. As metaclasses permitem esses casos de uso ao oferecer uma maneira confiável para executar o código de validação a cada vez que uma nova subclasse for definida.

Na maioria das vezes, o código de validação de uma classe é executado no método __init__, no momento em que um objeto dessa classe está sendo construído (consulte o Item 28: “Herde da classe collections.abc para obter tipos de contêiner personalizados” para ver um exemplo). O uso de metaclasses para validação pode levantar exceções mais cedo no tempo de vida do programa.

Antes de saber como definir metaclasses para validar subclasses, é importante entender a ação da metaclasse para objetos comuns. Uma metaclasse é definida a partir de um tipo de dado. No caso mais comum, uma metaclasse recebe o conteúdo de comandos class em seu método __new__. No trecho de código a seguir, podemos modificar a informação de classe antes que o tipo seja realmente construído:

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print((meta, name, bases, class_dict))
        return type.__new__(meta, name, bases, class_dict)

class MyClass(object, metaclass=Meta):
    stuff = 123

    def foo(self):
```

```
        pass
```

A metaclasse tem acesso ao nome da classe, às classes ancestrais de quem ela herda e a todos os atributos de classe que forem definidos no corpo de class.

```
>>>
(<class '__main__.Meta'>,
 'MyClass',
 (<class 'object'>,),
 {'__module__': '__main__',
  '__qualname__': 'MyClass',
  'foo': <function MyClass.foo at 0x102c7dd08>,
  'stuff': 123})
```

O Python 2 possui uma sintaxe ligeiramente diferente e especifica uma metaclasse usando o atributo de classe __metaclass__. A interface Meta.__new__ é a mesma.

```
# Python 2
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        # ...
class MyClassInPython2(object):
    __metaclass__ = Meta
    # ...
```

Podemos adicionar funcionalidade ao método Meta.__new__ para validar todos os parâmetros de uma classe antes que ela seja definida. Por exemplo, digamos que se queira representar qualquer tipo de polígono com mais de três lados. Podemos fazer isso definindo uma metaclasse especial de validação e usando-a na classe base da hierarquia de classes do polígono. Observe que é importante não aplicar a mesma validação para a classe-base.

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Não valide a classe abstrata Polygon!
        if bases != (object,):
            if class_dict['sides'] < 3:
```

```
            raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(object, metaclass=ValidatePolygon):
    sides = None  # Especificado por subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3
```

Se for tentado definir um polígono com menos de três lados, a validação fará com que o comando class falhe imediatamente após o corpo de class. Isso significa que o programa não conseguirá sequer ser iniciado se uma classe dessas for definida.

```
print('Before class')
class Line(Polygon):
    print('Before sides')
    sides = 1
    print('After sides')
print('After class')
>>>
Before class
Before sides
After sides
Traceback ...
ValueError: Polygons need 3+ sides
```

Lembre-se

- Use para assegurar que as subclasses são bem formadas no momento em que são definidas, bem antes dos objetos dessa classe serem construídos.
- As metaclasses possuem sintaxes ligeiramente diferentes entre Python 2 e

Python 3.

- O método __new__ em metaclasses é executado somente após a totalidade do corpo de um comando class ser processado.

Item 34: Registre a existência de uma classe com metaclasses

Outro uso comum para as metaclasses é registrar automaticamente os tipos em um programa. O registro é útil para pesquisas reversas, quando é necessário associar um identificador simples a uma classe.

Por exemplo, digamos que se queira implementar sua própria representação serializada de um objeto em Python usando o formato JSON. É preciso encontrar uma maneira de tomar um objeto e transformá-lo em uma string JSON. No exemplo de código a seguir, fazemos isso de forma genérica definindo uma classe base que armazena os parâmetros do construtor e os converte em um dicionário JSON:

```
class Serializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})
```

Com isso, fica fácil serializar em uma string uma estrutura de dados simples e imutável como a Point2D.

```
class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point2D(%d, %d)' % (self.x, self.y)
```

```
point = Point2D(5, 3)
print('Object:    ', point)
print('Serialized:', point.serialize())
```

```
>>>
Object:     Point2D(5, 3)
Serialized: {"args": [5, 3]}
```

Na direção inversa, precisamos desserializar a string JSON e construir novamente o objeto Point2D que ela representa. No exemplo de código a seguir, definimos outra classe para desserializar os dados da classe-mãe Serializable:

```
class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])
```

Deserializable deixa muito fácil serializar e desserializar esses objetos simples e imutáveis de forma genérica.

```
class BetterPoint2D(Deserializable):
    # ...

point = BetterPoint2D(5, 3)
print('Before:    ', point)
data = point.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After:     ', after)
```

```
>>>
Before:     BetterPoint2D(5, 3)
Serialized: {"args": [5, 3]}
After:      BetterPoint2D(5, 3)
```

O problema com essa técnica é que funciona apenas quando conhecemos de antemão o tipo dos dados serializados. (por exemplo, Point2D, BetterPoint2D). O ideal seria ter um grande número de classes que serializassem todo tipo de dado para o formato JSON e apenas uma função comum que desserializasse um

dado em JSON para qualquer objeto do Python.

Uma maneira de permitir isso é incluir o nome da classe original do objeto que está sendo serializado junto com seus dados no documento JSON.

```
class BetterSerializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
        # ...
```

É preciso ainda manter um mapa que relacione os nomes das classes aos construtores apropriados para cada um desses objetos. A função genérica deserialize deve funcionar para quaisquer classes repassadas a register_class.

```
registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

Para assegurar que deserialize sempre funcione corretamente, é preciso chamar register_class para cada classe que se queira desserializar no futuro.

```
class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
```

```
        self.y = y


register_class(EvenBetterPoint2D)
```

Agora podemos desserializar qualquer string arbitrária em formato JSON sem precisar saber a classe que ela contém.

```
point = EvenBetterPoint2D(5, 3)
print('Before:    ', point)
data = point.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After:     ', after)

>>>
Before:     EvenBetterPoint2D(5, 3)
Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}
After:      EvenBetterPoint2D(5, 3)
```

O problema com essa técnica é que podemos nos esquecer de chamar register_class.

```
class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z


# Esqueci de chamar register_class! OOOOPS!
```

Isso fará com que o código apresente um erro em tempo de execução, no momento em que estiver sendo desserializado algum objeto pertencente a uma classe que esquecemos de registrar.

```
point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)

>>>
```

```
KeyError: 'Point3D'
```

Mesmo que escolhamos criar uma subclasse de BetterSerializable, não obteremos todos os seus recursos se nos esquecermos de chamar register_class depois do corpo do comando class. Essa técnica atrai muitos erros e pode ser um tanto incompreensível para os programadores novatos. A mesma omissão pode acontecer com os *decoradores de classe* no Python 3.

E se conseguíssemos agir em nome do programador, quando ele usa nossa classe BetterSerializable, e garantirmos que register_class seja chamada em qualquer caso? As metaclasses podem implementar esse comportamento ao interceptar os comandos class no momento em que as subclasses estão sendo definidas (consulte o Item 33: “Valide subclasses com metaclasses”) e permitem o registro do novo ripo imediatamente após o corpo da classe.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable, metaclass=Meta):
    pass
```

No momento em que definimos uma subclasse de RegisteredSerializable, podemos ter certeza de que a chamada a register_class acontecerá em qualquer caso e que deserialize sempre funcionará como esperado.

```
class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z


v3 = Vector3D(10, -7, 3)
print('Before:    ', v3)
data = v3.serialize()
print('Serialized:', data)
print('After:     ', deserialize(data))
```

```
>>>
Before:     Vector3D(10, -7, 3)
Serialized: {"class": "Vector3D", "args": [10, -7, 3]}
After:      Vector3D(10, -7, 3)
```

Usar metaclasses para serialização de classe garante que uma classe jamais será esquecida, desde que a árvore de heranças esteja correta. Funciona bem com serialização, como demonstrado, e também se aplica aos mapeamentos entre objetos e relacionamento (Object-Relationship Mappings – ORMs) em um banco de dados, além de gerenciadores de plugins e ganchos do sistema.

Lembre-se

- Os registros de classe são úteis para a construção de programas modulares em Python.
- As metaclasses permitem executar código de registro automaticamente a cada vez que sua classe-base seja herdada por uma subclasse em um programa.
- As metaclasses, quando usadas para o registro de classes, evitam erros ao garantir que as chamadas de registro jamais sejam esquecidas.

Item 35: Crie anotações de atributos de classe com metaclasses

Um dos recursos mais úteis das metaclasses é a possibilidade de modificar ou criar anotações em propriedades depois que uma classe for definida, mas antes de a classe ser realmente utilizada. Essa técnica é comumente empregada nos *descritores* (consulte o Item 31: “Use descritores para implementar métodos reutilizáveis de @property”) para introspecção sobre como eles estão sendo usados internamente nas classes que os contém.

Por exemplo, digamos que se queira definir uma nova classe que representa uma linha em seu banco de dados de clientes. Será necessária uma propriedade correspondente na classe para cada coluna na tabela do banco. No código a seguir, definimos uma classe descritora que associa os atributos aos nomes das colunas.

```
class Field(object):
    def __init__(self, name):
```

```
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

Com o nome da coluna armazenado no descritor Field, podemos salvar todos os estados de cada instância diretamente no dicionário da mesma instância como campos protegidos, usando as funções nativas setattr e getattr. À primeira vista, isso parece ser muito mais conveniente que montar descritores com weakref para evitar vazamento de memória.

Para definir uma classe que represente uma linha da tabela é necessário fornecer o nome da coluna para cada atributo da classe.

```
class Customer(object):
    # Atributos da classe
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

Usar a classe é muito simples. No trecho de código a seguir, podemos ver como os descritores Field modificam o dicionário da instância __dict__ como esperado:

```
foo = Customer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euclid'
print('After: ', repr(foo.first_name), foo.__dict__)
>>>
Before: '' {}
After:  'Euclid' {'_first_name': 'Euclid'}
```

Todavia, tudo isso parece redundante. Já havíamos declarado o nome do campo quando, no corpo do comando class, atribuímos o campo Customer.first_name ao objeto Field. Por que precisaria passar novamente o campo nome ('first_name' neste caso) ao construtor Field?

O problema é que a ordem das operações na definição da classe Customer é invertida em relação a como é lida, da esquerda para a direita. Primeiro, o construtor Field é chamado como Field('first_name'). Depois, o valor de retorno é atribuído a Customer.field_name. Não há maneira de fazer com que Field saiba de antemão qual atributo de classe será atribuído a ele.

Para eliminar essa redundância, podemos usar uma metaclasse. As metaclasses permitem acessar o comando class diretamente e agir assim que o corpo de class terminar de ser processado. Neste caso, podemos usar a metaclasse para atribuir Field.name a Field.internal_name no descritor automaticamente, em vez de especificar o campo manualmente uma infinidade de vezes.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

No exemplo de código a seguir, definimos uma classe-base que usa a metaclasse. Todas as classes que representam linhas em bancos de dados precisam herdar esta classe para garantir que a metaclasse seja usada por todas:

```
class DatabaseRow(object, metaclass=Meta):
    pass
```

Para trabalhar com a metaclasse, o descritor de campo fica inalterado grande parte do tempo. A única diferença é que ele não mais requer que quaisquer argumentos sejam passados para seu construtor. Em vez disso, seus atributos têm valores preenchidos pelo método Meta.__new__ do código anterior.

```
class Field(object):
    def __init__(self):
```

```
        # Esses atributos serão preenchidos pela metaclasse.
        self.name = None
        self.internal_name = None
    # ...
```

Com a metaclasse, a nova classe-base DatabaseRow e o novo descritor Field, a definição de classe para a linha na tabela no banco de dados não precisa mais depender da redundância anterior.

```
class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

O comportamento da nova classe é idêntico ao da antiga.

```
foo = BetterCustomer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euler'
print('After: ', repr(foo.first_name), foo.__dict__)
```

```
>>>
Before: " {}
After:  'Euler' {'_first_name': 'Euler'}
```

Lembre-se

- As metaclasses permitem modificar os atributos de uma classe antes de a classe ser definida por completo.
- Os descritores e metaclasses formam uma combinação poderosa para comportamento declarativo e introspecção em tempo de execução.
- Podemos evitar tanto os vazamentos de memória quanto o uso o módulo weakref empregando metaclasses e descritores trabalhando em equipe.

---

1 N. do T.: Na vida real, um balde furado funciona como um “buffer” de água. O fluxo de água que entra no balde pela boca pode ser inconstante, com diferentes quantidades de água em cada momento e até mesmo alguns períodos sem entrar água alguma. Entretanto, pelo furo localizado no fundo do balde sai um gotejamento constante. Caso o fluxo de entrada seja muito intenso, mesmo com o furo, o balde derrama. Um balde furado pode ser usado, portanto, para regular e homogeneizar o fluxo de água em determinado

sistema hidráulico. Em programação, usa-se o conceito do balde furado para homogeneizar fluxos de dados – não importa qual a taxa de entrada, nem se ela varia, na saída do "balde digital" os dados serão entregues a uma taxa constante. Se o balde "derramar", temos uma sobrecarga (overflow) que precisa ser tratada. A Wikipedia possui um artigo (em inglês) relativamente completo sobre a teoria do balde furado em computação: *https://en.wikipedia.org/wiki/Leaky_bucket*.

2 N. do T.: Em linguagem jurídica norte-americana, um texto boilerplate é o conteúdo-padrão que deve haver em todo documento legal. Por exemplo, nos termos de garantia de produtos, é comum vermos a frase (em inglês): "Esta garantia é limitada a X meses, exceto nos estados em que não é permitida a limitação da garantia". O texto é sempre esse, em qualquer termo de garantia. O termo vem dos primórdios das artes gráficas, quando um clichê de borracha (chamado em inglês de boilerplate) imprimia em todas as páginas a informação-padrão e deixava em branco as posições em que seriam impressos textos variáveis com tipos móveis. Em programação, boilerplate é todo código de abertura e fechamento de funções, classes e módulos obrigatório, que não pode ser suprimido, mas que é absolutamente igual para toda e qualquer função, classe ou módulo.

CAPÍTULO 5

Simultaneidade e paralelismo

Simultaneidade (ou concorrência, do inglês *concurrency*) é o estado operacional no qual um computador *aparentemente* faz muitas coisas ao mesmo tempo. Por exemplo, em um computador com um processador de apenas um núcleo, o sistema operacional rapidamente alterna entre os programas que estão na memória, de modo que, num determinado momento e por um período muito curto de tempo, só um programa esteja recebendo a atenção da CPU. Essa execução intercalada e em alta velocidade dos programas cria a ilusão de que estão rodando simultaneamente.

Já o *paralelismo* é a capacidade de um computador de *realmente* fazer duas ou mais coisas ao mesmo tempo. Computadores com CPUs de mais de um núcleo podem executar vários programas simultaneamente. Cada núcleo roda as instruções de um programa diferente, permitindo que todos eles avancem ao mesmo tempo em suas listas de tarefas.

Dentro de um mesmo programa, a simultaneidade é uma ferramenta que facilita a solução de certos tipos de problemas. Programas concorrentes fazem com que caminhos de execução distintos possam ser trilhados de uma maneira que aparente ser tanto simultânea como independente.

A diferença-chave entre paralelismo e simultaneidade (ou concorrência) é a *velocidade de execução*. Quando dois caminhos de execução distintos em um programa avançam em paralelo, o tempo que levam para produzir o resultado final desejado é cortado pela metade. A velocidade de execução é duas vezes maior. Por outro lado, os programas concorrentes podem rodar centenas de caminhos de execução separados, que aparentam estar em paralelo, mas que na realidade não alteram a velocidade de execução.

O Python facilita a criação de programas concorrentes, e pode também ser usado para desempenhar tarefas paralelas por meio de chamadas de sistema, subprocessos e extensões na linguagem C. Todavia, pode ser bastante difícil fazer com que programas concorrentes em Python rodem verdadeiramente em

paralelo. É importante entender como utilizar o Python da melhor maneira possível nessas duas situações sutilmente diferentes.

Item 36: Use subprocess para gerenciar processos-filho

O Python possui bibliotecas já testadas e aprovadas em campo para executar e administrar processos-filho, o que o torna uma senhora linguagem para “colar” outras ferramentas, como utilitários de linha de comando, em um mesmo sistema. Quando os scripts em shell existentes começam a ficar muito complicados, e eles ficam mesmo com o passar do tempo, reescrevê-los como um programa em Python é a escolha natural para melhorar a legibilidade e facilidade de manutenção.

Os processos-filho iniciados pelo Python são capazes de rodar em paralelo, permitindo que se use o Python para consumir todos os núcleos de CPU de sua máquina e maximizar o volume de trabalho de seus programas. Embora o Python esteja vinculado a apenas uma CPU (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”), é fácil usar o Python para iniciar e coordenar processos que usam intensivamente todos os núcleos.

O Python já teve muitas maneiras de executar subprocessos ao longo de sua vida, incluindo os comandos popen, popen2 e os.exec*. Com o Python de hoje, a melhor maneira, e também a mais simples, de gerenciar processos-filho é usando o módulo nativo subprocess.

É muito simples executar um processo-filho com o subprocess. No exemplo de código a seguir, o construtor Popen inicia o processo. O método communicate lê a saída do processo-filho e espera que ele termine.

```
proc = subprocess.Popen(
    ['echo', 'Hello from the child!'],
    stdout=subprocess.PIPE)
out, err = proc.communicate()
print(out.decode('utf-8'))
>>>
Hello from the child!
```

Os processos-filho rodam de forma independente de seu processo-pai, que é o

próprio interpretador do Python. Seu estado pode ser consultado periodicamente enquanto o Python está ocupado com outras tarefas.

```
proc = subprocess.Popen(['sleep', '0.3'])
while proc.poll() is None:
    print('Working...')
    # Algum tipo de trabalho que consome bastante tempo
    # está sendo executado aqui
    # ...

print('Exit status', proc.poll())
```

```
>>>
Working...
Working...
Exit status 0
```

Desacoplar o processo-filho de seu pai faz com que o processo-pai fique livre para iniciar e manter outros processos-filho em paralelo. Isso pode ser feito iniciando todos os processos-filho juntos, desde o começo.

```
def run_sleep(period):
    proc = subprocess.Popen(['sleep', str(period)])
    return proc

start = time()
procs = []
for _ in range(10):
    proc = run_sleep(0.1)
    procs.append(proc)
```

Mais tarde, podemos aguardar que todos os processos terminem suas operações de I/O e os encerramos com o método communicate.

```
for proc in procs:
    proc.communicate()
end = time()
print('Finished in %.3f seconds' % (end - start))
```

```
>>>
Finished in 0.117 seconds
```

Nota

Se esses processos forem iniciados em sequência, o tempo de espera seria de aproximadamente 1 segundo, e não os décimos de segundo que medimos no exemplo.

Podemos também desviar dados do programa em Python para um subprocesso e depois obter sua saída – em inglês, o termo usado para isso é pipe. Isso permite utilizar outros programas para fazer trabalho em paralelo. Por exemplo, digamos que se queira usar a ferramenta de linha de comando openssl para criptografar dados. É muito fácil iniciar o processo-filho com argumentos de linha de comando e pipes de I/O.

```
def run_openssl(data):
    env = os.environ.copy()
    env['password'] = b'\xe24U\n\xd0Ql3S\x11'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush()  # Garante que o filho receba algum dado na entrada
    return proc
```

No exemplo de código a seguir, desviamos (com pipe) bytes aleatórios para dentro da função de criptografia, mas na prática esses dados deveriam vir de um campo digitado pelo usuário, ou um manipulador de arquivos (handle), ou um socket de rede, ou qualquer outra fonte de dados do mundo real:

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_openssl(data)
    procs.append(proc)
```

Os processos-filho rodam em paralelo e consomem os dados de entrada. No

exemplo de código a seguir, esperamos que eles sejam encerrados para obter seus dados de saída:

```
for proc in procs:
    out, err = proc.communicate()
    print(out[-10:])
>>>
b'o4,G\x91\x95\xfe\xa0\xaa\xb7'
b'\x0b\x01\\\xb1\xb7\xfb\xb2C\xe1b'
b'ds\xc5\xf4;j\x1f\xd0c-'
```

Podemos também criar cadeias de processos paralelos da mesma forma que nos pipes do UNIX, conectando a saída de um processo-filho à entrada do próximo, e assim sucessivamente. O código a seguir mostra uma função que inicia um processo-filho, e este chama a ferramenta de linha de comando md5. Por sua vez, o md5 consome um fluxo de dados de entrada:

```
def run_md5(input_stdin):
    proc = subprocess.Popen(
        ['md5'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
    return proc
```

Nota

O módulo nativo `hashlib` possui uma função de `md5`, portanto executar processos dessa maneira nem sempre é necessário. O objetivo aqui é demonstrar como os subprocessos podem ser encadeados com pipe, conectando suas entradas e saídas como se fossem peças encaixáveis.

Agora podemos iniciar um conjunto de processos openssl para criptografar dados, bem como outro conjunto de processos para criar hashes em formato md5 a partir da saída já criptografada.

```
input_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(10)
```

```
    proc = run_openssl(data)
    input_procs.append(proc)
    hash_proc = run_md5(proc.stdout)
    hash_procs.append(hash_proc)
```

O I/O entre os processos-filho acontecerá automaticamente uma vez que estejam iniciados. Tudo o que precisamos fazer é esperar que mostrem na tela seus resultados.

```
for proc in input_procs:
    proc.communicate()
for proc in hash_procs:
    out, err = proc.communicate()
    print(out.strip())
```

```
>>>
b'7a1822875dcf9650a5a71e5e41e77bf3'
b'd41d8cd98f00b204e9800998ecf8427e'
b'1720f581cfdc448b6273048d42621100'
```

Se houver a possibilidade de os processos-filho nunca serem encerrados, ou se algum deles bloquear algum dos pipes de entrada ou saída, certifique-se de passar o parâmetro timeout para o método communicate. Com ele, uma exceção será elevada caso o processo-filho não responda dentro do prazo indicado e, assim, será fácil assassinar o filho malcriado.

```
proc = run_sleep(10)
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()


print('Exit status', proc.poll())
```

```
>>>
Exit status -15
```

Infelizmente, o parâmetro timeout só está disponível do Python 3.3 em diante.

Versões mais antigas nos obrigam a usar o módulo nativo select sobre proc.stdin, proc.stdout e proc.stderr para garantir o cumprimento dos timeouts de I/O.

Lembre-se

- Use o módulo subprocess para executar processos-filho e gerenciar seus fluxos de entrada e saída.
- Os processos-filho rodam em paralelo com o interpretador do Python, possibilitando maximizar o uso de CPU.
- Use o parâmetro timeout com communicate para evitar deadlocks e processos-filho travados (“zumbis”).

Item 37: Use threads para bloquear I/O e evitar paralelismo

A implementação-padrão do Python é chamada de CPython. O CPython precisa de duas etapas para executar um programa. Primeiro, analisa o texto-fonte e o compila em bytecode. Depois, executa esse bytecode usando um interpretador baseado em pilhas (stack-based interpreter). O interpretador de bytecode possui estados que precisam ser administrados e coerentes enquanto o programa em Python está sendo executado. O Python garante a coerência com um mecanismo chamado de *trava global do interpretador* (*Global Interpreter Lock – GIL*).

Essencialmente, a GIL é uma trava mutuamente excludente (mutual-exclusion lock, ou mutex), cuja função é impedir que o CPython seja afetado pelo recurso de multitarefa preemptiva do sistema operacional. Nesse tipo de multitarefa, uma thread toma o controle de um programa interrompendo ativamente a execução de outra thread. Esse tipo de interrupção pode arruinar os estados do interpretador se acontecerem em um momento inesperado. A GIL evita essas interrupções e garante que cada instrução do bytecode funcione corretamente com a implementação do CPython e seus módulos de extensão desenvolvidos em C.

A GIL tem um efeito colateral negativo muito importante. Nos programas escritos em linguagens como C++ ou Java, ter múltiplas threads significa que o programa poderia, pelo menos em tese, utilizar mais de um núcleo de CPU ao mesmo tempo. Embora o Python suporte a execução de múltiplas threads, a GIL é um fator limitante que permite o processamento de apenas uma thread por vez.

Isso significa que quando pensamos em empregar threads para implementar computação paralela e acelerar nossos programas em Python, ficaremos amargamente desapontados.

Por exemplo, digamos que se queira realizar algum trabalho intensivamente computacional no Python. Usaremos um algoritmo bastante ingênuo de fatoração de números como cobaia.

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

A fatoração de um conjunto de números, um depois do outro, toma bastante tempo.

```
numbers = [2139079, 1214759, 1516637, 1852285]
start = time()
for number in numbers:
    list(factorize(number))
end = time()
print('Took %.3f seconds' % (end - start))
>>>
Took 1.040 seconds
```

Usar múltiplas threads para esse cálculo faria muito sentido em outras linguagens, porque podemos tirar vantagem de todos os núcleos de CPU do computador. Vamos então tentar o mesmo em Python e observar o resultado. No exemplo de código a seguir, definimos uma thread no Python para fazer esse mesmo cálculo:

```
from threading import Thread


class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number
```

```
    def run(self):
        self.factors = list(factorize(self.number))
```

Depois, iniciamos uma thread para fatorar cada número em paralelo.

```
start = time()
threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Por fim, esperamos que todas as threads terminem.

```
for thread in threads:
    thread.join()
end = time()
print('Took %.3f seconds' % (end - start))
>>>
Took 1.061 seconds
```

Surpresa! O resultado levou ainda mais tempo que a implementação serial de factorize. Com uma thread por número, poderíamos esperar um desempenho um pouco menor que 4× mais rápido, e de fato isso acontece em outras linguagens. Não se chega a 4× por conta do trabalho adicional que o Python teve para criar as threads e coordená-las durante a execução. Em máquinas com apenas dois núcleos (dual-core e afins) poderíamos esperar uma melhora no desempenho de quase 2×. Contudo, jamais poderíamos esperar que o desempenho dessas threads fosse pior quando se tem mais de um núcleo de CPU para explorar. Esse pequeno exemplo demonstra o efeito nefasto do GIL em programas que rodem sobre o interpretador-padrão CPython.

Existem maneiras de fazer o CPython usar mais núcleos, mas elas não funcionam com a classe Thread padrão (consulte o Item 41: “Considere usar concurrent.futures para obter paralelismo real”) e por isso uma implementação dessas precisaria de um esforço substancial por parte do programador. Sabendo dessas limitações, poderíamos honestamente questionar: afinal, o Python suporta mesmo esse negócio de threads? Há duas boas razões para tal.

Primeiro, muitas threads fazem com que nosso programa pareça fazer muitas

coisas ao mesmo tempo. Implementar você mesmo um subsistema próprio para administrar tarefas simultâneas é como aquele número de circo em que o artista equilibra sozinho vários pratos sobre palitos (consulte o Item 40: “Considere usar corrotinas para rodar muitas funções simultaneamente” para ver um exemplo). Com threads, deixamos para o Python o gerenciamento de nossas funções, que assim dão a impressão de rodar em paralelo. Isso funciona porque o CPython garante um certo equilíbrio entre as threads sendo executadas, mesmo que apenas uma delas esteja realmente sendo processada num dado momento, por conta da GIL.

A segunda razão pela qual o Python suporta threads é gerenciar o bloqueio de I/O, que acontece quando o Python faz certos tipos de chamada ao sistema. As chamadas de sistema são como o programa em Python pede ao sistema operacional para interagir com o mundo externo. Os bloqueios de I/O incluem coisas como ler e escrever em arquivos, interação com redes, comunicação com dispositivos como monitores etc. As threads ajudam a lidar com os bloqueios de I/O porque isolam o programa, fazendo com que não seja afetado pelo tempo que o sistema operacional perde para responder aos seus pedidos.

Por exemplo, digamos que se queira enviar comandos para um helicóptero radiocontrolado. A comunicação com o controle remoto é pela porta serial. Usaremos uma chamada de sistema bastante lenta (select) para simular essa atividade, pois não temos um helicóptero como esse por aqui. A função pede ao sistema operacional que crie um bloqueio que dura 0,1 segundo, e depois desse tempo retorna o controle para o nosso programa, de forma semelhante ao que aconteceria se estivéssemos usando uma porta serial síncrona.

```
import select


def slow_systemcall():
    select.select([], [], [], 0.1)
```

Executar essa chamada de sistema requer uma quantidade de tempo que cresce linearmente.

```
start = time()
for _ in range(5):
    slow_systemcall()
end = time()
```

```
print('Took %.3f seconds' % (end - start))
>>>
Took 0.503 seconds
```

O problema é que enquanto a função slow_systemcall estiver rodando, nosso programa fica completamente parado, não pode continuar fazendo outra coisa. A thread principal de execução do programa é bloqueada pela chamada de sistema select. Em um caso real, essa situação é pavorosa! É preciso ser capaz de computar o próximo movimento do helicóptero ao mesmo tempo em que o comando anterior estiver sendo enviado, de outro modo ele pode bater em alguma coisa ou cair. Sempre que se encontrar em alguma situação na qual seja preciso executar paralelamente uma operação que bloqueia I/O e outra que continua o fluxo do programa, considere embutir as chamadas de sistema em threads.

No exemplo de código a seguir, rodo múltiplas chamadas à função slow_systemcall em threads separadas. Isso permitiria, por exemplo, nos comunicar com mais de uma porta serial (e mais de um helicóptero) simultaneamente, ao mesmo tempo em que a thread principal do programa poderia continuar fazendo quaisquer computações que fossem necessárias sem interrupções.

```
start = time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

Com as threads iniciadas, o código a seguir faz alguns cálculos para determinar a próxima manobra do helicóptero sem esperar que as threads de chamada de sistemas terminem.

```
def compute_helicopter_location(index):
    # ...

for i in range(5):
    compute_helicopter_location(i)
```

```
for thread in threads:
    thread.join()
end = time()
print('Took %.3f seconds' % (end - start))
>>>
Took 0.102 seconds
```

O tempo do processamento paralelo é 5× mais rápido que o tempo gasto para resolver os cálculos de forma serial. Isso mostra que as chamadas de sistema rodarão em paralelo para múltiplas threads do Python mesmo estando limitados pela GIL. A GIL impede que meu código em Python rode em paralelo, mas não tem impacto negativo nenhum sobre as chamadas de sistema. Isso funciona porque as threads em Python liberam a GIL logo antes de fazer chamadas de sistema e rearma a GIL tão logo as chamadas de sistema tenham terminado.

Existem muitas outras maneiras de lidar com operações que bloqueiam I/O além das threads, como o módulo nativo asyncio. Essas alternativas trazem benefícios muito importantes. Contudo, essas opções requerem trabalho extra para refatorar o código de forma a funcionar em um modelo diferente de execução (consulte o Item 40: “Considere usar corrotinas para rodar muitas funções simultaneamente”). Usar threads é o meio mais simples de executar operações que bloqueiam I/O em paralelo, com um mínimo de alterações no programa principal.

Lembre-se

- As threads do Python não conseguem rodar bytecode em paralelo nos sistemas com mais de um núcleo de CPU por conta da global interpreter lock (GIL).
- Mesmo com a GIL, as threads do Python são bastante úteis porque oferecem uma maneira bem fácil de executar múltiplas tarefas ao mesmo tempo.
- Use as threads do Python para fazer múltiplas chamadas de sistema em paralelo. Com isso, podemos disparar operações que bloqueiam I/O ao mesmo tempo em que o programa principal continua computando outras coisas.

Item 38: Use Lock para evitar que as threads iniciem condições de corrida nos dados

Depois de aprender sobre a global interpreter lock (GIL) (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”), muitos programadores novatos em Python inferem que podem deixar de usar as travas de exclusão mútua (mutual-exclusion locks, ou mutexes) no código. Se a GIL já está impedindo que as threads do Python sejam executadas paralelamente em mais de um núcleo de CPU, também deve atuar como trava para as estruturas de dados do programa, certo? Se fizermos alguns testes em tipos como listas e dicionários, poderíamos até supor que essa premissa seja verdadeira.

Cuidado! Essa inferência é completamente falsa! A GIL não protegerá seus dados. Embora apenas uma thread de Python esteja sendo executada em um dado momento, a operação de uma thread em estruturas de dados pode ser interrompida a qualquer tempo entre duas instruções bytecode no interpretador Python. Isso é perigoso quando mais de uma thread acessa simultaneamente o mesmo objeto. Essas interrupções podem, a qualquer momento, violar as invariantes de suas estruturas de dados, deixando seu programa em um estado de total corrupção.

Por exemplo, digamos que se queira escrever um programa que conte muitas coisas em paralelo, como, por exemplo, amostrar os níveis de luz em uma rede de sensores fotossensíveis. Se for necessário determinar o número total de amostras em função do tempo, podemos agregá-los em uma nova classe.

```
class Counter(object):
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Imagine que cada sensor tem sua própria thread de trabalho porque a ação de leitura no sensor bloqueia I/O. Depois de cada medição, a thread de trabalho incrementa um contador até um limite máximo de leituras desejadas.

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
```

```
        # Lendo a informação do sensor
        # ...
        counter.increment(1)
```

No exemplo de código a seguir, definimos uma função que inicia uma thread de trabalho para cada sensor e espera que eles terminem suas leituras:

```
def run_threads(func, how_many, counter):
    threads = []
    for i in range(5):
        args = (i, how_many, counter)
        thread = Thread(target=func, args=args)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
```

Rodar cinco threads em paralelo parece ser simples e o resultado esperado deve ser óbvio.

```
how_many = 10**5
counter = Counter()
run_threads(worker, how_many, counter)
print('Counter should be %d, found %d' %
    (5 * how_many, counter.count))

>>>
Counter should be 500000, found 278328
```

No entanto, o resultado está absurdamente fora do esperado! O que aconteceu aqui? Como uma coisa tão simples pode ter errado tanto, especialmente porque apenas uma thread do interpretador Python pode rodar em um dado momento?

O interpretador Python forçosamente impõe o mais perfeito equilíbrio entre todas as threads sendo executadas para garantir que consumam exatamente o mesmo tempo de CPU. Para que isso possa ser garantido, o Python suspende abruptamente uma thread que esteja rodando e passa o controle da CPU para a próxima da fila. O problema é que não se sabe exatamente quando o Python suspenderá a thread, que pode inclusive ser pausada bem no meio de uma

operação atômica. É o que aconteceu neste caso.

O método increment do objeto Counter parece bem simples.

```
counter.count += offset
```

Contudo, o operador += usado em um atributo de objeto instrui o Python para, na verdade, fazer três coisas separadas por debaixo dos panos. O comando anterior é equivalente ao seguinte código:

```
value = getattr(counter, 'count')
result = value + offset
setattr(counter, 'count', result)
```

As threads do Python que estão incrementando o contador podem ser suspensas após o término de qualquer uma dessas três operações. Isso é problemático quando a maneira com que as operações podem ser escalonadas fazem com que versões antigas de value sejam atribuídas ao contador. O código a seguir mostra um exemplo de má interação entre duas threads, A e B:

```
# Rodando na thread A
value_a = getattr(counter, 'count')
# Contexto é transferido para a thread B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Contexto é transferido de volta para a thread A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

A thread A “atropelou” a thread B, apagando todos os seus passos de incremento do contador. Foi exatamente isso o que aconteceu com o exemplo do sensor ali atrás.

Para evitar esse tipo de condição de corrida em seus dados (também conhecidos como data racing conditions ou simplesmente data races) e outras formas de corrupção em estruturas de dados, o Python oferece um conjunto bastante robusto de ferramentas no módulo nativo threading. A mais simples e útil delas é a classe Lock, que implementa uma trava de exclusão mútua (mutual-exclusion lock, ou mutex).

Usando uma mutex (ou seja, uma trava), podemos fazer com que a classe Counter proteja seu valor atual contra acesso simultâneo de múltiplas threads. Apenas uma thread terá a posse da trava em um dado momento. No exemplo de código a seguir, usamos um comando with para obter a mutex e posteriormente liberá-la. Dessa forma, fica fácil determinar qual código está sendo executado enquanto a trava existir (consulte o Item 43: “Considere os comandos contextlib e with para um comportamento reutilizável de try/finally” para mais informações):

```
class LockingCounter(object):
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Agora, ao executar as threads de trabalho como antes, mas usando LockingCounter, temos:

```
counter = LockingCounter()
run_threads(worker, how_many, counter)
print('Counter should be %d, found %d' %
      (5 * how_many, counter.count))
>>>
Counter should be 500000, found 500000
```

O resultado é exatamente o esperado. A classe Lock resolveu a parada.

Lembre-se

- Mesmo que o interpretador do Python tenha uma trava global, ainda é responsabilidade do programador proteger seu código contra condições de corridas nos dados provocadas pelas threads do programa.
- Os programas corromperão suas próprias estruturas de dados se for permitido que mais de uma thread modifique o mesmo objeto sem que haja travas de proteção.

- A classe Lock do módulo nativo threading é a implementação-padrão de travas de exclusão mútua (mutex) no Python.

Item 39: Use Queue para coordenar o trabalho entre as threads

Os programas em Python que fazem muitas coisas de forma simultânea precisam coordenar o trabalho. Um dos arranjos mais úteis para implementar concorrência é com uma pipeline de funções.

Uma pipeline funciona como uma linha de montagem em uma fábrica. As pipelines têm muitas fases em sequência, cada uma com uma função específica. Novas tarefas são constantemente adicionadas ao começo da fila. Cada função pode operar na tarefa que lhe é confiada de forma concorrente às demais threads. A fila anda à medida que cada função é finalizada até que não haja nenhuma fase a processar. Essa técnica é especialmente eficiente para trabalho que inclua bloqueio de I/O ou subprocessos – atividades que podem ser postas em paralelo facilmente usando Python (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”).

Por exemplo, digamos que se queira construir um sistema que receba na entrada uma sequência de imagens de sua câmera digital, redimensione cada uma delas e as adicione em ordem a um álbum de fotos que esteja na internet. Esse programa pode ser dividido em três fases dentro de uma pipeline. Novas imagens são transferidas da câmera na primeira fase. As imagens baixadas são repassadas a uma função de redimensionamento na segunda fase. As imagens redimensionadas são consumidas pela função de upload na fase final.

Imagine que já tenhamos escrito funções em Python que executem as três fases: download, resize, upload. Como montamos uma pipeline para que o trabalho possa ser feito de forma concorrente?

A primeira coisa a implementar é uma maneira de transferir o trabalho entre uma fase e outra. Isso pode ser modelado como uma fila produtor-consumidor que seja segura para as threads (consulte o Item 38: “Use Lock para evitar que as threads iniciem condições de corrida nos dados” para entender a importância da segurança de threads no Python. Adicionalmente, consulte o Item 46: “Use algoritmos e estruturas de dados nativos” para saber mais sobre a classe deque).

```
class MyQueue(object):
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

O produto, sua câmera digital, adiciona novas imagens ao fim da lista de itens pendentes.

```
    def put(self, item):
        with self.lock:
            self.items.append(item)
```

O consumidor, a primeira fase de nossa pipeline, remove a imagem que está na primeira posição de nossa lista de itens pendentes.

```
    def get(self):
        with self.lock:
            return self.items.popleft()
```

No exemplo de código a seguir, representamos cada fase da pipeline como uma thread em Python que aceita trabalho como a partir de uma fila como esta, roda uma função nele e coloca o resultado em outra fila. Também podemos saber a qualquer momento quantas vezes a thread de trabalho verificou a fila de entrada para consultar se havia itens lá, bem como a quantidade de trabalho que já foi completada.

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

A parte mais crítica é que a thread de trabalho precisa saber o que fazer quando a fila de entrada estiver vazia porque a fase anterior ainda não tiver completado seu trabalho. Isso acontece quando a exceção IndexError é capturada, como demonstrado no exemplo a seguir. Podemos associar essa situação a um gargalo na linha de montagem.

```
    def run(self):
        while True:
            self.polled_count += 1
            try:
                item = self.in_queue.get()
            except IndexError:
                sleep(0.01)  # Nenhum trabalho a fazer
            else:
                result = self.func(item)
                self.out_queue.put(result)
                self.work_done += 1
```

Agora podemos conectar as três fases juntas, criando as filas para seus pontos de coordenação e as threads de trabalho correspondentes.

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

Podemos iniciar as threads e, depois, injetar certa quantidade de trabalho na primeira fase da pipeline. No exemplo de código a seguir, usamos uma instância simples de object como um substituto improvisado para os dados verdadeiros que a função download está esperando.

```
for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
```

Agora, esperamos que todos os itens sejam processados pela pipeline até que tudo acabe indo parar na fila done_queue.

```
while len(done_queue.items) < 1000:
    # Alguma coisa útil é feita aqui, enquanto esperamos...
    # ...
```

Tudo roda como esperado, mas há um efeito colateral interessante, causado pelas constantes verificações das threads às filas de entrada de cada uma. A parte crítica, quando capturamos as exceções IndexError no método run, é executada um número excessivo de vezes.

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print('Processed', processed, 'items after polling',
    polled, 'times')
>>>
Processed 1000 items after polling 3030 times
```

Quando as velocidades das funções de trabalho variam entre si, uma fase anterior pode impedir que a pipeline avance em fases posteriores, gerando um gargalo. Com isso, os processos mais adiante na pipeline vão, às vezes, ficar sem nenhum trabalho para fazer e constantemente verificarão suas filas de entrada em busca de mais trabalho a fazer. Como resultado, as threads de trabalho desperdiçam tempo de CPU com tarefas inúteis (constantemente elevando e subsequentemente capturando exceções IndexError).

No entanto, esse é só o começo do que está errado com essa implementação. Há três outros problemas que precisamos evitar. Primeiro, a ação de determinar que todo o trabalho na entrada já foi processado requer um ciclo de espera adicional (busy wait) na fila done_queue. Segundo, na classe Worker o método run (que ocupa muitos recursos do sistema) está sendo executado num laço infinito. Não há maneira de sinalizar a uma thread de trabalho que é hora de encerrar o expediente.

Terceiro, e pior de todos, um atraso na pipeline pode fazer o programa travar arbitrariamente. Se a primeira avançar rapidamente sobre sua fila de entrada, mas a segunda fase for bem mais lenta, a fila conectando a primeira e a segunda fase crescerá de tamanho indefinidamente. A segunda fase não será capaz de aguentar por muito tempo. Dê ao programa tempo suficiente e dados de entrada em fluxo constante, e chegará um momento em que a memória disponível para o

programa estará repleta. Nesse momento, o programa travará de forma espetacular.

A lição a aprender aqui é que as pipelines não são ruins em si, mas é muito difícil construir você mesmo uma fila produtor-consumidor decente.

Chamem a cavalaria, digo, a classe Queue

A classe Queue, do módulo nativo queue, oferece toda a funcionalidade de que precisamos para resolver todos esses problemas.

Queue elimina a espera nas filas (busy waiting) das threads de trabalho ao fazer com que o método get fique bloqueado enquanto não existam dados a processar na fila de entrada. Por exemplo, o código inicia uma thread que espera dados em uma fila de entrada:

```
from queue import Queue
queue = Queue()

def consumer():
    print('Consumer waiting')
    queue.get()             # Só é executada depois no put() mais
                            # adiante no código
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

Mesmo que a thread tenha sido iniciada primeiro, não será finalizada até que um item seja colocado (put) na instância de Queue, ou seja, a thread só prosseguirá quando o método get tenha algo para devolver.

```
print('Producer putting')
queue.put(object())         # Roda antes do get() que está lá no
                            # início do código
thread.join()
print('Producer done')
>>>
```

```
Consumer waiting
Producer putting
Consumer done
Producer done
```

Para resolver o problema de gargalo na pipeline, a classe Queue permite que se especifique o tamanho máximo de trabalho pendente entre duas fases. Esse tamanho de buffer faz com que as chamadas a put bloqueiem a fila quando ela já estiver cheia. Por exemplo, o código a seguir define uma thread que espera um momentinho antes de consumir o próximo item da fila.

```
queue = Queue(1)              # Tamanho do buffer = 1

def consumer():
    time.sleep(0.1)           # Espera
    queue.get()               # Segunda a ser executada
    print('Consumer got 1')
    queue.get()               # Quarta a ser executada
    print('Consumer got 2')

thread = Thread(target=consumer)
thread.start()
```

A espera deve permitir que a thread produtora ponha um segundo objeto na fila (com put) antes que a thread consume tenha oportunidade de chamar a função get. O tamanho de Queue, aqui, é 1, o que significa que o produtor sempre terá que esperar a thread consumidora chamar a função get ao menos uma vez para que a segunda chamada a put pare de bloquear seu próprio processamento e adicione um segundo item à fila.

```
queue.put(object())           # Roda primeiro
print('Producer put 1')
queue.put(object())           # Terceira a ser executada
print('Producer put 2')
thread.join()
print('Producer done')
>>>
```

```
Producer put 1
Consumer got 1
Producer put 2
Consumer got 2
Producer done
```

A classe Queue pode também acompanhar o progresso do trabalho empregando o método task_done. Isso permite que o processamento espere pelo esvaziamento da fila de entrada e elimina a necessidade de consultar periodicamente a lista done_queue no fim da pipeline. Por exemplo, o código a seguir define uma thread consumidora que chama task_done quando termina de processar um item.

```
in_queue = Queue()

def consumer():
    print('Consumer waiting')
    work = in_queue.get()      # Segunda a ser executada
    print('Consumer working')
    # Trabalhando
    # ...
    print('Consumer done')
    in_queue.task_done()       # Terceira a ser executada

Thread(target=consumer).start()
```

Agora, o código produtor não precisa se conectar diretamente à thread consumidora nas consultas, bastando que o produtor espere que in_queue termine e chame join na instância de Queue. Mesmo depois de esvaziada, a fila in_queue não estará liberada até que task_done tenha sido chamada para todo e qualquer item que tenha sido posto na fila.

```
in_queue.put(object())        # Executada primeiro
print('Producer waiting')
in_queue.join()               # Quarta a ser executada
print('Producer done')
>>>
```

```
Consumer waiting
Producer waiting
Consumer working
Consumer done
Producer done
```

Podemos colocar todos esses comportamentos juntos em uma subclasse de Queue que também diz à thread de trabalho o momento em que deve parar de processar. No exemplo de código a seguir, eu defino um método close que adiciona um item especial à fila de entrada, indicando que não haverá mais itens a processar depois dele:

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Depois, definimos um iterador para a fila que procura por esse objeto especial e interrompe a iteração quando o encontra. Esse método __iter__ também chama task_done nos momentos oportunos, permitindo que o programador possa acompanhar o avanço no processamento da fila.

```
    def __iter__(self):
        while True:
            item = self.get()
            try:
                if item is self.SENTINEL:
                    return  # Obriga a thread a encerrar-se
                yield item
            finally:
                self.task_done()
```

Em seguida, podemos redefinir a thread de trabalho para se apoiar no comportamento da classe ClosableQueue. A thread será encerrada após a exaustão do laço.

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
```

```
        # ...

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

No exemplo de código a seguir, recriamos o conjunto de threads de trabalho usando a nova classe de trabalho:

```
download_queue = ClosableQueue()
# ...
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    # ...
]
```

Depois de executar as threads de trabalho como antes, também enviamos o sinal de encerramento logo após todos os dados de entrada terem sido injetados, e fechamos a fila de entrada da primeira fase.

```
for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
download_queue.close()
```

Finalmente, esperamos pelo término do trabalho unindo as filas que conectam as fases. Cada vez que uma fase termina, é enviado um sinalizador para a próxima fase fechando sua fila de entrada. No final, a fila done_queue contém todos os objetos de saída, como esperado.

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')
```

```
>>>
1000 items finished
```

Lembre-se

- Pipelines são uma excelente maneira de organizar sequências de trabalho que rodam de forma simultânea (em concorrência) usando múltiplas threads do Python.
- Esteja ciente dos muitos problemas envolvendo pipelines concorrentes: busy waiting, threads de trabalho erráticas e explosão no uso de memória.
- A classe Queue tem todos os recursos necessários para construir pipelines robustas: operações de bloqueio, tamanho de buffer e união de filas.

Item 40: Considere usar corrotinas para rodar muitas funções simultaneamente

As threads dão aos programadores do Python uma maneira de executar múltiplas funções aparentemente ao mesmo tempo (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”). Mas há três grandes problemas com as threads:

- Requerem ferramentas especiais para coordenar entre elas o trabalho de forma que uma não interfira com a outra (consulte o Item 38: “Use Lock para evitar que as threads iniciem condições de corrida nos dados” e o Item 39: “Use Queue para coordenar o trabalho entre as threads”). Isso torna o código que usa threads muito mais difícil de entender que um código procedural, de apenas uma thread. Essa complexidade torna o código com threads também difícil de estender e manter ao longo do tempo.
- Requerem muita memória, em torno de 8 MB por thread. Na maioria dos computadores, essa quantidade de memória não é relevante se o programa dispara uma ou duas dezenas de threads. Contudo, e se um determinado programa disparar não dezenas, mas centenas de funções “simultaneamente”? Essas funções podem corresponder a solicitações dos usuários a um servidor, pixels na tela, partículas em uma simulação etc. Executar uma thread exclusiva para cada atividade única simplesmente não funciona.

- Threads são ávidas consumidoras de recursos da máquina quando estão sendo inicializadas. Se o programa, em vez de trabalhar com as mesmas threads do começo ao fim, fica o tempo todo criando e encerrando funções concorrentes, acaba gerando um processamento adicional espúrio (overhead) que não é desprezível. Esse overhead, à medida que o número de threads aumenta, também cresce e torna o conjunto todo mais lento.

O Python consegue contornar todos esses problemas com o uso das *corrotinas*, ou *coroutines*. As corrotinas permitem iniciar funções aparentemente simultâneas nos programas em Python. Elas foram implementadas como uma extensão dos geradores (consulte o Item 16: “Prefira geradores em vez de retornar listas”). O custo de iniciar uma corrotina geradora é o de uma chamada a função. Uma vez ativa, cada corrotina usa menos de 1 KB de memória durante toda a sua vida útil.

As corrotinas permitem que o código que consome um gerador envie (coma função send) um valor de retorno para essa mesma função geradora depois de cada expressão yield. A função geradora recebe o valor passado pela função send como resultado da expressão yield correspondente.

```
def my_coroutine():
    while True:
        received = yield
        print('Received:', received)

it = my_coroutine()
next(it)            # Inicializa a corrotina
it.send('First')
it.send('Second')
>>>
Received: First
Received: Second
```

A chamada inicial a next é necessária para preparar o gerador para receber o primeiro send. Essa preparação se dá avançando-o para a primeira expressão yield. Juntos, yield e send dão aos geradores uma maneira padrão de variar, em resposta a um estímulo externo, o próximo valor gerado por yield.

Por exemplo, digamos que se queira implementar uma corrotina geradora que indique o valor mínimo que enviou até o momento. No exemplo de código a seguir, o yield sozinho prepara a corrotina com um valor mínimo inicial enviado do mundo externo. O gerador, então, repetidamente informa o novo mínimo em troca do próximo valor a considerar:

```
def minimize():
    current = yield
    while True:
        value = yield current
        current = min(value, current)
```

O código que consome o gerador pode ser executado um passo de cada vez e enviará para a saída o valor mínimo visto logo depois que cada valor seja apresentado na entrada.

```
it = minimize()
next(it)          # Inicializa o gerador
print(it.send(10))
print(it.send(4))
print(it.send(22))
print(it.send(-1))
>>>
10
4
4
-1
```

A função geradora poderia aparentemente rodar para sempre, progredindo a cada nova chamada a send. Da mesma forma que nas threads, as corrotinas são funções independentes que consomem valores de entrada entregues por qualquer elemento de seu ambiente e produzem resultados na saída. A diferença é que as corrotinas entram em pausa a cada expressão yield na função geradora e retomam o processamento após cada chamada a send vinda do mundo exterior. Este é o mecanismo mágico das corrotinas.

Esse comportamento permite que o código que esteja consumindo um gerador possa iniciar alguma ação logo depois de cada expressão yield na corrotina. O

código consumidor pode usar os valores de saída do gerador para chamar outras funções e atualizar estruturas de dados. Mais importante, pode avançar as funções de outras funções geradoras até a próxima expressão yield. Extrapolando, dado um grande número de geradores, se cada um disparar o avanço de outro separadamente, nessa sequência de “avança um passo e depois para” (ou, em inglês, lockstep), eles todos parecerão estar rodando simultaneamente, imitando o comportamento concorrente das threads do Python.

Jogo da Vida

O comportamento simultâneo das corrotinas pode ser demonstrado com um exemplo bastante conhecido. Digamos que se queira usar corrotinas para implementar o Jogo da Vida, de John Conway[1]. As regras do jogo são bastante simples. Temos uma grade bidimensional de tamanho arbitrário. Cada célula na grade pode tanto estar viva ou vazia.

```
ALIVE = '*'
EMPTY = '-'
```

O jogo progride um passo por vez a cada pulso (“tic-tac”) do relógio. Cada vez que o relógio faz tic (ou tac) cada uma das células conta quantas células vizinhas (num total de 8) estão vivas. Baseado na contagem de vizinhos, cada célula decide se continuará viva, morrerá ou voltará dos mortos. O código a seguir mostra um exemplo de um Jogo da Vida em uma grade de 5×5, com o tempo passando da esquerda para a direita. Veremos as regras específicas mais tarde.

```
  0   |   1   |   2   |   3   |   4
----- | ----- | ----- | ----- | -----
-*--- | --*-- | --**- | --*-- | -----
--**- | --**- | -*--- | -*--- | -**--
---*- | --**- | --**- | --*-- | -----
----- | ----- | ----- | ----- | -----
```

Podemos modelar esse jogo representando cada célula como uma corrotina geradora rodando em lockstep (“avança um passo e depois para”) em relação a todas as outras.

Para essa implementação, primeiro precisamos de uma maneira de obter o estado das células vizinhas. Podemos fazer isso com uma corrotina chamada

count_neighbors, que funciona produzindo (com yield) objetos Query. A classe Query é definida em nosso próprio código. Seu propósito é fazer com que a corrotina geradora possa perguntar às células vizinhas sobre seu estado.

```
Query = namedtuple('Query', ('y', 'x'))
```

A corrotina produz um Query para cada vizinho. O resultado de cada expressão yield poderá ter valores ALIVE ou EMPTY. Esse é o contrato de interface que definimos entre a corrotina e o código consumidor. O gerador count_neighbors vê os estados dos vizinhos e retorna a contagem dos vizinhos vivos.

```
def count_neighbors(y, x):
    n_ = yield Query(y + 1, x + 0)  # Norte
    ne = yield Query(y + 1, x + 1)  # Nordeste
    # Aqui vai o código que define e_, se, s_, sw, w_, nw ...
    # ...
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Podemos alimentar a corrotina count_neighbors com dados falsos para testá-la. No exemplo de código a seguir, percebemos como os objetos Query são produzidos para cada vizinho. count_neighbors recebe os estados da célula correspondente a cada Query por meio do método send da corrotina. A contagem final é devolvida na exceção StopIteration que é levantada quando o gerador é exaurido pelo comando return.

```
it = count_neighbors(10, 5)
q1 = next(it)                # Obtém a primeira Query
print('First yield: ', q1)
q2 = it.send(ALIVE)          # Manda o estado de q1, obtém q2
print('Second yield:', q2)
q3 = it.send(ALIVE)          # Manda o estado de q2, obtém q3
# ...
try:
```

```
    it.send(EMPTY)             # Manda o estado de q8, obtém a contagem
  except StopIteration as e:
    print('Count: ', e.value)  # Valor obtido pelo comando return
  >>>
  First yield:  Query(y=11, x=5)
  Second yield: Query(y=11, x=6)
  ...
  Count:  2
```

O próximo passo é implementar alguma indicação de que uma célula mudará de estado em resposta à contagem de vizinhos obtida de count_neighbors. Para isso, definimos outra corrotina chamada step_cell. Esse gerador indicará a transição do estado de uma célula produzindo objetos Transition, outra classe definida por nós mesmos, como fizemos com Query.

```
  Transition = namedtuple('Transition', ('y', 'x', 'state'))
```

A corrotina step_cell recebe suas coordenadas na grade como argumentos. Ela produz uma Query para obter o estado inicial dessas coordenadas. Depois, roda count_neighbors para inspecionar as células ao seu redor e executa a lógica do jogo para determinar que estado a célula deve adotar no próximo pulso de relógio. Por fim, produz um objeto Transition para anunciar ao ambiente o próximo estado da célula.

```
  def game_logic(state, neighbors):
    # ...

  def step_cell(y, x):
    state = yield Query(y, x)
    neighbors = yield from count_neighbors(y, x)
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)
```

Observe: a chamada a count_neighbors usa a expressão yield from. Essa expressão permite que o Python monte uma composição de várias corrotinas geradoras, facilitando a reutilização de trechos pequenos de funcionalidade para construir corrotinas complexas a partir das mais simples. Quando count_neighbors for exaurido, o valor final que ele devolve (no comando return)

será passado a step_cell como resultado da expressão yield from.

Depois dessa preparação toda, podemos finalmente definir a lógica simples do Jogo da Vida de Conway. Há apenas três regras.

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY     # Morre: poucos vizinhos
        elif neighbors > 3:
            return EMPTY     # Morre: vizinhos demais
    else:
        if neighbors == 3:
            return ALIVE     # Regenera (volta das tumbas)
    return state
```

Podemos alimentar a corrotina step_cell com dados falsos para testá-la.

```
it = step_cell(10, 5)
q0 = next(it)          # Obtém localização inicial
print('Me:      ', q0)
q1 = it.send(ALIVE)    # Envia o estado, obtém query do primeiro vizinho
print('Q1:      ', q1)
# ...
t1 = it.send(EMPTY)     # Envia para q8, obtém a decisão do jogo
print('Outcome: ', t1)
>>>
Me:       Query(y=10, x=5)
Q1:       Query(y=11, x=5)
...
Outcome:  Transition(y=10, x=5, state='-')
```

O objetivo do jogo é executar esta lógica em uma grade completa de células e em lockstep. Para isso, podemos montar uma composição da corrotina step_cell para criar a corrotina simulate. Esta avança sobre a grade de células e produz resultados de step_cell ciclicamente, inúmeras vezes. Depois de varrer todas as coordenadas da grade, produz um objeto TICK, indicando que a geração atual de

células já passou completamente pela transição e está na geração seguinte.

```
TICK = object()


def simulate(height, width):
    while True:
        for y in range(height):
            for x in range(width):
                yield from step_cell(y, x)
        yield TICK
```

O mais impressionante a respeito de simulate é que a corrotina é completamente desconectada do ambiente a seu redor. Ainda não definimos como a grade será representada em termos de objetos do Python, como os valores de Query, Transition e TICK serão tratados no mundo externo e como o jogo obtém seu estado inicial. Porém, a lógica é clara. Cada célula passa pela transição ao executar step_cell. Quando todas as células tiverem determinado qual será seu estado na próxima geração, o relógio do jogo avançará um passo, ou seja, um tic (ou tac). O processo continua ininterruptamente enquanto a corrotina simulate estiver sendo executada passo a passo.

Essa é a beleza das corrotinas! Elas ajudam o programador a se concentrar apenas na lógica do que está tentando implementar, desacoplando e tornando independentes as duas partes de seu código: as instruções para o ambiente e a implementação que realiza o objetivo final do programa. Isso permite que as corrotinas pareçam estar sendo executadas em paralelo, e também aprimorar a implementação que rastreia essas instruções ao longo do tempo sem que as corrotinas precisem ser modificadas.

Passados os testes, obviamente queremos rodar simulate em um ambiente real. Para isso, precisamos representar o estado de cada célula na grade. No exemplo de código a seguir, definimos uma classe que contém a grade:

```
class Grid(object):
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
```

```
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def __str__(self):
        # ...
```

A grade permite que se obtenha (get) e defina (set) o valor de cada coordenada. As coordenadas que estiverem além dos limites da grade serão recalculadas para aparecer na margem oposta, tornando a grade um espaço infinito esférico.

```
    def query(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def assign(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state
```

Por fim, e até que enfim, podemos definir a função que interpreta os valores produzidos por simulate e todas as suas corrotinas inferiores. Essa função toma as instruções das corrotinas e as transforma em interações com o ambiente em volta da célula. Toda a grade é varrida, uma célula por vez, e retorna uma nova grade contendo os estados da próxima geração de células.

```
def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Query):
            state = grid.query(item.y, item.x)
            item = sim.send(state)
        else:  # Deve necessariamente passar pela transição
            progeny.assign(item.y, item.x, item.state)
            item = next(sim)
    return progeny
```

Para ver essa função fazer seu trabalho, precisamos criar uma grade e determinar seu estado inicial. No exemplo de código a seguir, implementamos uma das formas clássicas que possuem comportamento matemático previsto: o planador ou, como é mais conhecido, glider:

```
grid = Grid(5, 9)
grid.assign(0, 3, ALIVE)
# ...
print(grid)
>>>
---*-----
----*----
--***----
---------
---------
```

Agora, posso avançar a grade no tempo, uma geração por vez. Como podemos notar, o glider move-se em diagonal, para baixo e para a direita. Esse comportamento é ditado exclusivamente pelo seu formato e pelas regras simples da função game_logic.

```
class ColumnPrinter(object):
    # ...

columns = ColumnPrinter()
sim = simulate(grid.height, grid.width)
for i in range(5):
    columns.append(str(grid))
    grid = live_a_generation(grid, sim)

print(columns)
>>>
   0    |    1    |    2    |    3    |    4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

A melhor característica desse tipo de implementação é que podemos mudar a

função game_logic sem ter que atualizar o código que o cerca. Eu posso modificar as regras ou adicionar esferas de influência maiores usando a mecânica existente de Query, Transition e TICK, o que demonstra como as corrotinas segregam muito bem os problemas, um importante princípio no projeto e planejamento de um sistema.

Corrotinas no Python 2

Infelizmente, o Python 2 não tem muito do açúcar sintático que torna as corrotinas tão elegantes no Python 3. Há duas limitações. Primeiro, não existe uma expressão yield from. Isso significa que quando queremos montar composições de corrotinas geradores no Python 2, é necessário incluir um laço no ponto de delegação.

```
# Python 2
def delegated():
    yield 1
    yield 2

def composed():
    yield 'A'
    for value in delegated():  # No Python 3 isso seria um "yield from"
        yield value
    yield 'B'

print list(composed())
>>>
['A', 1, 2, 'B']
```

A segunda limitação é que não há suporte para o comando return nos geradores do Python. Para obter o mesmo comportamento que interaja corretamente com blocos try/except/finally, é necessário definir seu próprio tipo de exceção e elevá-lo quando quiser retornar um valor.

```
# Python 2
class MyReturn(Exception):
    def __init__(self, value):
```

```
        self.value = value

def delegated():
    yield 1
    raise MyReturn(2)  # No Python 3 isso seria simplesmente "return 2"
    yield 'Not reached'

def composed():
    try:
        for value in delegated():
            yield value
    except MyReturn as e:
        output = e.value
    yield output * 4

print list(composed())
>>>
[1, 8]
```

Lembre-se

- As corrotinas são um meio muito eficiente de rodar dezenas de milhares de funções aparentemente ao mesmo tempo.
- Dentro de um gerador, o valor de uma expressão yield será qualquer valor passado ao método send do código exterior.
- As corrotinas oferecem uma ferramenta poderosa para separar a lógica central do código (o que você quer fazer) de sua interação com o ambiente ao seu redor.
- O Python 2 não suporta yield from nem permite que os geradores retornem valores.

Item 41: Considere usar concurrent.futures para obter paralelismo real

Em algum momento de nossa vida de programador Python atingimos o chamado muro do desempenho, (performance wall, como é conhecido em inglês). Mesmo depois de otimizar seu código (consulte o Item 58: “Meça os perfis de desempenho antes de otimizar o código”), a execução dos seus programas pode continuar muito lenta. Em computadores modernos, com cada vez mais núcleos de CPU à medida em que novos modelos vão surgindo no mercado, é razoável supor que uma das soluções possíveis seja o paralelismo. Como seria bom se pudéssemos dividir o esforço de computação de nosso código em pequenos pedaços e entregá-los a núcleos distintos para serem executados em paralelo, não é mesmo?

Infelizmente, a Global Interpreter Lock (GIL) do Python impede o paralelismo verdadeiro em threads (consulte o Item 37: “Use threads para bloquear I/O e evitar paralelismo”), portanto essa opção está fora de cogitação. Outra sugestão bastante comum é reescrever em linguagem C o trecho de código que mais sofre com problemas de desempenho. O C é uma linguagem que chega muito perto do baixo nível, ou seja, “fala” diretamente com o hardware. Por isso, é também muito rápida, eliminando a necessidade de paralelismo em muitos casos. As extensões em C podem, além disso, iniciar suas próprias threads nativas que rodam em paralelo e utilizam múltiplos núcleos de CPU. A API do Python para escrever extensões em C é muito bem documentada e uma ótima escolha, caso precise de uma saída de emergência.

Contudo, reescrever seu código em tem um custo altíssimo. Trechos de código que seriam curtos e absolutamente inteligíveis em Python podem tornar-se palavrosos e complicados em C. Uma portagem dessa envergadura necessitaria de muitos e profundos testes para garantir que a funcionalidade seja equivalente à do código original em Python e que nenhum bug foi introduzido no processo. Algumas vezes, o esforço vale a pena, o que explica o ecossistema bastante diverso e populoso de módulos de extensão em C disponíveis para a comunidade de programadores do Python. Esses módulos melhoram o desempenho de um sem-número de tarefas como análise de texto (parsing), composição de imagens e cálculo matricial. Existem até mesmo ferramentas de código aberto como o Cython (*http://cython.org/*) e o Numba (*http://numba.pydata.org/*) que facilitam sobremaneira a transição para o C.

O problema é que portar um trecho do seu programa para C não é suficiente na maioria das vezes. Programas otimizados em Python não possuem um único

fator determinante de lentidão. Em vez disso, há muitos elementos que contribuem para tal. Para que os benefícios de usar threads e estar mais próximo do hardware com o C possam ser empregados, seria necessário portar trechos muito grandes do programa, aumentando drasticamente o risco e as necessidades de teste. Deve haver uma maneira melhor de preservar seu investimento em Python para resolver problemas computacionais difíceis.

O módulo nativo multiprocessing, facilmente acessível por dentro de outro módulo nativo, concurrent.futures, é exatamente o que precisamos. Esse módulo permite que o Python utilize mais de um núcleo de CPU em paralelo, e o faz de maneira surpreendentemente simples: rodando em paralelo instâncias adicionais do interpretador do Python como processos-filho. Tais processos são separados do interpretador principal, portanto cada um dos processos-filho tem uma GIL separada para si. Cada processo-filho tem permissão de utilizar apenas um núcleo de CPU, e possui uma conexão com o processo principal, por onde recebe instruções para computar e devolve os resultados.

Por exemplo, digamos que se queira realizar uma tarefa computacionalmente intensiva com o Python e usar múltiplos núcleos de CPU. Usarei o problema de encontrar o maior divisor comum entre dois números como substituto para um algoritmo mais intenso computacionalmente, como a simulação da dinâmica dos fluídos com a equação de Navier-Stokes.

```
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Rodar essa função em série aumenta linearmente a quantidade de tempo de processamento porque não há paralelismo.

```
numbers = [(1963309, 2265973), (2030677, 3814172),
          (1551645, 2229620), (2039045, 2020802)]
start = time()
results = list(map(gcd, numbers))
end = time()
```

```
print('Took %.3f seconds' % (end - start))
>>>
Took 1.170 seconds
```

Rodar esse código em múltiplas threads no Python não resultaria em nenhum ganho de velocidade porque a GIL impede que o Python use mais de uma CPU em paralelo. No exemplo de código a seguir, implementamos a mesma computação proposta pelo código anterior, mas desta vez usando o módulo concurrent.futures com sua classe ThreadPoolExecutor e duas threads de trabalho (no meu caso, usei duas porque é o número de núcleos que eu tenho disponível na CPU de meu computador):

```
start = time()
pool = ThreadPoolExecutor(max_workers=2)
results = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))
>>>
Took 1.199 seconds
```

Estranhamente, ficou mais lento ainda. Isso se deve ao trabalho adicional de iniciar todas as threads e da comunicação entre o interpretador principal e o conjunto de threads.

Porém, a surpresa maior é que, alterando apenas uma única linha de código, algo mágico acontece. Se eu substituir ThreadPoolExecutor por ProcessPoolExecutor, também do módulo concurrent.futures, tudo fica muito mais rápido.

```
start = time()
pool = ProcessPoolExecutor(max_workers=2)  # Única linha modificada
results = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))
>>>
Took 0.663 seconds
```

Em minha máquina dual-core, é significativamente mais rápido! Como isso é

possível? O código a seguir mostra o que a classe ProcessPoolExecutor realmente faz (fazendo uso das estruturas de linguagem de baixo nível disponibilizados pelo módulo multiprocessing):

1. Toma cada item de numbers, na entrada, e o entrega a map.
2. Os dados são serializados em binário usando o módulo pickle (consulte o Item 44: “Aumente a confiabilidade de pickle com copyreg”).
3. Os dados serializados são copiados do processo principal do interpretador para um processo-filho por um socket local.
4. Depois, os dados são desserializados para seu formato original, objetos do Python, usando pickle no processo-filho.
5. O módulo de Python contendo a função gcd é importado.
6. A função é executada nos dados de entrada em paralelo com outros processos-filho.
7. O resultado é novamente serializado em bytes.
8. Os bytes são copiados através do socket para o processo principal do interpretador.
9. Os bytes são desserializados e convertidos em objetos do Python no processo-pai.
10. Por fim, os resultados devolvidos pelos diversos processos-filho são unidos em uma única lista para ser devolvida com return.

Embora pareça simples ao programador, o módulo multiprocessing e a classe ProcessPoolExecutor executam uma quantidade monstruosa de trabalho para que o paralelismo possa ser possível. Em muitas linguagens, o único ponto de contato necessário para coordenar duas threads é uma única trava ou operação atômica. O tempo de CPU adicional necessário para usar multiprocessing é alto por conta de todas as serializações e desserializações que precisam ocorrer para que o processo-pai possa conversar com os processos-filho.

Esse esquema é bastante apropriado para certos tipos de tarefas isoladas e de alta alavancagem. Por isoladas, entendemos funções que não precisam compartilhar seus estados com outras partes do programa. Por alta alavancagem entendemos situações nas quais apenas uma pequena porção dos dados precisa ser transferida entre os processos pai e filho, mas mesmo assim a computação efetuada é bastante intensa. O algoritmo de máximo denominador comum é um exemplo

disso, mas muitos outros algoritmos matemáticos trabalham de forma semelhante.

Caso a computação que se queira fazer não possua essas características, o peso adicional na CPU provocado por multiprocessing pode impedir que o programa seja acelerado pelo paralelismo. Quando nos deparamos com situações assim, o mesmo multiprocessing oferece recursos mais avançados de memória compartilhada, travas interprocessos, filas e substitutos (proxies). Todos esses recursos, todavia, são por demais complexos. Já é bastante difícil entender o funcionamento dessas ferramentas no espaço de memória de um único processo compartilhado por outras threads do Python. Aumentar ainda mais essa complexidade para outros processos e incluir sockets na receita torna tudo isso difícil demais de entender.

Minha sugestão é: evite usar diretamente os recursos de multiprocessing e, em vez disso, empregue os recursos mais "mastigados" pelo módulo concurrent.futures, bem mais simples. Comece usando a classe ThreadPoolExecutor para executar funções isoladas e de alta alavancagem em threads. Mais tarde, substitua a classe por ProcessPoolExecutor para ter um incremento na velocidade. Por fim, depois de ter tentado todas as opções mais simples, aventure-se em usar diretamente o módulo multiprocessing.

Lembre-se

- Transferir os gargalos de CPU para as extensões em C pode ser uma maneira bastante eficiente de aumentar o desempenho enquanto maximiza seu investimento no código em Python. Entretanto, o custo disso é muito alto e pode introduzir bugs.
- O módulo multiprocessing oferece ferramentas poderosas que podem paralelizar certos tipos de computação em Python com um mínimo de esforço.
- O poder de multiprocessing é mais bem utilizado por meio do módulo nativo concurrent.futures e sua classe ProcessPoolExecutor que, por sua natureza, é bastante simples.
- As partes avançadas do módulo multiprocessing devem ser evitadas porque são complexas demais.

---

1 N. do T.: O Jogo da Vida (The Game of Life) é um experimento de autômato celular, ou seja, cada célula da grade pode tomar decisões por si só, sem a intervenção do usuário. A única coisa que o jogador humano pode fazer é determinar o estado inicial do jogo e vê-lo progredir. Dependendo de como o estado inicial (ou seja, quais células começam vivas), a colônia pode se extinguir após algumas gerações, pode crescer até um desenho estável e estático, ou pode crescer até formar um desenho em movimento cíclico. Foi inventado em 1970 pelo matemático britânico John Horton Conway e é um pioneiro dos softwares de simulação, provando, entre outras coisas, que um universo complexo pode surgir a partir de poucos elementos e regras simples. Mais informações sobre o Jogo da Vida podem ser obtidas na Wikipédia: *https://pt.wikipedia.org/wiki/Jogo_da_vida*. Uma interessante implementação do Jogo da Vida para navegadores web, em um grid de tamanho variável, pode ser experimentada em *http://www.bitstorm.org/gameoflife/*.

CAPÍTULO 6

Módulos nativos

Em se tratando de sua biblioteca-padrão, o Python é como aqueles produtos que já vêm com tudo incluso, inclusive as pilhas. Muitas outras linguagens são distribuídas com um número muito pequeno de pacotes comuns e obrigam o pobre programador a varrer o mundo em busca daquela funcionalidade importante que está faltando. Embora o Python ofereça um repositório fabuloso de módulos criados pela comunidade, a linguagem tenta entregar, já na instalação default, uma grande quantidade de módulos importantes para tornar a linguagem útil e abrangente.

O conjunto completo de módulos-padrão é grande demais para ser abordado num livro como este. Contudo, alguns desses módulos nativos são tão entrelaçados com o Python básico que poderiam muito bem ser considerados como parte da especificação da linguagem. Esses módulos nativos essenciais são especialmente importantes quando estamos escrevendo as partes mais intrincadas e propensas a erros de nossos programas.

Item 42: Defina decoradores de função com functools.wraps

O Python possui uma sintaxe especial para definir *decoradores* (*decorators*, em inglês) que podem ser aplicados a funções. Os decoradores permitem executar código adicional antes e depois das chamadas às funções que eles envolvem, o que por sua vez permite acessar e modificar os argumentos de entrada e os valores de retorno. Essa funcionalidade pode ser bastante útil para forçar semântica, depurar erros, registrar funções e muito mais.

Por exemplo, digamos que se queira mostrar no console os argumentos e valores de retorno da chamada a uma função. Isso é especialmente útil quando estamos depurando uma pilha de chamada a funções em uma função recursiva. No exemplo de código a seguir, definimos um decorador como esse:

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper
```

Para aplicá-lo a uma função, usamos o símbolo @.

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

O símbolo @ é equivalente a chamar o decorador na função que ele envolve e associar o valor de retorno ao nome original no mesmo escopo.

```
fibonacci = trace(fibonacci)
```

Chamar essa função decorada rodará o código envolvente (wrapper) antes e depois da função fibonacci ser executada, mostrando no console os argumentos e o valor de retorno a cada passo da tarefa recursiva.

```
fibonacci(3)
```

```
>>>
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2
```

Funciona bem, mas tem um efeito colateral indesejado. O valor devolvido pelo decorador – a função que é chamada acima – não acredita que seu nome seja fibonacci.

```
print(fibonacci)
>>>
```

```
<function trace.<locals>.wrapper at 0x107f7ed08>
```

O motivo é óbvio. A função trace retorna a função wrapper que ela define dentro de si. É wrapper o nome atribuído a fibonacci no módulo, por causa do decorador. Esse comportamento é problemático porque atrapalha ferramentas que fazem introspecção, como os depuradores (consulte o Item 57: “Prefira usar depuradores interativos como o pdb”) e os serializadores de objetos (consulte o Item 44: “Aumente a confiabilidade de pickle com copyreg”).

Por exemplo, a função nativa help torna-se inútil na função decorada fibonacci.

```
help(fibonacci)
>>>
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

A solução é usar a função auxiliar wraps no módulo nativo functools. Em resumo, wraps é um decorador que ajuda a escrever decoradores. Aplicá-lo à função wrapper copiará todos os metadados importantes da função interna para a função externa.

```
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # ...
    return wrapper

@trace
def fibonacci(n):
    # ...
```

Se executarmos a função help agora, produziremos o resultado esperado, mesmo que a função esteja decorada.

```
help(fibonacci)
>>>
Help on function fibonacci in module __main__:
```

```
fibonacci(n)
    Return the n-th Fibonacci number
```

Chamar o atributo help é apenas um exemplo de como os decoradores podem causar problemas bastante sutis. As funções do Python têm muitos outros atributos-padrão (por exemplo, __name__, __module__) que devem ser preservados para manter funcionando a interface das funções na linguagem. Ao usar wraps, garantimos que o comportamento será sempre correto.

Lembre-se

- Os decoradores são a sintaxe oficial do Python para permitir que uma função modifique outra em tempo de execução.
- Usar decoradores pode provocar comportamento errático em ferramentas que fazem introspecção, como os depuradores (debuggers).
- Use o decorador wraps, disponível no módulo nativo functools, quando definir seus próprios decoradores, evitando assim diversos problemas.

Item 43: Considere os comandos contextlib e with para um comportamento reutilizável de try/finally

No Python, o comando with é usado para indicar quando algum código está rodando em um contexto especial. Por exemplo, travas de exclusão mútua, ou mutex (consulte o Item 38: “Use Lock para evitar que as threads iniciem condições de corrida nos dados”), podem ser usadas com comandos with para indicar que o código indentado só pode ser executado quando a trava estiver em ação.

```
lock = Lock()
with lock:
    print('Lock is held')
```

O exemplo anterior é equivalente a uma construção try/finally porque a classe Lock ativa, de forma apropriada, o comando with.

```
lock.acquire()
try:
    print('Lock is held')
```

```
finally:
    lock.release()
```

A versão do código com o comando with é melhor porque elimina a necessidade de escrever o código repetitivo da construção try/finally. O módulo nativo contextlib facilita tornar seus objetos e funções capazes de serem usados com o comando with. Esse módulo contém o decorador contextmanager, que permite uma única função ser usada com comandos with. Esse módulo é muito mais fácil que definir uma nova classe com métodos especiais __enter__ e __exit__ (a maneira-padrão de fazer as coisas).

Por exemplo, digamos que se queira incluir mais mensagens de depuração em alguma região do código de um programa qualquer. O exemplo de código a seguir define uma função que registra esses logs respeitando dois níveis de severidade:

```
def my_function():
    logging.debug('Some debug data')
    logging.error('Error log here')
    logging.debug('More debug data')
```

O nível de log default para o programa é WARNING, portanto apenas a mensagem de erro será mostrada na tela quando a função for chamada.

```
my_function()
>>>
Error log here
```

Podemos, temporariamente, elevar o nível de log da função definindo um gerenciador de contexto. Essa função auxiliar aumenta o nível de severidade do log antes de rodar o código circunscrito pelo bloco with e reduz a severidade depois que o bloco tiver sido executado.

```
@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
```

```
    finally:
        logger.setLevel(old_level)
```

A expressão yield é o ponto no qual o conteúdo do bloco with será executado. Qualquer exceção que apareça no bloco with será levantada novamente pela expressão yield para que seja possível capturá-las na função auxiliar (consulte o Item 40: “Considere usar corrotinas para rodar muitas funções simultaneamente” para uma explicação sobre como isso funciona).

Agora, podemos chamar novamente a mesma função de log, mas no contexto debug_logging. Desta vez, todas as mensagens de depuração são mostradas na tela durante a execução do bloco de with. A mesma função rodando fora do bloco with não mostraria nenhuma das mensagens de depuração.

```
with debug_logging(logging.DEBUG):
    print('Inside:')
    my_function()
print('After:')
my_function()
>>>
Inside:
Some debug data
Error log here
More debug data
After:
Error log here
```

Usando os alvos do with

O gerenciador de contextos passado para o comando with também pode retornar um objeto. Esse objeto é atribuído a uma variável local no item as do comando composto. Com isso, o código sendo executado no bloco with ganha o poder de interagir diretamente com seu contexto.

Por exemplo, digamos que se queira escrever em um arquivo e garantir que ele sempre seja fechado corretamente. Podemos fazer isso passando a função open para o comando with. open devolve um manipulador de arquivo (handle) para o alvo as, que é parte do with, e fecha o manipulador quando o código dentro do

bloco with finaliza a execução.

```
with open('/tmp/my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

Essa técnica é muito melhor que abrir e fechar manualmente o manipulador de arquivo toda vez. Assim, temos a segurança de que o arquivo será fechado quando a execução do programa sair de dentro do bloco with. A técnica também encoraja o programador a reduzir a quantidade de código a ser executado enquanto o manipulador mantém o arquivo aberto, o que é sempre uma boa prática a seguir.

Para que suas próprias funções possam fornecer valores para alvos as, é necessário produzir um valor (com yield) a partir do gerenciador de contexto. No exemplo a seguir, definimos um gerenciador de contexto que busca uma instância de Logger, configura seu nível e o entrega (com yield) ao alvo de as.

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

Chamar métodos de log, como o debug do exemplo, no alvo de as produzirá alguma saída porque o nível de segurança de log está ajustado como baixo no bloco with. Usar diretamente o módulo logging não mostraria nenhuma mensagem porque o nível default de severidade de log para o programa principal é WARNING.

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug('This is my message!')
    logging.debug('This will not print')

>>>
This is my message!
```

Depois que o código dentro do with termina, chamar métodos de log em nível de depuração na instância 'my-log' da classe Logger não mostrará nada na tela porque o nível de severidade de log original foi restaurado. As mensagens de log de erros serão sempre impressas, independente do contexto.

```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')
>>>
Error will print
```

Lembre-se

- O comando with permite reusar lógica de blocos try/finally e reduzir a poluição visual.
- O módulo nativo contextlib disponibiliza o decorador contextmanager que permite ao programador usar suas próprias funções com o comando with.
- O valor produzido pelos gerenciadores de contexto são entregues ao trecho as do comando with. É útil para permitir que o código acesse diretamente a causa do contexto especial.

Item 44: Aumente a confiabilidade de pickle com copyreg

O módulo nativo pickle pode serializar os objetos do Python em um fluxo de bytes e desserializar os bytes novamente como objetos. Os fluxos de dados gerados pelo pickle não devem ser usados para comunicação entre dois elementos quando um deles (ou ambos) não for confiável. O objetivo de pickle é permitir a passagem de objetos do Python entre dois programas que você controle usando canais de comunicação exclusivamente binários.

Nota

O formato de serialização do módulo pickle é propositalmente inseguro, foi projetado para ser assim. Os dados serializados contêm o que é, essencialmente, um programa que descreve como reconstruir o objeto original do Python. Isso significa que uma carga maliciosa codificada pelo pickle poderia ser usada para comprometer qualquer parte de um programa em

Python que seja usada para desserializá-la.

O módulo json, pelo contrário, foi projetado para ser seguro. Dados serializados em JSON contêm uma descrição simples da hierarquia do objeto. A desserialização de dados JSON não expõem um programa em Python a nenhum risco adicional. Formatos como o JSON devem ser usados para a comunicação entre programas ou pessoas que não confiam um no outro.

Por exemplo, digamos que se queira usar um objeto do Python para representar o estado do progresso de um jogador em um jogo. O estado do jogo inclui o nível ou fase em que o jogador está e o número de vidas que ele ou ela ainda possuem.

```
class GameState(object):
    def __init__(self):
        self.level = 0
        self.lives = 4
```

O programa modifica o objeto à medida que o jogo se desenrola.

```
state = GameState()
state.level += 1  # Jogador passa de fase
state.lives -= 1  # Jogador morreu e tem que recomeçar a fase
```

Quando o usuário sai do jogo, o programa pode salvar o estado do jogo em um arquivo para que o jogador possa voltar outro dia à posição em que ele estava e continuar jogando. O módulo pickle facilita a implementação de algo assim. No exemplo de código a seguir, gravamos (com dump) o objeto GameState diretamente em um arquivo:

```
state_path = '/tmp/game_state.bin'
with open(state_path, 'wb') as f:
    pickle.dump(state, f)
```

Mais tarde, podemos carregar novamente o arquivo (com load) e obter novamente o objeto GameState como se nunca tivesse sido serializado.

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)
>>>
{'lives': 3, 'level': 1}
```

O problema com essa técnica é o que acontece à medida que mais recursos vão sendo adicionados a cada nova versão do jogo. Imagine um novo recurso no qual o jogador possa ir acumulando pontos para figurar num ranking e evidenciar o resultado mais alto (high score). Para acompanhar os pontos do jogador, é preciso adicionar um novo campo na classe GameState.

```
class GameState(object):
    def __init__(self):
        # ...
        self.points = 0
```

Serializar a nova versão de GameState usando pickle funcionaria como antes, sem problemas. No exemplo de código a seguir, simulamos o caminho de ida e volta: serializamos um objeto em uma string usando dumps, gravamos no arquivo, lemos a string do arquivo e a convertemos em objeto novamente com loads:

```
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'lives': 4, 'level': 0, 'points': 0}
```

Funciona, mas o que aconteceria se quiséssemos abrir objetos GameState mais antigos, de versões anteriores do jogo, salvos em outros arquivos? É justo imaginar que o jogador queira voltar a esses jogos salvos também. No exemplo de código a seguir, se tentarmos desserializar (com pickle) um jogo salvo em uma versão anterior usando uma versão mais nova, com a nova definição da classe GameState:

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)
```

```
>>>
{'lives': 3, 'level': 1}
```

O atributo points não foi encontrado! Isso acaba gerando confusão porque o

objeto devolvido é uma instância da nova versão da classe GameState.

```
assert isinstance(state_after, GameState)
```

Esse comportamento é um subproduto da maneira com que o módulo pickle funciona. Seu principal uso é facilitar a serialização de objetos. No momento em que seu uso de pickle começa a fugir dos usos mais triviais, a funcionalidade do módulo começa a se esfacelar e apresentar comportamentos surpreendentes.

A maneira mais simples e direta de sanar esses problemas é pelo módulo nativo copyreg. O módulo copyreg permite registrar as funções responsáveis pela serialização dos objetos do Python, permitindo o controle do comportamento de pickle e, assim, tornando-o mais confiável.

Valores default dos atributos

No caso mais simples, podemos usar um construtor com argumentos default (consulte o Item 19: “Implemente comportamento opcional usando palavras-chave como argumentos”) para garantir que os objetos de GameState sempre tenham todos os atributos preenchidos depois de descompactados com o pickle. O exemplo de código a seguir redefine o construtor para que funcione dessa maneira:

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0):
        self.level = level
        self.lives = lives
        self.points = points
```

Para usar esse construtor com o pickle, definimos uma função auxiliar que recebe um objeto GameState e o converte em uma tupla de parâmetros para o módulo copyreg. A tupla resultante contém a função usada para o desempacotamento com pickle e os parâmetros a serem passados para a função de “despicklização”.

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs,)
```

Agora, precisamos definir a função auxiliar unpickle_game_state. Essa função recebe parâmetros e dados serializados de pickle_game_state e devolve o objeto

GameState correspondente. Como podemos observar, não passa de uma pequena função que apenas envolve o construtor.

```
def unpickle_game_state(kwargs):
    return GameState(**kwargs)
```

Basta agora registrar essas funções com o módulo nativo copyreg.

```
copyreg.pickle(GameState, pickle_game_state)
```

A serialização e desserialização funcionam como antes:

```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'lives': 4, 'level': 0, 'points': 1000}
```

Com o registro efetuado, podemos alterar a definição de GameState para dar ao jogador uma contagem de feitiços que ele pode usar ao longo do jogo. Essa mudança é semelhante à que fizemos quando adicionamos o campo points a GameState.

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0, magic=5):
        # ...
```

Contudo, desta vez, ao desserializar um objeto GameState de uma versão antiga do jogo, obtemos dados válidos em vez de atributos faltantes. Isso só funciona porque unpickle_game_state chama o construtor GameState diretamente. Os argumentos por palavra-chave do construtor têm valores default que são adotados quando há parâmetros faltando. Com isso, arquivos de jogo salvo de versões anteriores recebem o valor default em seu novo campo magic ao serem desserializados.

```
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'level': 0, 'points': 1000, 'magic': 5, 'lives': 4}
```

Controlando as versões das classes

Às vezes, precisamos fazer modificações em nossos objetos de Python que se mostram totalmente incompatíveis com as versões anteriores desses objetos. Quando é esse o caso, a técnica de usar argumentos com valores default deixa de funcionar.

Por exemplo, digamos que num dado momento o desenvolvedor do jogo perceba não fazer sentido ter um número limitado de vidas e queira remover o conceito de vida. O exemplo de código a seguir redefine o objeto GameState para não mais possuir o campo “lives”:

```
class GameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

A partir desse momento, a desserialização de jogos salvos em versões anteriores para de funcionar. Todos os campos dos dados antigos, mesmo os removidos da classe, serão passados ao construtor GameState pela função unpickle_game_state.

```
pickle.loads(serialized)
>>>
TypeError: __init__() got an unexpected keyword argument 'lives'
```

A solução é incluir nas funções repassadas a copyreg um parâmetro que indique a versão. Os dados serializados na versão nova serão marcados como sendo da versão 2, especificada sempre que um novo objeto GameState for criado.

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs,)
```

Versões antigas não conterão o argumento version, permitindo que manipulemos apropriadamente os argumentos passados ao construtor GameState.

```
def unpickle_game_state(kwargs):
    version = kwargs.pop('version', 1)
    if version == 1:
```

```
        kwargs.pop('lives')
    return GameState(**kwargs)
```

Agora, a desserialização de um objeto antigo funcionará corretamente.

```
copyreg.pickle(GameState, pickle_game_state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
>>>
{'magic': 5, 'level': 0, 'points': 1000}
```

Podemos continuar usando essa técnica para lidar com quaisquer mudanças que ocorram em implementações futuras da mesma classe. Qualquer lógica necessária para adaptar uma versão antiga da classe para um novo modelo pode ser direcionada para a função unpickle_game_state.

Caminhos estáveis de importação

Outro problema com que podemos nos deparar ao empregar pickle é o programa deixar de funcionar quando renomeamos uma classe. É muito frequente, durante o tempo de vida do programa, refatorarmos o código, ou seja, renomear classes e transferi-las de um módulo para outro. Infelizmente, isso acaba por confundir o módulo pickle e fazer com que ele deixe de funcionar, a não ser que sejamos cuidadosos.

No exemplo de código a seguir, renomeamos a classe GameState para BetterGameState, removendo completamente do programa a classe antiga:

```
class BetterGameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

Desserializar um antigo objeto GameState falhará, porque a classe não existe mais:

```
pickle.loads(serialized)
>>>
AttributeError: Can't get attribute 'GameState' on <module
➥'__main__' from 'my_code.py'>
```

A causa dessa exceção é simples: o caminho de importação do objeto serializado

está codificado nos dados anteriormente serializados pelo pickle:

```
print(serialized[:25])
>>>
b'\x80\x03c__main__\nGameState\nq\x00)'
```

A solução é, novamente, usar copyreg. Podemos especificar um identificador estável para a função empregar no momento de desserializar um objeto, permitindo converter dados serializados de uma classe para outras classes com nomes diferentes. A técnica nos dá um certo grau de liberdade para usar vias indiretas.

```
copyreg.pickle(BetterGameState, pickle_game_state)
```

Depois do copyreg, é fácil notar que o caminho de importação de pickle_game_state está codificado nos dados serializados em vez de BetterGameState.

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized[:35])
>>>
b'\x80\x03c__main__\nunpickle_game_state\nq\x00}'
```

A única pegadinha é que não podemos alterar o caminho do módulo no qual a função unpickle_game_state estiver implementada. Uma vez que os dados sejam serializados com uma função, esta deve estar eternamente disponível naquele caminho de importação para possibilitar a desserialização desses dados no futuro.

Lembre-se

- O módulo nativo pickle é útil apenas para serializar e desserializar objetos entre programas confiáveis.
- O módulo pickle pode deixar de funcionar quando usado em casos mais complexos que o trivial.
- Use o módulo nativo copyreg em conjunto com o pickle para incluir valores default a atributos faltantes, permitir o controle de versão das classes e garantir caminhos de importação estáveis.

Item 45: Use datetime em vez de time para relógios locais

O Tempo Coordenado Universal (Coordinated Universal Time, em inglês, mais conhecido pela sua sigla UTC) é a representação-padrão da passagem de tempo, completamente independente de fusos horários. O UTC é ótimo para os computadores: o tempo, para as máquinas, é representado como o número de segundos que se passaram desde um momento no tempo chamado de UNIX Epoch[1]. No entanto, o UTC não é lá muito amigável para nós, humanos. Nossa referência de tempo é sempre relativa ao local em que nos encontramos. As pessoas dizem “meio-dia” ou “oito da noite” em vez de “UTC 15:00 menos 7 horas”. Se um programa lida com tempo, provavelmente o programador teve que criar rotinas de conversão entre o tempo UTC e os relógios locais para que os humanos possam entender.

O Python oferece duas maneiras de efetuar conversões de fuso horário. A maneira antiga, usando o módulo nativo time, é desastrosamente propensa a erros. A nova maneira, com o módulo datetime, também nativo, funciona muito bem com a ajuda de um pacote desenvolvido pela comunidade chamado pytz.

Precisamos conhecer com certa profundidade o funcionamento de ambos os módulos, time e datetime, para entender completamente o porquê de datetime ser melhor e de evitar time a todo custo.

Módulo time

A função localtime do módulo nativo time permite converter um tempo em formato UNIX (segundos decorridos desde a UNIX Epoch com o relógio ajustado para UTC) para um horário local, de acordo com o fuso horário configurado no sistema do computador local. No meu caso, o fuso é chamado de Pacific Daylight Time (Hora Norte-Americana do Pacífico, Horário de Verão).

```
from time import localtime, strftime

now = 1407694710
local_tuple = localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = strftime(time_format, local_tuple)
print(time_str)
```

```
>>>
2014-08-10 11:18:30
```

É preciso fazer o caminho de volta também, pois o usuário fatalmente informará horários no fuso local que devem ser convertidos para tempo Unix em UTC. A função strptime faz isso, analisando a string de entrada e a armazenando numa tupla junto com o formato usado. A tupla é repassada à função mktime, que a converte em um valor de tempo Unix.

```
from time import mktime, strptime


time_tuple = strptime(time_str, time_format)
utc_now = mktime(time_tuple)
print(utc_now)
>>>
1407694710.0
```

Como faríamos para converter o horário de um fuso para outro? Por exemplo, digamos que estejamos em um avião entre São Francisco e Nova York, e gostaríamos de saber qual é o horário em São Francisco no momento em que chegarmos a Nova York.

Manipular diretamente os valores devolvidos pelas funções time, localtime e strptime para realizar conversões de fuso horário é má ideia. Os fusos horários mudam a toda hora devido a leis locais e horário de verão. É extremamente complicado gerenciar tudo isso sozinho, especialmente se quisermos incluir todas as cidades no mundo que tenham voos chegando ou partindo.

Muitos sistemas operacionais têm arquivos de configuração que são atualizados automaticamente quando as informações de fuso horário mudarem. O módulo time permite usar esses fusos horários. Por exemplo, o código a seguir analisa o horário de partida no fuso horário de São Francisco, que é o já mencionado Pacific Daylight Time (representado no código como PDT):

```
parse_format = '%Y-%m-%d %H:%M:%S %Z'
depart_sfo = '2014-05-01 15:45:16 PDT'
time_tuple = strptime(depart_sfo, parse_format)
time_str = strftime(time_format, time_tuple)
print(time_str)
```

```
>>>
2014-05-01 15:45:16
```

Depois de ver como o PDT funciona bem com a função strptime, poderíamos supor que qualquer outro fuso funcionaria da mesma maneira. No entanto, não é o caso. Infelizmente, strptime levanta uma exceção quando converte a partir do Eastern Daylight Time, ou EDT (Hora Norte-Americana do Leste, o fuso horário de Nova York).

```
arrival_nyc = '2014-05-01 23:33:24 EDT'
time_tuple = strptime(arrival_nyc, time_format)
>>>
ValueError: unconverted data remains:  EDT
```

O problema neste caso é a natureza extremamente dependente de plataforma do módulo time. Seu comportamento real é determinado pela maneira como a linguagem C funciona no sistema operacional instalado na máquina. Isso faz com que o módulo time não seja confiável no Python. O módulo time é falho quando mais de um fuso horário está envolvido. Portanto, devemos evitar o uso do módulo time para esse fim. Caso seja absolutamente imprescindível usar time, use-o apenas para converter entre UTC e a hora local do seu computador. Para todos os outros tipos de conversão, use em seu lugar o módulo datetime.

Módulo datetime

A segunda opção para representar horários em Python é a classe datetime do módulo nativo datetime. Assim como no módulo time, o datetime (entre outras coisas) converte a hora local para UTC e vice-versa.

No exemplo de código a seguir, convertemos o horário atual em UTC e depois o convertemos para o horário local do PC, que no meu caso é a Hora do Pacífico (Pacific Daylight Time – PDT):

```
from datetime import datetime, timezone

now = datetime(2014, 8, 10, 18, 18, 30)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)
```

```
>>>
2014-08-10 11:18:30-07:00
```

O módulo datetime pode facilmente converter um horário local para o UTC em formato UNIX.

```
time_str = '2014-08-10 11:18:30'
now = datetime.strptime(time_str, time_format)
time_tuple = now.timetuple()
utc_now = mktime(time_tuple)
print(utc_now)
>>>
1407694710.0
```

Diferente do módulo time, o módulo datetime possui recursos para, de forma confiável, converter de um fuso horário para outro. Contudo, o datetime só fornece o maquinário necessário para as operações de fuso horário empregando a classe tzinfo e métodos relacionados. Não há, no módulo, informações sobre outros fusos que não sejam o próprio UTC.

A comunidade do Python resolveu o problema com um módulo não nativo, pytz, disponível para download no Python Package Index (*https://pypi.python.org/pypi/pytz/*). O pytz contém um banco de dados completo sobre quaisquer fusos horários de que se necessite.

Para usar o pytz de forma eficaz, devemos sempre converter primeiro o horário local para UTC. Depois, faça todas as operações que precisar com o datetime sobre os valores em UTC (como por exemplo deslocamento). Por fim, converta de UTC para o fuso horário desejado.

Por exemplo, o código a seguir converte o horário de chegada de um voo a NYC para UTC com o datetime. Embora algumas dessas chamadas pareçam redundantes, todas são necessárias quando usamos pytz.

```
arrival_nyc = '2014-05-01 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)
```

```
>>>
2014-05-02 03:33:24+00:00
```

Uma vez tendo um datetime em formato UTC, posso convertê-lo para o horário de São Francisco.

```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)
>>>
2014-05-01 20:33:24-07:00
```

Da mesma forma, é muito fácil converter o mesmo horário para o fuso do Nepal.

```
nepal = pytz.timezone('Asia/Katmandu')
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))
print(nepal_dt)
>>>
2014-05-02 09:18:24+05:45
```

Com datetime e pytz, essas conversões são consistentes em todos os ambientes computacionais independente de qual sistema operacional estiver no comando.

Lembre-se

- Evite usar o módulo time para converter entre fusos horários diferentes.
- Use o módulo nativo datetime junto com o módulo pytz para converter de forma eficiente os horários entre dois fusos distintos.
- Sempre represente o horário em UTC durante o processamento e o converta para o fuso de destino só no final.

Item 46: Use algoritmos e estruturas de dados nativos

Ao implementar programas em Python que lidam com quantidades não triviais de dados, fatalmente experimentaremos momentos de lentidão causados pela complexidade algorítmica de nosso código. Isso não se deve a uma deficiência de desempenho do Python enquanto linguagem (caso seja mesmo culpa do Python, consulte o Item 41: “Considere usar concurrent.futures para obter

paralelismo real”), mas ao fato de que seu código talvez não esteja usando os melhores algoritmos e estruturas de dados possíveis.

Felizmente, a biblioteca-padrão do Python oferece muitos dos algoritmos e estruturas de dados que usaremos em todos os nossos programas, e são todos nativos. Além do ganho em velocidade que isso acarreta, esses algoritmos e estruturas de dados comuns podem tornar sua vida bem mais fácil. Algumas das ferramentas mais valiosas que gostaríamos de usar podem ser intrincadas demais para implementarmos nós mesmos. Evitar a reimplementação de funcionalidade comum economiza tempo e evita dores de cabeça.

Fila de mão dupla

A classe deque, presente no módulo collections, é uma fila com duas extremidades (double-ended queue) ou, se preferir, uma fila de mão dupla. Ela disponibiliza operações de tempo constante para inserir e remover itens tanto do início como do fim da fila, o que a torna ideal para implementar filas do tipo FIFO (first-in-first-out, ou o primeiro que entra é o primeiro que sai).

```
fifo = deque()
fifo.append(1)      # Produtor
x = fifo.popleft()  # Consumidor
```

O tipo nativo list também contém uma sequência ordenada de itens, assim como uma fila. Podemos inserir ou remover itens do fim da lista em tempo constante. Todavia, fazer o mesmo no início da lista consome um tempo linear, o que é muito mais demorado que o tempo constante de deque.

Dicionário ordenado

Os dicionários comuns do Python não são ordenados. Isso significa que dois objetos dict contendo as mesmas chaves e valores podem resultar em ordens diferentes de iteração. Esse comportamento é um subproduto surpreendente da maneira como a tabela de hash rápido dos dicionários é implementada.

```
a = {}
a['foo'] = 1
a['bar'] = 2
```

```
# Populamos 'b' aleatoriamente para causar conflitos de hash
while True:
    z = randint(99, 1013)
    b = {}
    for i in range(z):
        b[i] = i
    b['foo'] = 1
    b['bar'] = 2
    for i in range(z):
        del b[i]
    if str(b) != str(a):
        break

print(a)
print(b)
print('Equal?', a == b)
>>>
{'foo': 1, 'bar': 2}
{'bar': 2, 'foo': 1}
Equal? True
```

A classe OrderedDict do módulo collections é um tipo especial de dicionário que mantém um registro da ordem na qual suas chaves foram inseridas. Interagir com as chaves em um OrderedDict tem comportamento bastante previsível. Isso pode simplificar drasticamente os procedimentos de testes e depuração ao fazer com que todo o código seja absolutamente determinístico.

```
a = OrderedDict()
a['foo'] = 1
a['bar'] = 2
b = OrderedDict()
b['foo'] = 'red'
b['bar'] = 'blue'

for value1, value2 in zip(a.values(), b.values()):
```

```
    print(value1, value2)
>>>
1 red
2 blue
```

Dicionário default

Os dicionários são úteis para fazer registros e controlar estatísticas. Um dos problemas que os acometem é que não se pode considerar que qualquer chave já esteja presente antecipadamente. Com isso, não é muito prático realizar tarefas simples como incrementar um contador armazenado em um dicionário.

```
stats = {}
key = 'my_counter'
if key not in stats:
    stats[key] = 0
stats[key] += 1
```

A classe defaultdict do módulo collections simplifica a vida do programador ao armazenar automaticamente um valor-padrão quando a chave não existir. A única coisa que precisamos definir é uma função que retorne o valor default toda vez que uma chave não possa ser encontrada. No exemplo a seguir, a função nativa int retorna o valor 0 (consulte o Item 23: “Aceite funções para interfaces simples em vez de classes” para ver outro exemplo). Agora, incrementar o contador é bastante simples.

```
stats = defaultdict(int)
stats['my_counter'] += 1
```

Heap Queue, a “fila-pilha”

As pilhas (heaps[2]) são estruturas de dados muito úteis para manter uma fila de prioridade. O módulo heapq oferece funções para criar pilhas dentro de um objeto do tipo list padrão do Python, com funções como heappush, heappop e nsmallest.

Os itens de uma prioridade podem ser inseridos na pilha em qualquer ordem.

```
a = []
heappush(a, 5)
```

```
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)
```

Os itens de maior prioridade (número mais baixo) são sempre removidos primeiro.

```
print(heappop(a), heappop(a), heappop(a), heappop(a))
>>>
3 4 5 7
```

A lista resultante (um objeto list) é fácil de usar fora do heapq. Ao acessar o índice 0 da pilha, sempre obtemos o item menor.

```
a = []
heappush(a, 5)
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)
assert a[0] == nsmallest(1, a)[0] == 3
```

Chamar o método sort no objeto list não altera nada na pilha.

```
print('Before:', a)
a.sort()
print('After: ', a)
>>>
Before: [3, 4, 7, 5]
After:  [3, 4, 5, 7]
```

Cada uma dessas operações de heapq consome um tempo logarítmico que é proporcional ao comprimento da lista. Fazer a mesma operação com uma lista comum do Python consome apenas tempo linear.

Bisseção

Procurar por um item usando o método index em um objeto list consome um tempo que aumenta de forma linear à medida que a lista cresce.

```
x = list(range(10**6))
i = x.index(991234)
```

As funções do módulo bisect, como, por exemplo, bisect_left, executam uma busca binária bem eficiente ao longo de uma sequência de itens ordenados. O índice devolvido é o ponto de inserção do valor na sequência.

```
i = bisect_left(x, 991234)
```

A complexidade de uma busca binária é logarítmica. Isso significa que usar bisect para buscar um valor em uma lista de um milhão de itens consome mais ou menos o mesmo tempo que usar index para fazer uma busca linear em uma lista de 14 itens. É muito mais rápido!

Ferramentas de iteração

O módulo nativo itertools contém uma grande quantidade de funções muito úteis para organizar e manipular os iteradores (consulte o Item 16: “Prefira geradores em vez de retornar listas” e o Item 17: “Seja conservador quando iterar sobre argumentos” para saber mais). Nem todos eles estão disponíveis no Python 2, mas podem ser construídos à mão usando receitas simples documentadas no módulo. Consulte help(itertools) em uma sessão interativa do Python para mais detalhes.

As funções de itertools recaem em três categorias principais:

- Reunir iteradores
  - chain: Combina múltiplos iteradores em um único iterador sequencial.
  - cycle: Repete ciclicamente e sem interrupção os itens de um iterador.
  - tee: Divide um único iterador em múltiplos iteradores paralelos.
  - zip_longest: Uma variante da função nativa zip que funciona muito bem com iteradores de tamanhos diferentes.
- Filtrar itens em um iterador
  - islice: Fatia um iterador pelo seu índice numérico sem criar uma cópia.
  - takewhile: Devolve itens de um iterador enquanto uma função predicada retornar True.
  - dropwhile: Devolve itens de um iterador quando a função predicada devolver False pela primeira vez.
  - filterfalse: Devolve itens de um iterador sempre que uma função predicada devolver False. Exatamente o oposto do que faz a função nativa filter.

- Combinação de itens entre iteradores
  - product: Devolve o produto cartesiano dos itens de um iterador, o que é uma alternativa bastante elegante a abrangências de listas com muitos níveis de aninhamento.
  - permutations: Devolve permutações ordenadas de tamanho N com os itens de um iterador.
  - combination: Devolve as combinações não ordenadas de tamanho N com os itens únicos (sem repetição) de um iterador.

Há ainda mais funções e receitas disponíveis no módulo itertools que não mencionei aqui. Sempre que sentir alguma dificuldade em resolver algum problema intrincado de iteração, vale a pena dar uma olhada na documentação do módulo itertools para ver se há algo lá que possa facilitar sua vida.

Lembre-se

- Use os módulos nativos do Python para algoritmos e estruturas de dados.
- Não reimplemente você mesmo essa funcionalidade. Reinventar a roda é difícil e trabalhoso.

Item 47: Use decimal quando a precisão for de importância vital

O Python é uma linguagem excelente para escrever código que interaja com dados numéricos. O tipo inteiro do Python pode representar valores de qualquer magnitude prática. Já o tipo em ponto flutuante de precisão dupla atende à norma IEEE 754. A linguagem também oferece um tipo-padrão de número complexo para valores imaginários. Entretanto, eles nem sempre são suficientes.

Por exemplo, digamos que se queira computar quanto cobrar de um cliente por uma ligação telefônica internacional. Sabemos o tempo de duração da chamada em minutos e segundos (digamos, 3 minutos e 42 segundos). Também temos uma tarifa-padrão para ligações entre a Antártida e os Estados Unidos ($1,45/minuto). Quanto devemos cobrar?

Com matemática em ponto flutuante, o valor cobrado parece razoável.

```
rate = 1.45
```

```
seconds = 3*60 + 42
cost = rate * seconds / 60
print(cost)
>>>
5.364999999999999
```

Porém, ao arredondar os centavos, percebemos que o Python arredonda para baixo. Gostaríamos de arredondar para cima para evitar prejuízos de arredondamento.

```
print(round(cost, 2))
>>>
5.36
```

Digamos que também se queira oferecer chamadas telefônicas de curtíssima duração entre localidades cujo custo de conexão, para nós, seja extremamente barato. No exemplo de código a seguir, computamos a conta de uma chamada com duração de 5 segundos e tarifa de $0,05/minuto:

```
rate = 0.05
seconds = 5
cost = rate * seconds / 60
print(cost)
>>>
0.004166666666666667
```

O float resultante é tão baixo que arredonda para zero! Aí não pode, né?

```
print(round(cost, 2))
>>>
0.0
```

A solução é usar a classe Decimal, presente no módulo nativo decimal. A classe Decimal oferece recursos de cálculo matemático com 28 casas decimais. Pode ser até mais, se necessário – 28 é apenas o default. A classe tenta contornar os problemas de precisão com número em ponto flutuante introduzidos pela própria norma IEEE 754, além de dar ao programador mais controle sobre o comportamento do arredondamento.

Por exemplo, o código a seguir refaz o cálculo da Antártida com um valor em

Decimal e o resultado é um cálculo exato do valor, em vez de uma aproximação.

```
rate = Decimal('1.45')
seconds = Decimal('222')  # Tempo em segundos: 3*60 + 42
cost = rate * seconds / Decimal('60')
print(cost)
>>>
5.365
```

A classe Decimal tem uma função nativa para arredondamento com o número exato de casas decimais definido pelo código, e com o comportamento (para cima ou para baixo) desejado.

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)
>>>
5.37
```

O método quantize, usado dessa maneira, também lida de forma esplêndida em nosso outro caso de uso, o das ligações baratas de curtíssima duração. No exemplo a seguir, podemos ver que o custo da ligação (variável cost, em tipo Decimal) ainda é menor que 1 centavo para essa chamada em particular.

```
rate = Decimal('0.05')
seconds = Decimal('5')
cost = rate * seconds / Decimal('60')
print(cost)
>>>
0.004166666666666666666666666667
```

Porém, o comportamento do quantize manda arredondar para cima, com um centavo inteiro.

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)
>>>
0.01
```

Embora Decimal funcione muito bem para números de ponto decimal fixo, ainda

mostra limitações em sua precisão (por exemplo, 1/3 será uma aproximação). Para representar números racionais sem limite de precisão, prefira usar a classe Fraction do módulo nativo fractions.

Lembre-se

- O Python tem tipos e classes nativos em módulos que podem representar praticamente todo tipo de valor numérico.
- A classe Decimal é ideal para situações que demandem grande precisão e comportamento previsível de arredondamento, como é o exemplo do cálculo de valores monetários.

Item 48: Saiba onde encontrar os módulos desenvolvidos pela comunidade

O Python tem um repositório central de módulos chamado Python Package Index, ou simplesmente PyPI (*https://pypi.python.org*) para instalar e usar em seus programas. Esses módulos são criados e mantidos por pessoas como você, membros da Grande Comunidade do Python. Quando nos deparamos com um desafio que não nos é familiar, o PyPI é o melhor lugar para procurar por código pronto que o deixará mais perto de seu objetivo.

Para usar PyPI, é preciso operar uma ferramenta de linha de comando chamada pip. O pip já vem instalado no Python 3.4 e posteriores (também acessível no shell com o comando python -m pip). Para as versões anteriores, as instruções de instalação manual do pip estão na página do Guia do Usuário para Empacotamento do Python, no endereço *https://packaging.python.org*.

Uma vez instalado, usar o pip para instalar um novo módulo é muito simples. Por exemplo, o comando a seguir instala o módulo pytz que eu sei em outro item deste mesmo capítulo (consulte o Item 45: “Use datetime em vez de time para relógios locais”):

```
$ pip3 install pytz
Downloading/unpacking pytz
  Downloading pytz-2014.4.tar.bz2 (159kB): 159kB downloaded
  Running setup.py (...) egg_info for package pytz
```

```
  Installing collected packages: pytz
    Running setup.py install for pytz

  Successfully installed pytz
  Cleaning up...
```

No exemplo, usei o comando pip3 para instalar a versão do módulo correspondente ao Python 3. O comando pip (sem o 3) instala os pacotes para o Python 2. A maioria dos pacotes mais populares já está disponível para ambas as versões do Python (consulte o Item 1: “Saiba qual versão de Python está em uso”). O pip também pode ser usado em conjunto com o pyvenv para analisar conjuntos de pacotes que possivelmente importaremos em algum projeto (consulte o Item 53: “Use ambientes virtuais para criar dependências isoladas e reprodutíveis”).

Cada módulo do PyPI tem sua própria licença de uso de software. A maioria dos pacotes, especialmente os bastante populares, estão sob licenças de software livre ou de código aberto (consulte *http://opensource.org* para mais detalhes). Na maioria dos casos, essas licenças permitem que você inclua uma cópia do módulo em seu programa. Em caso de dúvida, consulte sempre um advogado!

Lembre-se

- O Python Package Index (PyPI) contém um tesouro de pacotes úteis criados e mantidos pela comunidade do Python.
- pip é o comando usado para instalar pacotes vindos do PyPI. Para o Python 3, use o comando pip3.
- O pip já vem instalado no Python 3.4 e posteriores, mas é preciso instalá-lo manualmente nas versões mais antigas.
- A maioria dos módulos do PyPI tem licença livre ou de código aberto.

---

1 N. do T.: A UNIX Epoch, também chamado de UNIX Time ou POSIX Time, é um momento no tempo, especificamente, a zero hora do dia 01/01/1970. É puramente arbitrária, pois o Unix começou a ser desenvolvido antes disso, em 1969, e o registro de passagem de tempo só foi implementado em 1972. O UTC é outro padrão ligado a tempo: é a hora mundial no meridiano zero, o antigo Meridiano de Greenwich. O autor, talvez por brevidade, misturou esses dois conceitos, que politicamente não têm relação um com o outro, mas que dentro do computador são complementares: o horário UTC é a quantidade de segundos que se passaram desde a UNIX Epoch, estando a máquina situada sobre o meridiano zero.

2 N. do T.: Em outras linguagens de programação, como C e Java, há uma distinção técnica entre stack e heap – são coisas diferentes. O problema é que, em português, ambos os termos são traduzidos como pilha, gerando confusão. Alguns autores usam os termos, um tanto errôneos, de “a pilha stack” e “a pilha heap” para contornar o problema e diferenciar entre as duas ideias. Em Python, contudo, não há essa distinção e o programador não precisa se preocupar com isso, portanto neste livro traduzimos heap simplesmente como pilha.

CAPÍTULO 7

Colaboração

O Python tem recursos especialmente criados para implementar APIs bem definidas, com limites de interface muito claros. A comunidade do Python estabeleceu boas práticas para maximizar a facilidade de manutenção do código à medida que envelhece. Existem ferramentas nativas que permitem aos membros de grandes equipes o trabalho em conjunto mesmo em ambientes bastante heterogêneos.

Colaborar com outras pessoas em projetos de Python requer certa dose de atenção na maneira de escrever o código. Mesmo se estiver trabalhando sozinho, existe sempre a possibilidade de você incluir em seu projeto algum trecho de código escrito por outra pessoa, presente na biblioteca-padrão ou em algum pacote externo de código aberto. É importante entender os mecanismos que tornam tão fácil colaborar com outros programadores de Python.

Item 49: Escreva docstrings para toda e qualquer função, classe e módulo

Para o Python, a documentação é extremamente importante devido à natureza dinâmica da linguagem. O Python permite incluir documentação diretamente a blocos de código. Ao contrário de muitas outras linguagens, a documentação presente no código-fonte de um programa pode ser acessada diretamente enquanto o programa está rodando.

Por exemplo, podemos documentar uma função simplesmente incluindo uma *docstring* imediatamente após o comando def.

```
def palindrome(word):
    """Return True if the given word is a palindrome."""
    return word == word[::-1]
```

Podemos obter a docstring diretamente de dentro do próprio programa acessando o atributo especial __doc__, presente em qualquer função.

```
print(repr(palindrome.__doc__))
>>>
'Return True if the given word is a palindrome.'
```

As docstrings podem ser inseridas em funções, classes e módulos. Essa conexão é parte do processo de compilar e rodar um programa em Python. A existência das docstrings e do atributo __doc__ tem três consequências:

- A acessibilidade da documentação facilita o desenvolvimento interativo. Pode-se inspecionar funções, classes e módulos para ler sua documentação usando a função nativa help, fazendo com que o interpretador interativo do Python (conhecido como “shell do Python”) e ferramentas como o IPython Notebook[1] (*http://ipython.org*) sejam uma delícia de usar durante o desenvolvimento de algoritmos, teste de APIs e a criação de pequenos trechos utilitários de código.
- Ter uma maneira padronizada de definir documentação faz com que seja mais fácil converter o texto em formatos mais agradáveis visualmente (como o HTML). Por isso mesmo, surgiram ferramentas de geração de documentação de excelente qualidade, criadas pela comunidade do Python, como o Sphinx (*http://sphinx-doc.org*). Também permitiu que sites financiados pela comunidade como o Read the Docs (*https://readthedocs.org*), que oferece hospedagem gratuita para documentação lindamente formatada de projetos de código aberto do Python.
- A documentação acessível, de primeira classe e luxuosamente apresentada do Python encoraja as pessoas a escrever mais documentação. Os membros da comunidade do Python têm fé inabalável na importância da documentação. Consideram que código bom é código bem documentado. Isso significa que a maioria das bibliotecas open source do Python têm documentação decente.

Para participar desse verdadeiro movimento em prol da boa documentação, é necessário obedecer a algumas regrinhas quando for escrever docstrings. Os detalhes e orientações a respeito da boa redação de docstrings estão descritas na PEP 257 (*http://www.python.org/dev/peps/pep-0257/*), mas há algumas boas práticas que devemos sempre seguir.

Documentando módulos

Todo módulo deve, sem exceção, ter uma docstring no nível mais alto. Esse

literal em formato string é sempre a primeira declaração em um arquivo-fonte. As docstrings devem usar três aspas duplas ("""). O objetivo dessa primeira docstring é apresentar o módulo, dizer o que ele faz e o que contém.

A primeira linha da docstring deve ser uma única sentença descrevendo a finalidade do módulo. Os parágrafos seguintes devem conter os detalhes que todos os usuários desse módulo precisam saber para bem usá-lo. A docstring do módulo também serve como ponto de partida, no qual podemos destacar as classes e funções mais importantes.

O exemplo a seguir mostra uma típica docstring de módulo:

```
# words.py
#!/usr/bin/env python3
"""Library for testing words for various linguistic patterns.

Testing how words relate to each other can be tricky sometimes!
This module provides easy ways to determine when words you've
found have special properties.

Available functions:
- palindrome: Determine if a word is a palindrome.
- check_anagram: Determine if two words are anagrams.
...
"""


# ...
```

Se o módulo for uma ferramenta que pode ser usada na linha de comando, sua docstring também é um ótimo lugar para colocar as instruções de uso.

Documentando classes

Toda classe deve ter uma docstring principal, logo no início da classe, que segue em grande parte os mesmos moldes da docstring geral do módulo. A primeira linha traz uma única sentença que descreve a finalidade da classe. Os parágrafos seguintes discutem os detalhes importantes de como a classe funciona.

Os métodos e atributos públicos importantes devem ser destacados na docstring

da classe. Deve também haver orientação sobre como criar subclasses que interajam com atributos protegidos (consulte o Item 27: “Prefira atributos públicos em vez de privativos”) e os métodos da superclasse.

O exemplo a seguir mostra a docstring de uma classe:

```
class Player(object):
    """Represents a player of the game.

    Subclasses may override the 'tick' method to provide
    custom animations for the player's movement depending
    on their power level, etc.

    Public attributes:
    - power: Unused power-ups (float between 0 and 1).
    - coins: Coins found during the level (integer).
    """

    # ...
```

Documentando funções

Todo método ou função públicos precisam, necessariamente, ter uma docstring. Novamente, os moldes são mais ou menos os mesmos das classes e módulos. A primeira linha é a descrição, em uma frase, do que a função faz. O parágrafo seguinte descreve qualquer comportamento específico e os argumentos da função. Os valores de retorno esperados devem também ser descritos. Quaisquer exceções que os chamadores devam tratar como parte da interface da função devem ser explicados.

O exemplo a seguir mostra a docstring de uma função:

```
def find_anagrams(word, dictionary):
    """Find all anagrams for a word.

    This function only runs as fast as the test for
    membership in the 'dictionary' container. It will
    be slow if the dictionary is a list and fast if
```

```
    it's a set.

    Args:
        word: String of the target word.
        dictionary: Container with all strings that
            are known to be actual words.

    Returns:
        List of anagrams that were found. Empty if
        none were found.
    """
    # ...
```

Há ainda alguns casos especiais a considerar quando escrevemos docstrings para funções:

- Se a função não pede nenhum argumento e possui um valor de retorno muito simples, uma única frase descrevendo esse comportamento é mais que suficiente.
- Se a função não retornar nada, é melhor suprimir qualquer menção ao valor de retorno em vez de dizer “não retorna nada” (em inglês, “returns None.”[2]).
- Algumas funções levantam exceções durante a operação normal e não apenas em caso de erro; se a sua função for desse tipo, documente o fato; se não for, nem mencione.
- Caso a função aceite um número variável de argumentos (consulte o Item 18: “Reduza a poluição visual com argumentos opcionais”) ou argumentos por palavra-chave (consulte o Item 19: “Implemente comportamento opcional usando palavras-chave como argumentos”), use *args e **kwargs na lista de argumentos documentados para descrever seu propósito.
- Se a função tiver argumentos com valores default, os valores devem ser mencionados e claramente identificados como default (consulte o Item 20: “Use None e docstrings para especificar argumentos default dinâmicos e específicos”).
- Nas funções geradoras (consulte o Item 16: “Prefira geradores em vez de retornar listas”), a docstring deve descrever o que o gerador produz (com

yield) quando for iterado.

- Se a função for uma corrotina (consulte o Item 40: “Considere usar corrotinas para rodar muitas funções simultaneamente”), a docstring deve conter o que a corrotina produz (com yield), o que se espera receber das expressões yield e quando a iteração será encerrada.

Nota

Depois de escrever as docstrings para seus módulos, é importante manter a documentação atualizada. O módulo nativo `doctest` torna fácil testar os exemplos de uso descritos nas docstrings para garantir que o código-fonte e sua documentação não divirjam com o passar do tempo.

Lembre-se

- Escreva documentação para todo e qualquer módulo, classe e função usando docstrings. Mantenha a documentação atualizada à medida que o código evolui.
- Para módulos: apresente o conteúdo do módulo e inclua quaisquer classes e funções importantes que os usuários do módulo precisam, necessariamente, conhecer e entender.
- Para classes: documente o comportamento, atributos importantes e comportamento de subclasses na docstring logo na linha imediatamente seguinte ao comando class.
- Para funções e métodos: documente cada argumento, valor de retorno, exceções geradas e outros comportamentos na docstring logo após o comando def.

Item 50: Use pacotes para organizar módulos e criar APIs estáveis

À medida que o tamanho do programa (e de seu código-fonte) cresce, é natural que queiramos reorganizar a sua estrutura interna. Dividimos funções complicadas em várias funções mais simples, refatoramos as estruturas de dados em classes auxiliares (consulte o Item 22: “Prefira classes auxiliares em vez de administrar registros complexos com dicionários e tuplas”) e segregamos cada funcionalidade em um módulo em separado, criando dependências entre eles

para que trabalhem juntos.

Chegará o dia em que nos veremos com tantos módulos para administrar que será preciso mais uma camada de abstração no programa apenas para torná-lo inteligível. Para isso, o Python tem o recurso de *pacotes*, ou, em inglês, *packages*. Os pacotes são módulos que contêm outros módulos.

Em muitos casos, os pacotes são definidos pela criação de um arquivo vazio chamado __init__.py em uma pasta. Quando o __init__.py está presente, quaisquer outros arquivos de Python naquela pasta ficarão disponíveis para importação usando um caminho relativo a ela. Por exemplo, imagine que tenhamos a seguinte estrutura de diretórios em nosso programa:

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

Para importar o módulo utils, usamos o nome absoluto do módulo, o que inclui o nome da pasta que contém o módulo.

```
# main.py
from mypackage import utils
```

Esse padrão é ainda válido quando temos pastas de pacotes dentro de outros pacotes (por exemplo, mypackage.foo.bar).

Nota

O Python 3.4 introduziu o conceito de *namespace packages*, uma maneira mais flexível de definir pacotes. Os namespace packages podem ser compostos de módulos espalhados em pastas diferentes, arquivos compactados com zip ou mesmo em sistemas remotos. Para saber mais sobre os recursos avançados nos namespace packages e como usá-los, consulte a PEP 420 (*http://www.python.org/dev/peps/pep-0420/*).

A funcionalidade oferecida pelos packages tem dois usos muito importantes para nossos programas em Python.

Namespaces

O primeiro uso dos pacotes é dividir nossos módulos em namespaces (literalmente, espaços de nomes), o que permite ter muitos módulos com o

mesmo nome, mas com caminhos absolutos, diferentes e únicos. No exemplo a seguir, o programa importa atributos de dois módulos com o mesmo nome, utils.py. A importação não dá erro porque os módulos são chamados pelo seu caminho absoluto.

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify

bucket = stringify(log_base2_bucket(33))
```

Contudo, ainda é possível que erros sejam encontrados caso as funções, classes ou submódulos definidos nos pacotes tenham nomes repetidos. Por exemplo, digamos que se queira usar a função inspect presente nos dois módulos, analysis.utils e frontend.utils. A importação direta dos atributos não funcionará porque o segundo comando import sobrescreve o valor de inspect no escopo atual.

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect  # Grava por cima!
```

A solução é usar a cláusula as, parte do comando import, para renomear o que quer que tenha sido importado para o escopo atual.

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Inspection equal!')
```

A cláusula as pode ser usada para renomear qualquer coisa adquirida pelo comando import, inclusive módulos inteiros. Assim, é fácil acessar código organizado em namespaces e deixar clara sua identidade.

Nota

Outra maneira de evitar conflitos de nomes em módulos importados é sempre

acessá-los pelo seu nome único de nível mais alto.

Para o exemplo que acabamos de ver, poderíamos primeiramente fazer `import analysis.utils` e `import frontend.utils`. Depois, poderíamos acessar as funções `inspect` pelos seus caminhos completos: `analysis.utils.inspect` e `frontend.utils.inspect`.

Dessa maneira, evitamos ter de usar a cláusula `as` e deixamos o código abundantemente claro, de forma que novos leitores não familiarizados com ele saibam com toda a certeza o local em que as funções foram definidas.

APIs estáveis

O segundo uso dos pacotes no Python é criar APIs estáveis e rigorosas para consumidores externos.

Quando uma API precisa atender a um público amplo (por exemplo, um pacote de código aberto – consulte o Item 48: “Saiba onde encontrar os módulos desenvolvidos pela comunidade”), é de suma importância oferecer funcionalidade estável e que não mude de uma versão para outra. Esse tipo de garantia implica, obrigatoriamente, esconder dos usuários externos a organização interna do código, permitindo que possamos refatorar e aprimorar os módulos internos do pacote sem que os programas que empregam nosso pacote deixem de funcionar.

O Python pode limitar a superfície da API exposta aos consumidores usando o atributo especial `__all__` de um módulo ou pacote. O valor de `__all__` é uma lista de cada nome que o módulo exportará como parte de sua API pública. Quando o código consumidor declara `from foo import *`, apenas os atributos em `foo.__all__` serão importados de `foo`. Se `__all__` não estiver presente em `foo`, todos os atributos públicos, ou seja, aqueles que não têm um underscore no início do nome, são importados (consulte o Item 27: “Prefira atributos públicos em vez de privativos”).

Por exemplo, digamos que se queira oferecer um pacote para calcular a ocorrência de colisões entre projéteis. No exemplo a seguir, definimos que o módulo `models` de nosso pacote `mypackage` conterá a representação dos projéteis:

```
# models.py
__all__ = ['Projectile']
```

```
class Projectile(object):
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

Também definimos um módulo utils em mypackage para fazer operações nas instâncias de projéteis (classe Projectile), como simular colisões entre eles.

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    # ...

def simulate_collision(a, b):
    # ...
```

Agora, gostaríamos de oferecer todas as partes públicas desta API como um conjunto de atributos disponíveis no módulo mypackage. Isso permite que os consumidores deste módulo em seus projetos (os chamados consumidores downstream[3]) sempre importem diretamente de mypackage em vez de ter que importar de mypackage.models ou mypackage.utils. Com isso, o código consumidor que acessa a API continuará a funcionar mesmo se a organização interna de mypackage mudar (por exemplo, caso models.py seja deletado).

Para fazer isso nos pacotes do Python, é preciso modificar o arquivo __init__.py na pasta mypackage. Esse arquivo, na realidade, torna-se o índice de conteúdo do módulo mypackage quando importado. Assim, pode-se especificar uma API explícita para mypackage simplesmente limitando o que é possível importar para __init__.py. Uma vez que todos os módulos internos já especificam __all__, podemos expor a interface pública de mypackage simplesmente importando tudo o que estiver nos módulos internos e atualizando __all__ para refletir isso.

```
# __init__.py
__all__ = []
from . models import *
```

```
__all__ += models.__all__
from . utils import *
__all__ += utils.__all__
```

O exemplo a seguir mostra um consumidor da API que importa diretamente mypackage em vez de acessar os módulos internos:

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a, after_b = simulate_collision(a, b)
```

Salta aos olhos o fato de que funções como mypackage.utils._dot_product não estarão disponíveis ao consumidor da API de mypackage porque não estão presentes em __all__. Não estar listado em __all__ significa que eles não serão importados pelo comando from mypackage import *. Os nomes internos são, efetivamente, ocultos.

A técnica funciona bem quando é importante oferecer uma API estável e explícita. Entretanto, se estamos construindo uma API para uso interno de nossos próprios módulos, a funcionalidade de __all__ é provavelmente desnecessária e deve ser evitada. Os namespaces oferecidos pelos pacotes são normalmente suficientes para uma equipe de programadores que trabalha com grandes quantidades de código sob seu controle e, ao mesmo tempo, mantém limites de interface razoáveis.

Cuidado com o import *

Importações do tipo `from x import y` são claras porque a fonte de `y` é, explicitamente, o pacote ou módulo `x`. Importações com coringas como `from foo import *` podem ser úteis, especialmente em sessões interativas do Python. Entretanto, os coringas tornam o código mais difícil de entender.

- `from foo import *` esconde dos novos leitores do código a origem dos nomes. Se um módulo tiver múltiplos `import *`, será preciso verificar todos os módulos referenciados para descobrir onde determinado nome foi definido.
- Os nomes vindos de `import *` substituirão quaisquer nomes conflitantes

dentro do módulo mais externo. Isso pode levar a bugs estranhos causados pela interação acidental entre o seu código e os nomes sobrepostos de múltiplos comandos `import *`.

O melhor é sempre evitar o uso de `import *` em seu código e explicitamente importar nomes no formato `from x import y`.

Lembre-se

- Os pacotes no Python são módulos que contêm outros módulos. Os pacotes permitem organizar o código em namespaces separados e não conflitantes, com nomes de módulo únicos e absolutos.
- Pacotes simples podem ser definidos adicionando um arquivo __init__.py em uma pasta que contém outros arquivos-fonte. Esses arquivos tornam-se os módulos-filho do pacote que leva o nome da pasta. As pastas dos pacotes podem conter outros pacotes.
- Podemos criar uma API explícita para um módulo listando seus nomes visíveis publicamente no atributo especial __all__.
- Podemos esconder a implementação interna de um pacote, bastando para isso importar apenas nomes públicos no arquivo __init__.py do pacote, ou batizando os membros privativos com um underscore como primeiro caractere de seu nome.
- Se estivermos trabalhando com apenas uma equipe ou com uma base única de código, empregar __all__ para criar APIs explícitas é, provavelmente, desnecessário.

Item 51: Defina uma Exception-raiz para isolar chamadores e APIs

Ao definir a API de um módulo, as exceções criadas são parte da interface tanto quanto as funções e classes definidas (consulte o Item 14: “Prefira exceções em vez de devolver None”).

O Python tem uma hierarquia nativa de exceções para a linguagem e a biblioteca-padrão. Há uma regra tácita na comunidade para usar sempre os tipos nativos de exceção para reportar erros em vez de definir seus próprios novos

tipos. Por exemplo, podemos levantar uma exceção ValueError sempre que um parâmetro inválido é passado para a função.

```
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Density must be positive')
    # ...
```

Em alguns casos, ValueError é o mais apropriado, mas para APIs (contrariando a regra tácita citada) é muito mais vantajoso definir sua própria hierarquia de exceções. Podemos fazê-lo definindo uma exceção-raiz chamada Exception no módulo. Depois, todas as outras exceções levantadas por aquele módulo herdam a exceção-raiz.

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""


class InvalidDensityError(Error):
    """There was a problem with a provided density value."""
```

Ter uma exceção-raiz que é própria do módulo facilita a captura, pelos consumidores da API, de todas as exceções que o módulo levanta propositalmente. Por exemplo, o código a seguir mostra o consumidor de uma API fazendo uma chamada a função em uma estrutura try/except que captura a exceção-raiz do módulo:

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error as e:
    logging.error('Unexpected error: %s', e)
```

Esse try/except evita que as exceções da API se propaguem para os níveis superiores na hierarquia de classes e, assim, acabem por causar um erro no programa chamador. O try/except isola da API o código chamador, e esse isolamento acarreta três efeitos muito úteis.

Primeiro, as exceções-raiz permitem que os chamadores entendam quando há um problema em seu uso da API. Se os chamadores usam a API de forma apropriada, eles devem ser capazes de capturar as várias exceções elevadas

deliberadamente pelo nosso módulo. Se não o fizerem, a exceção se propagará por toda a hierarquia até se deparar com o bloco isolador do except, que captura a exceção-raiz do módulo. Esse bloco chama a atenção do consumidor da API para a exceção elevada, dando a eles a chance de adicionar código para tratar especificamente esse tipo de exceção.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
```

A segunda vantagem de usar exceções-raiz é que elas podem ajudar a encontrar bugs no código-fonte da API. Se o código deliberadamente eleva apenas exceções definidas dentro da hierarquia do módulo, quaisquer outras exceções levantadas pelo módulo serão, necessariamente, exceções que não tínhamos a intenção de gerar. Em última análise, elas indicam que há bugs no código da nossa API.

Usar a estrutura try/except acima não protegerá os consumidores da API de bugs no código. Para isolar-se desses bugs, o chamador precisará incluir outro bloco except que capture a classe Exception, nativa do Python. Dessa forma, o consumidor pode detectar também os bugs no código de nossa API que precisam ser consertados.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
except Exception as e:
    logging.error('Bug in the API code: %s', e)
    raise
```

O terceiro impacto de usar exceções-raiz é deixar a API à prova de futuro. À medida que o sistema evolui, vamos querer expandir a API para oferecer

exceções mais específicas em certas situações. Por exemplo, poderíamos querer adicionar uma subclasse de Exception que indique uma condição de erro especial quando são informadas densidades negativas.

```
# my_module.py
class NegativeDensityError(InvalidDensityError):
    """A provided density value was negative."""


def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError
```

O código chamador continuará funcionando exatamente como antes porque ele já captura exceções InvalidDensityError (a classe-mãe de NegativeDensityError). No futuro, o chamador pode decidir criar um novo tipo de exceção para esse caso especial e alterar o comportamento de acordo com a necessidade.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError as e:
    raise ValueError('Must supply non-negative density') from e
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
except Exception as e:
    logging.error('Bug in the API code: %s', e)
    raise
```

Podemos melhorar ainda mais a segurança futura da API ao oferecer um conjunto mais amplo de exceções diretamente abaixo da exceção-raiz. Por exemplo, imagine que tenhamos um conjunto de erros relacionados ao cálculo de pesos, outro relacionado ao cálculo de volume e um terceiro ao cálculo de densidade.

```
# my_module.py
class WeightError(Error):
```

```
    """Base-class for weight calculation errors."""

class VolumeError(Error):
    """Base-class for volume calculation errors."""

class DensityError(Error):
    """Base-class for density calculation errors."""
```

Exceções específicas devem herdar dessas exceções gerais. Cada exceção intermediária atua como um tipo especial de exceção-raiz, o que facilita criar níveis de isolamento entre código chamador e API baseados na funcionalidade desejada. Isso é muito melhor que fazer os chamadores terem de processar de uma só vez uma longa lista subclasses Exception específicas.

Lembre-se

- Definir exceções-raiz para seus módulos permite isolar da API os consumidores de seu código.
- A captura de exceções-raiz pode ajudar a encontrar bugs no código chamador que consome uma API.
- A captura da classe básica Exception do Python pode ajudar a encontrar bugs na própria API.
- Exceções-raiz intermediárias permitem adicionar mais tipos específicos de exceções no futuro sem prejudicar o código dos consumidores da API.

Item 52: Saiba como romper dependências circulares

Inevitavelmente, quando colaboramos com outros programadores, encontraremos uma interdependência mútua entre módulos. Isso pode até mesmo ocorrer quando trabalhamos sozinhos em um programa muito grande, dividido em inúmeras partes.

Por exemplo, digamos que se queira uma aplicação com interface gráfica (GUI) que mostre uma caixa de diálogo para que o usuário escolha onde salvar um documento. Os dados mostrados pela caixa devem ser especificados por meio de argumentos em seus manipuladores de evento (event handlers). Porém, o diálogo também precisa ler o estado global do programa para obter coisas como

preferências do usuário e afins para que possa renderizá-la corretamente.

No exemplo a seguir, definimos uma caixa de diálogo que lê a partir das preferências globais o local-padrão para salvamento do arquivo:

```
# dialog.py
import app

class Dialog(object):
    def __init__(self, save_dir):
        self.save_dir = save_dir
    # ...
save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    # ...
```

O problema é que o módulo app que contém o objeto prefs também importa a classe dialog para mostrar a caixa de diálogo no início do programa.

```
# app.py
import dialog

class Prefs(object):
    # ...
    def get(self, name):
        # ...

prefs = Prefs()
dialog.show()
```

É uma dependência circular. Ao tentar usar o módulo app do programa principal, teremos uma exceção ao importá-lo:

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    import app
  File "app.py", line 4, in <module>
```

```
    import dialog
  File "dialog.py", line 16, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: 'module' object has no attribute 'prefs'
```

Para entender o que está acontecendo, é preciso conhecer os detalhes do maquinário de importação do Python. Quando um módulo é importado, o que o Python na verdade faz é, nesta ordem (e do elemento mais interno em direção ao mais externo):

1. Procura por seu módulo nos locais determinados em sys.path.
2. Carrega o código do módulo e garante que ele seja compilado.
3. Cria um objeto de módulo correspondente que esteja vazio.
4. Insere o módulo em sys.modules.
5. Roda o código do objeto do módulo para definir seu conteúdo.

O problema com as dependências circulares é que os atributos de um módulo não estão ainda definidos até que o código para esses atributos seja executado (depois do passo 5). No entanto, o módulo pode ser carregado com o comando import imediatamente antes de ser inserido em sys.modules (depois do passo 4).

No exemplo anterior, o módulo app importa dialog antes que qualquer coisa tenha tido tempo de ser definida. Depois, o módulo dialog importa app. Como app ainda não foi executado completamente, pois está esperando o fim da importação de dialog, o módulo app é apenas uma casca vazia (criada no passo 4). A exceção AttributeError é elevada (durante o passo 5 de dialog) porque o código que define prefs ainda não foi executado (o passo 5 de app ainda não está completo).

A melhor solução para esse problema é refatorar o código para que a estrutura de dados de prefs esteja no final da árvore de dependências. Assim, tanto app quanto dialog podem importar o mesmo módulo auxiliar e evitar as dependências circulares. Infelizmente, essa divisão de tarefas nem sempre é tão clara ou possível, e numa situação extrema (mas bastante comum) pode até mesmo requerer muito trabalho de refatoração para valer o esforço.

Há, contudo, outras maneiras de desmanchar dependências circulares.

Reordenação das importações

A primeira técnica é alterar a ordem das importações. Por exemplo, se importarmos o módulo dialog mais para o fim do módulo app, depois que seu conteúdo tiver sido executado, a exceção AttributeError deixa de existir.

```
# app.py
class Prefs(object):
    # ...

prefs = Prefs()

import dialog  # Transferido para cá
dialog.show()
```

Isso funciona porque, quando o módulo dialog é carregado mais tarde, a importação recursiva de app dentro dele vai encontrar um app.prefs já definido (passo 5 já está quase terminado para o módulo app).

Embora evitemos o AttributeError, essa técnica fere o disposto no Guia de Estilo PEP 8 (consulte o Item 2: “Siga o Guia de Estilo PEP 8”). O guia sugere que sempre coloquemos os imports no início dos arquivos em Python. Isso esclarece, para quem for ler seu código-fonte mais tarde, quais são as dependências desse arquivo, além de garantir que qualquer módulo do qual seu código dependa está no escopo correto e disponível para todo o código em seu módulo.

Posicionar as importações mais tarde no arquivo pode deixar seu código muito frágil e fazer com que pequenas mudanças nele causem mau funcionamento do módulo. Devemos, portanto, evitar reordenar as importações para resolver problemas de dependência circular.

Importar, configurar, executar

Uma segunda solução para as importações circulares é fazer com que os módulos minimizem os efeitos colaterais no momento da importação. Nossos módulos devem apenas definir funções, classes e constantes. A execução desses elementos deve ser evitada. Cada um dos módulos deve fornecer uma função própria de configuração (chamada, por exemplo, de configure) que podemos chamar depois que todos os módulos terminarem de ser importados. A finalidade de configure é preparar o estado de cada módulo acessando os atributos de outros módulos. Podemos rodar configure depois que todos os módulos tiverem

sido importados (passo 5 estiver completado em todos), portanto todos os atributos já estarão definidos.

No exemplo a seguir, redefinimos o módulo dialog para acessar o objeto prefs apenas quando configure for chamado:

```
# dialog.py
import app

class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    # ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

Também redefinimos o módulo app para não executar nenhuma atividade durante a importação.

```
# app.py
import dialog

class Prefs(object):
    # ...

prefs = Prefs()

def configure():
    # ...
```

Por fim, o módulo main possui três fases distintas de execução: importar tudo, configurar tudo (com configure) e executar a primeira atividade.

```
# main.py
```

```
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

Funciona bem em muitas situações e permite empregar padrões de código como, por exemplo, *injeção de dependência*. Às vezes, pode ser difícil estruturar o código para que seja possível definir um passo explícito para o configure. Ter duas fases distintas em um módulo também torna seu código mais difícil de ler porque o local em que definição dos objetos foi feita está muito distante do local onde está sua configuração.

Importação dinâmica

A terceira solução – e na maioria das vezes a mais simples – para as importações circulares é usar um comando import dentro de uma função ou método. Isso é chamado de *importação dinâmica* porque a importação do módulo acontece durante a execução do programa, e não nos momentos iniciais em está iniciando e os módulos estão sendo inicializados.

No exemplo a seguir, redefinimos o módulo dialog para usar importação dinâmica. A função dialog.show importa o módulo app durante a execução em vez de o módulo dialog importar app durante sua inicialização.

```
# dialog.py
class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    import app  # Importação dinâmica
    save_dialog.save_dir = app.prefs.get('save_dir')
    # ...
```

O módulo app pode agora ser o mesmo do exemplo original. Ele importa dialog no topo e chama dialog.show no fim do arquivo.

```
# app.py
import dialog

class Prefs(object):
    # ...
prefs = Prefs()
dialog.show()
```

O efeito disso é semelhante ao da técnica importar, configurar e executar mostrada antes. A diferença está em não precisar de alterações estruturais na maneira com que os módulos são definidos e importados. Estamos, simplesmente, postergando a importação circular até o momento em que o acesso ao outro módulo seja estritamente necessário. Nesse ponto, temos certeza de que todos os outros módulos já foram inicializados (passo 5 foi finalizado para todo mundo).

No geral, recomenda-se evitar importações dinâmicas como essas. O custo do comando import não pode ser desprezado e é especialmente pernicioso em laços de repetição. Ao postergar a execução, as importações dinâmicas também contribuem para uma surpreendente taxa de falhas durante a execução, como exceções SyntaxError que aparecem muito tempo depois de o programa já estar em execução e uso (consulte o Item 56: “Teste absolutamente tudo com unittest” para descobrir uma maneira de evitar o problema). Entretanto, as desvantagens são normalmente menos traumáticas que a perspectiva de reestruturar completamente o programa.

Lembre-se

- As dependências circulares acontecem quando dois módulos precisam chamar um ao outro no momento da importação. Quando isso acontece, normalmente o programa trava.
- A melhor maneira de desmantelar uma dependência circular é refatorar as dependências mútuas em um módulo separado e colocar esse módulo no final da árvore de dependências.
- As importações dinâmicas são a solução mais simples para resolver uma

dependência circular entre módulos sem que seja preciso refatorar profundamente o código ou torná-lo mais complexo.

Item 53: Use ambientes virtuais para criar dependências isoladas e reprodutíveis

Construir programas cada vez maiores e mais complexos fatalmente leva a empregar inúmeros pacotes da comunidade do Python (consulte o Item 48: “Saiba onde encontrar os módulos desenvolvidos pela comunidade”). Estaremos cada vez mais usando pip para instalar pacotes como pytz, numpy e muitos outros.

O problema é que, por default, pip instala os novos pacotes de forma global. Dessa forma, todos os programas em Python no sistema são afetados por esses módulos. Em tese, não deveria ser um problema. Se um pacote for instalado e nunca for importado (com o comando import), como isso afetaria os programas?

O problema advém das dependências transitivas: os pacotes que são dependências desses que instalamos manualmente. Por exemplo, podemos ver que de quais pacotes o Sphinx depende, basta perguntar ao pip (desde que o Sphinx já esteja instalado):

```
$ pip3 show Sphinx
---
Name: Sphinx
Version: 1.2.2
Location: /usr/local/lib/python3.4/site-packages
Requires: docutils, Jinja2, Pygments
```

Se instalarmos outro pacote como o flask, podemos ver que ele também depende de Jinja2:

```
$ pip3 show flask
---
Name: Flask
Version: 0.10.1
Location: /usr/local/lib/python3.4/site-packages
Requires: Werkzeug, Jinja2, itsdangerous
```

O conflito surge com o tempo porque, à medida que evoluem, Sphinx e flask divergem. É possível que, no momento, ambos requeiram a mesma versão de Jinja2 e tudo corra bem. Contudo, daqui a seis meses o Jinja2 pode lançar uma nova versão que implemente mudanças catastróficas aos programas que usam essa biblioteca. Se atualizarmos a versão global de Jinja2 com o comando pip install --upgrade, podemos descobrir estarrecidos que Sphinx deixou de funcionar, enquanto flask roda sem problemas.

A causa dessa quebradeira é que o Python só pode ter uma única versão global de cada módulo. Se um dos pacotes instalados precisar da versão nova e outro pacote precisar da antiga, o sistema não funcionará corretamente.

Esse problema aparece até mesmo quando os mantenedores do pacote dão seu melhor para preservar a compatibilidade de API entre os lançamentos (consulte o Item 50: “Use pacotes para organizar módulos e criar APIs estáveis”), novas versões de uma biblioteca podem subitamente mudar de comportamento, e o código externo que consome a API às vezes depende desse comportamento. Os usuários em um sistema podem manualmente atualizar um pacote mas não os demais. Há sempre o risco constante do chão se abrir sobre seus pés.

Essas dificuldades são multiplicadas quando colaboramos com outros desenvolvedores que fazem seu trabalho em computadores separados. É razoável imaginar que não só as versões dos pacotes, mas até mesmo a versão do Python, sejam ligeiramente diferentes de um desenvolvedor para outro, provocando situações altamente frustrantes nas quais a base de código funciona perfeitamente na máquina de um dos programadores, mas falha espetacularmente na de outro.

A solução para todos esses problemas é uma ferramenta chamada pyvenv, que cria *ambientes virtuais*. Desde o Python 3.4, a ferramenta pyvenv está disponível por default na instalação do Python (também acessível via python -m venv). Versões anteriores do Python requerem instalação de um pacote à parte (pip install virtualenv) e o nome do comando é diferente, virtualenv.

O pyvenv permite criar ambientes Python isolados. Usando pyvenv, podemos ter muitas versões diferentes do mesmo pacote instalado no mesmo sistema ao mesmo tempo sem qualquer conflito. Isso permite trabalhar em muitos projetos simultaneamente e usar muitas ferramentas diferentes no mesmo computador.

O pyvenv instala versões explícitas dos pacotes e suas dependências em

estruturas de diretório distintas e isoladas. Isso torna possível reproduzir um dado ambiente Python que funciona corretamente com o código sendo desenvolvido. É a maneira mais confiável de evitar problemas inesperados.

Comando pyvenv

O exemplo a seguir mostra um tutorial rápido de como usar o pyvenv de forma eficiente. Antes de usar a ferramenta, é importante observar o ambiente indicado pelo comando python3 em seu sistema. Em meu caso, o python3 está localizado na pasta /usr/local/bin e informa a versão 3.4.2 (consulte o Item 1: “Saiba qual versão de Python está em uso”).

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.4.2
```

Para demonstrar a configuração de meu ambiente, façamos o seguinte teste: a importação do módulo pytz não deve causar erros. Em meu caso funciona porque já tenho o pacote pytz instalado como módulo global.

```
$ python3 -c 'import pytz'
$
```

Agora, podemos chamar pyvenv para criar um novo ambiente virtual chamado myproject. Cada ambiente virtual precisa viver em seu próprio diretório único. O resultado do comando é uma árvore de pastas e arquivos.

```
$ pyvenv /tmp/myproject
$ cd /tmp/myproject
$ ls
bin    include    lib    pyvenv.cfg
```

Para começar a usar o ambiente virtual, é preciso chamar o comando source do shell de meu sistema operacional[4] para carregar o conteúdo do script bin/activate. activate ajusta todas as minhas variáveis de ambiente ao ambiente virtual, e também atualiza o prompt da linha de comando para incluir o nome do ambiente virtual ('myproject' no meu caso) para deixar absolutamente claro o ambiente em que estamos operando agora.

```
$ source bin/activate
```

```
(myproject)$
```

Depois da ativação, podemos observar que o caminho do comando python3, que chama o interpretador Python, foi transferido para dentro da estrutura de pastas do ambiente virtual.

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /tmp/myproject/bin/python3.4
(myproject)$ ls -l /tmp/myproject/bin/python3.4
... -> /usr/local/bin/python3.4
```

Com isso, garantimos que mudanças no sistema externo não afetarão o ambiente virtual. Mesmo se o sistema externo atualizar a versão do python3 para 3.5, meu ambiente virtual ainda apontará explicitamente para a versão 3.4.

O ambiente virtual que criamos com o pyvenv não tem, de início, nenhum pacote instalado à exceção de pip e setuptools. Se tentarmos usar o pacote pytz que foi instalado como módulo global no sistema externo seremos agraciados com um erro, porque esse pacote é desconhecido para o ambiente virtual.

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'pytz'
```

Podemos usar o pip para instalar o módulo pytz exclusivo do ambiente virtual.

```
(myproject)$ pip3 install pytz
```

Uma vez instalado, podemos verificar que tudo funciona com o mesmo teste do comando de importação.

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

Para encerrar a atividade no ambiente virtual e voltar ao sistema padrão, use o comando deactivate, que restaura seu ambiente-padrão de usuário, incluindo o local em que o comando python3 está.

```
(myproject)$ deactivate
$ which python3
```

```
/usr/local/bin/python3
```

Caso queira trabalhar novamente com o ambiente myproject, basta executar source bin/activate no diretório, como fizemos antes.

Reproduzindo dependências

Agora que já temos o ambiente virtual, podemos continuar instalando pacotes com pip à medida que forem necessários. Algum dia, depararemos com a necessidade de copiar o ambiente para outra máquina. Por exemplo, digamos que se queira reproduzir o ambiente de desenvolvimento no servidor de produção. Ou, talvez, clonar o ambiente de outra pessoa em sua própria máquina para poder rodar seu código.

O pyvenv facilita o trabalho nessas situações. Podemos usar o comando pip freeze para salvar todas as dependências explícitas de pacote em um arquivo. Por convenção, esse arquivo deve se chamar requirements.txt.

```
(myproject)$ pip3 freeze > requirements.txt
(myproject)$ cat requirements.txt
numpy==1.8.2
pytz==2014.4
requests==2.3.0
```

Agora, imagine que precisamos de outro ambiente idêntico ao de myproject. Para isso, basta criar uma pasta como antes usando o pyvenv e ativar o ambiente com activate.

```
$ pyvenv /tmp/otherproject
$ cd /tmp/otherproject
$ source bin/activate
(otherproject)$
```

O novo ambiente não terá nenhum pacote instalado.

```
(otherproject)$ pip3 list
pip (1.5.6)
setuptools (2.1)
```

Para instalar todos os pacotes do primeiro ambiente, basta executar pip install sobre o arquivo requirements.txt que foi gerado com o comando pip freeze.

```
(otherproject)$ pip3 install -r /tmp/myproject/requirements.txt
```

As engrenagens do Python vão remoer por um certo tempo enquanto baixam e instalam os pacotes necessários para reproduzir o primeiro ambiente. Uma vez terminado, se listarmos o conjunto de pacotes do segundo ambiente veremos que a lista de dependências será idêntica à do primeiro.

```
(otherproject)$ pip list
numpy (1.8.2)
pip (1.5.6)
pytz (2014.4)
requests (2.3.0)
setuptools (2.1)
```

Empregar um arquivo requirements.txt é o ideal quando se pensa em colaborar com outros programadores por meio de um sistema de controle de versões. Os commits de código ocorrem simultaneamente à atualização da lista de dependências de pacotes, garantindo que tudo esteja sempre atualizado a cada passo.

A pegadinha com os ambientes virtuais é que transferi-los faz com que seu código deixe de funcionar por conta dos caminhos absolutos. Assim como vimos com o python3, todos eles são fixos e imutáveis em relação à pasta de instalação. Porém, isso não importa muito. A finalidade dos ambientes virtuais é facilitar a reprodução de uma mesma configuração. Em vez de mover a pasta do ambiente virtual, simplesmente congele o anterior (com freeze), crie um outro onde desejar e reinstale tudo a partir do arquivo requirements.txt.

Lembre-se

- Os ambientes virtuais permitem usar pip para instalar inúmeras versões diferentes do mesmo pacote na mesma máquina sem que haja conflitos.
- Os ambientes virtuais são criados com o comando pyvenv, ativados com source bin/activate e desativados com deactivate.
- É possível empacotar todas as dependências de um ambiente com pip freeze e reproduzir mais tarde o mesmo ambiente fornecendo o arquivo requirements.txt para o comando pip install -r.
- Nas versões do Python anteriores a 3.4, a ferramenta pyvenv deve ser baixada

e instalada separadamente. O comando no shell é virtualenv em vez de pyvenv.

---

1 N. do T.: O IPython Notebook será em breve desmembrado do projeto IPython e passará a fazer parte de outro projeto maior chamado Jupyter, que oferece as mesmas ferramentas do IPython (e outras mais) para um número maior de linguagens. Até o momento, são mais de 40 suportadas. Mais informações nos sites do IPython (*ipython.org*) e do Jupyter (*jupyter.org*).

2 N. do T.: Como já dissemos em outros capítulos, as docstrings (assim como os nomes de variáveis, métodos, classes e funções) devem necessariamente ser escritas em inglês. A probabilidade de seu código acabar nas mãos de alguém que não fale seu idioma natural é grande, especialmente se o programa for de código aberto – mas não apenas nessa situação.

3 N. do T.: No jargão do desenvolvimento de software, downstream refere-se aos programas que usarão meu código para criar programas derivados, enquanto upstream é algum código de outra pessoa ou organização que eu uso para fazer o meu programa derivado. Por exemplo, no desenvolvimento de distribuições Linux, a distribuição Ubuntu considera o Debian Linux como upstream, porque o Ubuntu é (ou era) um derivado do Debian. Por sua vez, o Debian considera o Ubuntu como downstream. Da mesma forma, o kernel do Linux é upstream para o Debian, e o Debian é downstream para o Linux. Esses termos não têm tradução para o português.

4 N. do T.: Atenção usuários do Windows: o comando source é uma ferramenta do Windows, não do Python. O sistema operacional do autor é o Mac OS X, que usa um shell POSIX como a maioria dos sistemas derivados do Unix ou Unix-like: todos os Linux, todos os BSDs, Solaris, HP-UX, AIX etc. Apesar das diferenças, todos eles têm um comando source ou substituto que permitem fazer o que o autor está descrevendo. Entretanto, o Windows não é descendente do Unix, portanto não respeita a estrutura de diretórios e não usa um Shell no padrão POSIX. Para saber como lidar com ambientes virtuais de Python no Windows, bem como entender as limitações de portabilidade entre essa plataforma e as demais, consulte a documentação oficial do Python em *https://docs.python.org/3/library/venv.html*.

CAPÍTULO 8

Produção

Colocar em uso um programa em Python implica transferi-lo do ambiente de desenvolvimento para o ambiente de produção. Dar suporte a configurações disparatadas como essa pode ser um desafio e tanto. Criar programas confiáveis em qualquer situação é tão ou mais importante que implementar neles a funcionalidade correta.

O objetivo do programador deve ser sempre *deixar os programas em Python prontos para produção* e torná-los à prova de balas enquanto estiverem em uso. O Python tem módulos nativos para aprimorar a robustez do código, com recursos para depuração, otimização e testes que maximizam a qualidade e o desempenho de nossos programas na hora da verdade – ou seja, nas mãos do cliente ou usuário final.

Item 54: Crie código com escopo no módulo para configurar os ambientes de implementação

Um ambiente de implementação (ou, no jargão comumente usado, o ambiente de deploy, ou deployment) é a configuração na qual os programas serão executados. Todo e qualquer programa tem pelo menos um deployment, o *ambiente de produção*. O objetivo original do programador, quando começou a escrever o programa, era, de fato, usá-lo em um ambiente de produção e obter algum resultado com ele.

Escrever ou modificar um programa requer a capacidade de executá-lo no computador usado para o desenvolvimento. A configuração do *ambiente desenvolvimento* pode ser bastante diferente da do ambiente de produção. Por exemplo, podemos estar escrevendo em um simples PC com Linux um programa que será executado em supercomputadores.

Ferramentas como o pyvenv (consulte o Item 53: “Use ambientes virtuais para criar dependências isoladas e reprodutíveis”) facilitam garantir que todos os

ambientes tenham sempre os mesmos pacotes do Python instalados. O problema é que os ambientes de produção às vezes têm muitos pré-requisitos externos, que são difíceis de reproduzir em um ambiente de desenvolvimento.

Por exemplo, digamos que se queira executar o programa em um contêiner de servidor web e dar a ele acesso ao banco de dados. Isso significa que toda vez que quisermos modificar o código do programa é preciso rodar, na máquina de desenvolvimento, o contêiner do servidor web, o banco de dados deve estar configurado corretamente e seu programa precisa das senhas de acesso. É uma quantidade considerável de trabalho e preocupações, especialmente quando tudo o que estamos tentando fazer é testar se a pequena modificação que fizemos em uma única linha funciona corretamente.

A melhor maneira de contornar esses problemas é sobrepor partes do programa durante sua inicialização para oferecer alguma funcionalidade diferente, dependendo do ambiente em que está sendo executado. Por exemplo, poderíamos ter dois arquivos __main__ diferentes, um para produção e outro para desenvolvimento.

```
# dev_main.py
TESTING = True
import db_connection
db = db_connection.Database()
```

```
# prod_main.py
TESTING = False
import db_connection
db = db_connection.Database()
```

A única diferença entre os dois arquivos é o valor da constante TESTING. Os outros módulos do programa podem então importar o módulo __main__ e usar o valor de TESTING para decidir por qual deles definir seus atributos.

```
# db_connection.py
import __main__

class TestingDatabase(object):
    # ...
```

```
class RealDatabase(object):
    # ...

if __main__.TESTING:
    Database = TestingDatabase
else:
    Database = RealDatabase
```

O comportamento-chave a ser observado aqui é que o código rodando no escopo do módulo – e não dentro de nenhuma função ou método – é código normal em Python. Podemos usar um comando if no nível do módulo para decidir como o módulo definirá os nomes, permitindo que os módulos sejam feitos sob medida para os vários ambientes de deploy. Assim, evitamos ter de reproduzir itens custosos, como, por exemplo, as configurações de banco de dados, quando não são necessários. Podemos injetar implementações falsas ou forjadas que facilitam o desenvolvimento e os testes interativos (consulte o Item 56: “Teste absolutamente tudo com unittest”).

Nota

Em algum momento seus ambientes de desenvolvimento ficarão complicados. Quando isso acontecer, você deve considerar removê-los de constantes no módulo principal (como o `TESTING` do exemplo anterior) e transferi-los para arquivos externos de configuração. Ferramentas como o módulo nativo `configparser` permite manter as configurações de produção separadas do código, uma distinção crucial quando colaboramos com as equipes de operação.

Essa técnica pode ser usada para outros casos em que seja necessário contornar condições externas. Por exemplo, caso saibamos que o programa deve funcionar de forma diferente, dependendo da plataforma, podemos inspecionar o módulo sys antes de definir estruturas de alto nível no módulo.

```
# db_connection.py
import sys

class Win32Database(object):
    # ...
```

```
class PosixDatabase(object):
    # ...

if sys.platform.startswith('win32'):
    Database = Win32Database
else:
    Database = PosixDatabase
```

Da mesma forma, podemos usar variáveis de ambiente de os.environ para guiar as definições de nosso módulo.

Lembre-se

- Os programas normalmente precisam rodar em múltiplos ambientes de deploy que pedem, cada um, requisitos e configurações únicos.
- Podemos modificar os módulos para que usem conteúdos diferentes em diferentes ambientes de deploy, escolhendo quando usar um ou outro com comandos normais do Python no escopo do módulo.
- O conteúdo dos módulos pode ser o produto de qualquer condição externa, incluindo introspecção pelos módulos sys e os.

Item 55: Use strings com a função repr para depuração

Ao depurar um programa em Python, a função print[1] (ou a saída por meio do módulo nativo logging) nos leva longe, para nossa surpresa. As entranhas do Python são, na maioria das vezes, fáceis de acessar por atributos simples (consulte o Item 27: “Prefira atributos públicos em vez de privativos”). Tudo o que precisamos fazer é mostrar na tela (com print) as mudanças no estado do programa durante sua execução, e prestar atenção caso algo dê errado.

A função print mostra na tela uma versão em string, legível para nós, humanos, de qualquer coisa que forneçamos a ela. Por exemplo, usar print com uma string básica mostrará na tela o conteúdo da string. O comando até mesmo toma o cuidado de retirar as aspas.

```
print('foo bar')
```

```
>>>
foo bar
```

Isso é equivalente a usar o formatador '%s' e o operador %.

```
print('%s' % 'foo bar')
>>>
foo bar
```

O problema é que a string humanamente legível não deixa claro qual é o tipo em que o valor foi definido. Por exemplo, observe como a saída default de print não consegue distinguir entre o tipo número do valor 5 e o tipo string do valor '5'.

```
print(5)
print('5')
>>>
5
5
```

Se estivermos depurando um programa com print, é importante identificar esses tipos diferentes. Na maioria esmagadora das vezes, o que queremos ver enquanto depuramos é a versão repr do objeto. A função nativa repr devolve a *representação exibíveis* de um objeto, que deve ser sua representação em string mais claramente distinguível. Para tipos nativos, a string devolvida por repr é uma expressão válida em Python.

```
a = '\x07'
print(repr(a))
>>>
'\x07'
```

Passar o valor de repr para a função nativa eval deve resultar no mesmo objeto Python com que começamos (mas tenha em mente que, na prática, devemos usar o eval apenas onde for estritamente necessário e com extrema cautela).

```
b = eval(repr(a))
assert a == b
```

Quando estamos depurando com print, devemos usar repr no valor antes de mostrá-lo na tela para garantir que quaisquer diferenças entre tipos fiquem muito claras.

```
print(repr(5))
print(repr('5'))
>>>
5
'5'
```

Isso é equivalente a usar o formatador '%r' e o operador %.

```
print('%r' % 5)
print('%r' % '5')
>>>
5
'5'
```

Para objetos dinâmicos em Python, a string humanamente legível default é o mesmo valor que o mostrado por repr. Isso significa que passar um objeto dinâmico diretamente ao print será o mesmo que repassá-lo explicitamente via repr. Infelizmente, o valor default de repr para instâncias de objeto não é lá muito útil. No exemplo a seguir, definimos uma classe muito simples e mostramos seu valor na tela:

```
class OpaqueClass(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = OpaqueClass(1, 2)
print(obj)
>>>
<__main__.OpaqueClass object at 0x107880ba8>
```

Essa saída não pode ser passada à função eval, e não diz nada sobre os campos de interface do objeto.

Temos duas soluções para o problema. Se a classe está sob nosso controle, podemos definir nosso próprio método especial __repr__ que devolve strings contendo a expressão em Python que recria o objeto. No exemplo a seguir, definimos essa função para a classe acima:

```
class BetterClass(object):
    def __init__(self, x, y):
        # ...
    def __repr__(self):
        return 'BetterClass(%d, %d)' % (self.x, self.y)
```

Agora, o valor de repr é muito mais útil.

```
obj = BetterClass(1, 2)
print(obj)
>>>
BetterClass(1, 2)
```

Quando não temos controle sobre a definição de classe, podemos nos valer do dicionário da instância do objeto, que está armazenado no atributo __dict__. No exemplo a seguir, mostramos na tela o conteúdo de uma instância da classe "opaca" (isto é, cujo interior não conseguimos enxergar) chamada OpaqueClass:

```
obj = OpaqueClass(4, 5)
print(obj.__dict__)
>>>
{'y': 5, 'x': 4}
```

Lembre-se

- Chamar print em tipos nativos do Python produzirá strings legíveis por nós, meros humanos, dos valores armazenados nesses tipos. O valor é mostrado, mas as informações sobre o tipo não.
- Podemos usar repr em tipos nativos do Python para obter uma versão em string do valor. Essas strings do repr podem ser passadas à função nativa eval para obter o valor original.
- Os formatadores %s produzem strings legíveis para os humanos como str. Já o formatador %r produz strings exibíveis como repr.
- Podemos definir o método __repr__ para customizar a representação exibível de uma classe e oferecer informações de depuração mais detalhadas.
- Sempre é possível consultar o atributo _dict__ de um objeto para conhecer suas entranhas.

Item 56: Teste absolutamente tudo com unittest

O Python não tem verificação de tipos estáticos. Não há nada no compilador que garanta o funcionamento de seu programa quando for executado. Com o Python não sabemos se as funções que o programa chama serão definidas em tempo de execução, mesmo que sua existência seja evidente no código-fonte. Esse comportamento dinâmico é uma bênção e uma maldição.

O número astronômico de programadores de Python no mundo é justificado pela produtividade que se ganha por sua concisão e simplicidade. Contudo, não é pequeno o número de pessoas que já ouviram pelo menos uma história de choro e ranger de dentes sobre o Python no qual um programa encontrou um erro cretino durante a execução.

Um dos exemplos mais escabrosos que já encontrei foi quando uma exceção de erro de sintaxe (SyntaxError) foi levantada em produção como efeito colateral de uma importação dinâmica (consulte o Item 52: “Saiba como romper dependências circulares”). Um programador meu conhecido que presenciou (e teve que corrigir) essa ocorrência surpreendente jurou por todos os nomes sagrados jamais usar Python novamente.

Eu, de minha parte, fiquei matutando: por que o código não foi testado antes de o programa ser colocado em produção? Segurança de tipos não é tudo. É obrigatório sempre testar o código, independente de qual linguagem ele é escrito. Todavia, admito que a grande diferença entre o Python e muitas outras linguagens é que *a única maneira* de confiar em um programa em Python é escrevendo testes. Não há nenhum véu de verificação de tipos estáticos para que nos sintamos seguros.

Felizmente, os mesmos recursos dinâmicos que impedem a verificação de tipos no Python também facilitam grandemente escrever testes para verificar o código. Podemos usar a natureza dinâmica do Python e seus comportamentos facilmente substituíveis para implementar testes e garantir que os programas funcionem como esperado.

Devemos pensar nos testes como uma apólice de seguros para o código. Bons testes dão confiança de que o código está correto. Se refatorarmos ou expandirmos o código, os testes facilitam a identificação de como os comportamentos mudaram. Parece conversa fiada, mas não é: bons testes na verdade facilitam modificar o código mais tarde, em vez de dificultar.

A maneira mais simples de escrever testes é usar o módulo nativo unittest. Por exemplo, digamos que exista uma função utilitária definida no arquivo utils.py:

```
# utils.py
def to_str(data):
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Must supply str or bytes, '
                        'found: %r' % data)
```

Para definir os testes, crio um segundo arquivo chamado test_utils.py ou utils_test.py, que contém testes para cada comportamento esperado.

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))

    def test_to_str_str(self):
        self.assertEqual('hello', to_str('hello'))

    def test_to_str_bad(self):
        self.assertRaises(TypeError, to_str, object())

if __name__ == '__main__':
    main()
```

Os testes são organizados em classes TestCase. Cada teste é um método começando com a palavra test. Se um método de teste roda sem levantar nenhum tipo de exceção (incluindo o AssertionError dos comandos assert), considera-se que o teste teve sucesso, ou seja, passou sem encontrar erros.

A classe TestCase oferece métodos auxiliares para definir assertivas nos testes, como, por exemplo, assertEqual, que verifica uma condição de igualdade, assertTrue, que verifica expressões booleanas e assertRaises, que verifica se alguma exceção que deveria ser elevada realmente o foi (consulte help(TestCase) para mais informações). Podemos definir nossos próprios métodos auxiliares nas subclasses de TestCase para garantir que os testes sejam mais legíveis – assegure-se apenas de que os nomes de seus métodos não iniciem com a palavra test.

Nota

Outra prática comum ao escrever testes é usar funções e classes falsas para provocar certos comportamentos. Para esse fim, o Python 3 oferece o módulo nativo unittest.mock, também disponível no Python 2 como um pacote externo open source.

Às vezes, nossas classes TestCase precisam configurar o ambiente de testes antes de rodar os métodos de teste. Para isso, podemos sobrepor com substitutos os métodos setUp e tearDown. Esses métodos são chamados antes e depois de cada método de testes para garantir que cada um dos testes rode em um ambiente isolado (uma prática importante para criar testes como se deve). No exemplo a seguir, definimos um TestCase que cria uma pasta temporária antes de cada teste e a apaga depois que cada teste termina:

```
class MyTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
    def tearDown(self):
        self.test_dir.cleanup()
    # Os métodos de teste começam aqui
    # ...
```

Normalmente, agrupamos os testes que têm relação entre si em conjuntos e definimos um TestCase para cada conjunto. Às vezes temos um TestCase para cada função que tem muitos casos limítrofes. Outras vezes, um só TestCase contém testes para todas as funções em um módulo. Também criamos um TestCase para testar uma única classe e todos os seus métodos.

Quando os programas começam a ficar complicados, provavelmente serão necessários testes adicionais para verificar as interações entre os módulos, pois

até aqui o código era sempre testado em condição de isolamento. Essa é a diferença entre *unit tests* (testes de unidade) e *integration tests* (testes de integração). No Python, é importante escrever ambos os tipos de testes pela mesmíssima razão: não temos garantia de que os módulos conseguirão trabalhar juntos a não ser que tenhamos prova disso.

Nota

Dependendo do projeto, também pode ser útil definir testes baseados em dados (data-driven tests) ou organizar testes em diferentes conjuntos de funcionalidades relacionadas. Para esse fim, os relatórios de cobertura de código (code coverage reports) e outras ferramentas avançadas como os pacotes externos nose (*http://nose.readthedocs.org/*) e o pytest (*http://pytest.org/*), ambos de código aberto, podem ser especialmente úteis.

Lembre-se

- A única maneira de se ter confiança num programa em Python é escrevendo testes.
- O módulo nativo unittest fornece a maioria dos recursos de que precisamos para escrever bons testes.
- Podemos definir os testes construindo subclasses de TestCase e definindo um método por comportamento que gostaríamos de testar. Os métodos de teste nas classes TestCase devem obrigatoriamente começar com a palavra test.
- É importante escrever tanto testes de unidade (unit tests), para testar funcionalidades isoladamente, como testes de integração (integration tests), para testar a interação entre módulos quando existir.

Item 57: Prefira usar depuradores interativos como o pdb

Quem disser que nunca encontrou um bug em seu próprio código é mentiroso. Encontrá-los é preciso, e a função print pode ajudar bastante a localizar sua origem (consulte o Item 55: “Use strings com a função repr para depuração”). Escrever testes para casos específicos que causam problema é outra grande maneira de isolar problemas (consulte o Item 56: “Teste absolutamente tudo com unittest”).

Porém, essas ferramentas não são suficientes para encontrar todo e qualquer bug. Quando for preciso um remédio mais potente, é hora de tentar o *interactive debugger*, um depurador interativo do Python. Esse depurador permite inspecionar os estados do programa, mostrar variáveis locais e varrer passo a passo um programa em Python, uma instrução por vez.

Em muitas linguagens de programação usamos o depurador para especificar em qual linha de um arquivo-fonte deve parar, e executar o programa a partir daí (os chamados breakpoints). No Python, é mais fácil modificar diretamente nosso programa para iniciar o depurador imediatamente antes de onde achamos que haja um problema que mereça ser investigado. Não há diferença entre rodar um programa em Python sob um depurador e rodá-lo normalmente.

Para iniciar o depurador, tudo o que precisamos fazer é importar o módulo nativo pdb e executar sua função set_trace. Normalmente isso é feito em uma única linha para que os programadores possam facilmente desabilitá-la com um comentário, ou seja, iniciando a linha com o caractere #.

```
def complex_func(a, b, c):
    # ...
    import pdb; pdb.set_trace()
```

No momento em que essa instrução é executada, o programa entra em pausa. O terminal que iniciou o programa transforma-se em um shell interativo do Python.

```
-> import pdb; pdb.set_trace()
(Pdb)
```

No prompt (Pdb) podemos digitar o nome de uma variável local para ver seu valor. Podemos ver uma lista de todas as variáveis locais chamando a função locals. Podemos importar módulos, inspecionar o estado global do programa, construir novos objetos, chamar a função nativa help e mesmo modificar partes do programa – o que quer que seja necessário para ajudar na depuração. Além disso, o depurador tem três comandos que facilitam a inspeção:

- bt: Mostra na tela o histórico (backtrace ou traceback) atual da pilha de execução. Isso permite saber em que ponto estamos do programa e como chegamos até o ponto de disparo pdb.set_trace.
- up: Move o escopo um nível acima na pilha de chamadas de função, ou seja, para nível do chamador da função atual. Isso permite inspecionar as variáveis

locais em níveis mais altos da pilha de chamadas.

- down: Move o escopo um nível abaixo na pilha de chamadas de função.

Uma vez inspecionado o estado atual, podemos usar o depurador para continuar a execução do programa sob nosso rigoroso controle.

- step: Executa apenas a próxima linha do programa e devolve o controle ao depurador. Se a próxima linha incluir a chamada de uma função, o depurador entrará em pausa logo antes da próxima instrução interna dessa função.
- next: Executa apenas a próxima linha do programa e devolve o controle ao depurador. Contudo, se a próxima linha incluir a chamada de uma função, o depurador irá executar a função por completo e parar imediatamente depois de a função devolver seu valor de retorno.
- return: Executa o programa até que a função atual devolva seu valor de retorno e depois devolve o controle para o depurador.
- continue: Continua executando o programa até o próximo breakpoint (ou até que set_trace seja novamente chamado).

Lembre-se

- Podemos iniciar o depurador interativo do Python (pdb) em um determinado ponto de interesse do programa, bastando inserir os comandos import pdb; pdb.set_trace() nos locais apropriados.
- O prompt do pdb é um shell completo do Python que permite inspecionar e modificar o estado de um programa em execução.
- Os comandos do shell do pdb permitem controlar com precisão o programa em execução, permitindo alternar entre inspecionar o estado do programa e avançar passo a passo em sua execução.

Item 58: Meça os perfis de desempenho antes de otimizar o código

A natureza dinâmica do Python causa comportamentos surpreendentes em seu desempenho durante a execução. Operações que pensaríamos ser lentas são, na verdade, muito rápidas (manipulação de strings, geradores). Recursos da linguagem que esperaríamos ser mais rápidos são, para nossa surpresa,

extremamente lentos (acesso a atributos, chamadas a funções). A origem desses gargalos em um programa em Python pode ser um tanto obscura.

A melhor técnica aqui é ignorar sua intuição e medir diretamente o desempenho de um programa antes de tentar otimizá-lo. O Python oferece um recurso nativo chamado de *profiler* (traçador de perfil) para determinar que partes de um programa são responsáveis por sua execução. Isso permite que concentremos os esforços de otimização nas maiores fontes de problemas e ignoremos as partes do programa que não causam impacto na velocidade de execução.

Por exemplo, digamos que se queira determinar por que um algoritmo em seu programa é lento. No exemplo a seguir, definimos uma função que classifique a uma lista de dados usando um algoritmo de ordenação por inserção:

```
def insertion_sort(data):
    result = []
    for value in data:
        insert_value(result, value)
    return result
```

O mecanismo central da ordenação por inserção é a função que encontra o ponto de inserção de cada item dos dados. No exemplo a seguir, definimos uma versão extremamente ineficiente da função insert_value, que faz uma varredura linear sobre um array de entrada:

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
            return
    array.append(value)
```

Para traçar o perfil de insertion_sort e insert_value, criamos um conjunto de dados de números aleatórios e definimos uma função chamada test, que passamos ao profiler.

```
from random import randint

max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
```

```
test = lambda: insertion_sort(data)
```

O Python tem dois profilers nativos, um escrito puramente em Python (profile) e o outro que é um módulo em C (cProfile). O módulo nativo cProfile é melhor porque causa um impacto muito ínfimo no desempenho de seu programa enquanto está sendo analisado. A alternativa em Python puro coloca um fardo tão grande no desempenho que pode até mesmo influenciar os resultados.

Nota

Ao traçar o perfil de um programa em Python, certifique-se de que está medindo o desempenho apenas do código e não de sistemas externos. Esteja atento para funções que acessem a rede ou recursos no disco. Tais funções parecem ter grande impacto no tempo de execução do programa, mas quem é lento de fato são os sistemas subjacentes. Se seu programa usa cache para mascarar a latência de recursos vagarosos como esses, é também prudente deixar o cache “encher” por alguns momentos antes de iniciar o processo de determinação dos perfis de desempenho.

No exemplo a seguir, instanciamos um objeto Profile do módulo cProfile e rodamos nele a função de teste usando o método runcall:

```
profiler = Profile()
profiler.runcall(test)
```

Uma vez que o teste tenha terminado, podemos extrair dados estatísticos sobre seu desempenho usando o módulo nativo pstats, mais precisamente sua classe Stats. Os muitos métodos de um objeto Stats permitem ajustar como selecionar e ordenar as informações dos perfis coletados para mostrar apenas as que nos interessam.

```
stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

A saída é uma tabela de informações organizada por função. A amostra de dados é extraída somente durante o tempo de atividade do profiler, usando o método runcall citado anteriormente.

```
>>>
        20003 function calls in 1.812 seconds
```

```
   Ordered by: cumulative time

 ncalls tottime percall cumtime percall filename:lineno(function)
      1   0.000   0.000   1.812   1.812 main.py:34(<lambda>)
      1   0.003   0.003   1.812   1.812 main.py:10(insertion_sort)
  10000   1.797   0.000   1.810   0.000 main.py:20(insert_value)
   9992   0.013   0.000   0.013   0.000 {method 'insert' of 'list' objects}
      8   0.000   0.000   0.000   0.000 {method 'append' of 'list' objects}
      1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Cada coluna das estatísticas de perfil tem um significado importante:

- ncalls: número de chamadas à função durante a coleta de perfis.
- tottime: tempo, em segundos, gasto executando a função, excluindo o tempo de execução de outras funções chamadas por esta.
- tottime percall: tempo médio, em segundos, gasto na função a cada chamada, excluindo o tempo de execução de outras funções chamadas por esta. Estritamente, é tottime dividido por ncalls.
- cumtime: tempo acumulado, em segundos, gasto executando a função, incluindo o tempo gasto com chamadas a outras funções e sua execução.
- cumtime percall: tempo médio, em segundos, gasto na função a cada chamada, incluindo o tempo gasto com chamadas a outras funções e sua execução. Estritamente, é cumtime dividido por ncalls.

Na tabela de estatísticas mostrada, é evidente o excesso de uso de CPU pela função insert_value. O exemplo a seguir redefine essa função para usar o módulo nativo bisect (consulte o Item 46: “Use algoritmos e estruturas de dados nativos”):

```
from bisect import bisect_left


def insert_value(array, value):
    i = bisect_left(array, value)
    array.insert(i, value)
```

Podemos rodar o profiler novamente e gerar uma nova tabela de estatísticas de perfil de desempenho. A nova função será muito mais rápida, com um tempo cumulativo quase 100× menor que o da função insert_value anterior.

```
>>>
         30003 function calls in 0.028 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
        1   0.000   0.000   0.028   0.028 main.py:34(<lambda>)
        1   0.002   0.002   0.028   0.028 main.py:10(insertion_sort)
    10000   0.005   0.000   0.026   0.000 main.py:112(insert_value)
    10000   0.014   0.000   0.014   0.000 {method 'insert' of 'list' objects}
    10000   0.007   0.000   0.007   0.000 {built-in method bisect_left}
        1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Muitas vezes, ao traçar o perfil de um programa completo, descobriremos que uma única função utilitária é responsável pela maioria do tempo de execução. A saída-padrão do profiler dificulta o entendimento dessa situação porque não mostra como a função utilitária é chamada pelas diferentes partes do programa.

No exemplo a seguir, a função my_utility é chamada repetidamente por duas diferentes funções do programa:

```
def my_utility(a, b):
    # ...

def first_func():
    for _ in range(1000):
        my_utility(4, 5)

def second_func():
    for _ in range(10):
        my_utility(1, 3)

def my_program():
    for _ in range(20):
        first_func()
        second_func()
```

Ao traçar o perfil de desempenho desse código usando a saída default de print_stats, teremos resultados estatísticos bastante confusos.

```
>>>
         20242 function calls in 0.208 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.208    0.208 main.py:176(my_program)
       20    0.005    0.000    0.206    0.010 main.py:168(first_func)
    20200    0.203    0.000    0.203    0.000 main.py:161(my_utility)
       20    0.000    0.000    0.002    0.000 main.py:172(second_func)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

A função my_utility é claramente a causa da maior parte do tempo de execução, mas fica imediatamente óbvio o porquê de a função ser chamada tantas vezes. Se procurarmos no código-fonte, encontraremos muitos lugares em que my_utility é chamada e isso não contribuirá em nada para diminuir nossa confusão.

Para lidar com o problema, o Python profiler oferece uma maneira de conferir quais chamadores contribuíram para a informação de perfil em cada função.

```
stats.print_callers()
```

A tabela de estatísticas de perfil de desempenho mostra, à esquerda, as funções chamadas e, à direita, quem foi responsável por fazer a chamada. Fica bastante claro que first_func é quem mais chama my_utility:

```
>>>
   Ordered by: cumulative time

Function                          was called by...
                                ncalls   tottime  cumtime
main.py:176(my_program)           <-
main.py:168(first_func)           <-      20     0.005    0.206  main.py: 176(my_program)
main.py:161(my_utility)           <-   20000     0.202    0.202  main.py: 168(first_func)
                                  200     0.002    0.002  main.py:172(second_func)
main.py:172(second_func)          <-      20     0.000    0.002  main.py: 176(my_program)
```

Lembre-se

- É importante traçar o perfil de desempenho dos programas em Python antes de otimizar o código porque a origem dos episódios de lentidão pode ser obscura.
- Use o módulo cProfile em vez do módulo profile porque as informações de

perfil de desempenho são mais precisas.

- O método runcall, presente no objeto Profile, oferece tudo o que precisamos para traçar o perfil, em prefeito isolamento, das chamadas a uma árvore de funções.
- O objeto Stats permite selecionar e imprimir o subconjunto apropriado de informações de perfil de desempenho para que possamos entender o desempenho do programa.

Item 59: Use tracemalloc para entender o uso e os vazamentos de memória

O gerenciamento de memória na implementação default do Python, o interpretador CPython, usa contagem de referência. Isso garante que, no momento em que todas as referências ao objeto deixarem de existir (expirarem), o objeto referenciado seja também apagado. O CPython também tem um detector nativo de ciclos para garantir que objetos que se autorreferenciem sejam recolhidos como lixo periodicamente.

Em tese, isso significa que a maioria dos programadores em Python não precisa se preocupar com alocação e desalocação manual em seus programas. A linguagem cuida disso automaticamente, por meio do interpretador CPython. Na prática, contudo, alguns programas esgotam sua memória por conta de referências travadas. Descobrir onde seus programas em Python estão consumindo ou mesmo vazando memória às vezes é um pesadelo.

A primeira técnica para se depurar o uso de memória é pedir ao módulo nativo gc uma lista de todos os objetos reconhecidos pelo coletor de lixo (garbage collector). Embora seja uma ferramenta um tanto brusca, esse recurso permite ter uma noção rápida de como a memória de seu programa está sendo usada.

No exemplo a seguir, o programa desperdiça memória ao manter as referências ativas todo o tempo. Uma contagem dos objetos criados durante a execução é mostrada na tela, bem como uma pequena amostra dos objetos.

```
# using_gc.py
import gc
found_objects = gc.get_objects()
print('%d objects before' % len(found_objects))
```

```
import waste_memory
x = waste_memory.run()
found_objects = gc.get_objects()
print('%d objects after' % len(found_objects))
for obj in found_objects[:3]:
    print(repr(obj)[:100])
>>>
4756 objects before
14873 objects after
<waste_memory.MyObject object at 0x1063f6940>
<waste_memory.MyObject object at 0x1063f6978>
<waste_memory.MyObject object at 0x1063f69b0>
```

O problema com gc.get_objects é que não diz nada sobre *como* os objetos foram alocados. Em programas complicados, uma classe ou objeto específicos podem ser alocados de muitas maneiras diferentes. O número total de objetos não é tão importante quanto identificar o código responsável por alocar os objetos que estão com vazamento de memória.

O Python 3.4 introduziu um novo módulo nativo chamado tracemalloc para resolver esse problema. tracemalloc possibilita relacionar um objeto com o local em que foi alocado. No exemplo a seguir, usamos tracemalloc para mostrar na tela os três maiores culpados de vazamento de memória em um programa:

```
# top_n.py
import tracemalloc
tracemalloc.start(10)  # Reserva até 10 quadros na pilha

time1 = tracemalloc.take_snapshot()
import waste_memory
x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'lineno')
for stat in stats[:3]:
```

```
    print(stat)
>>>
waste_memory.py:6: size=2235 KiB (+2235 KiB), count=29981 (+29981), average=76 B
waste_memory.py:7: size=869 KiB (+869 KiB), count=10000 (+10000), average=89 B
waste_memory.py:12: size=547 KiB (+547 KiB), count=10000 (+10000), average=56 B
```

Fica imediatamente claro quais objetos estão dominando o uso de memória em meu programa e quais trechos do código-fonte alocam esses objetos.

O módulo tracemalloc também pode mostrar o rastro da pilha (stack trace) completo para cada alocação (o limite é o número de quadros de pilha passados ao método start). No exemplo a seguir, mostramos o stack trace do responsável pelo maior uso de memória no programa:

```
# with_trace.py
# ...
stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('\n'.join(top.traceback.format()))
>>>
File "waste_memory.py", line 6
  self.x = os.urandom(100)
File "waste_memory.py", line 12
  obj = MyObject()
File "waste_memory.py", line 19
  deep_values.append(get_data())
File "with_trace.py", line 10
  x = waste_memory.run()
```

Um stack trace como esse é valioso para descobrir em que situação uma função comum é responsável pelo consumo de memória do programa.

Infelizmente, o Python 2 não oferece o módulo tracemalloc nativamente. Há pacotes open source para rastrear o uso de memória no Python 2 (como o heapy), mas esses não replicam totalmente a funcionalidade do tracemalloc.

Lembre-se

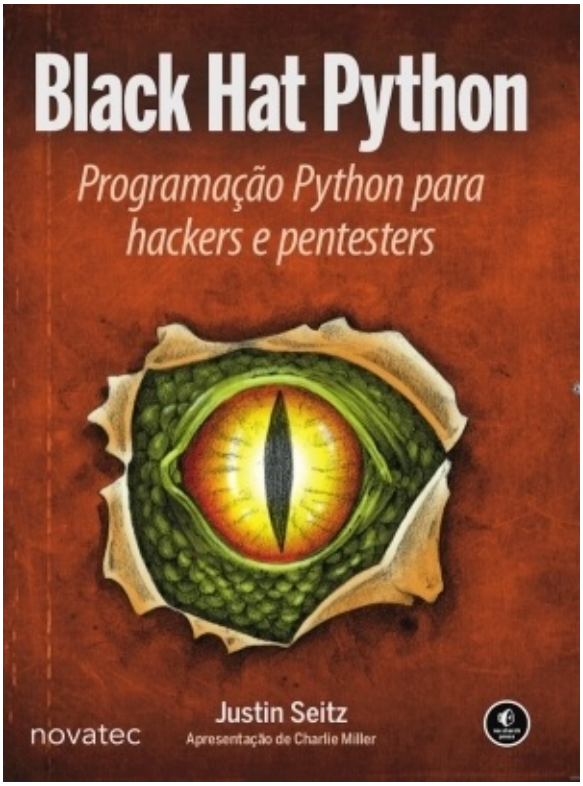
- Pode ser difícil entender como os programas em Python usam memória e

causam vazamentos.

- O módulo gc pode ajudar a entender quais objetos existem, mas não têm informação sobre como foram alocados em memória.
- O módulo nativo tracemalloc oferece ferramentas poderosas para entender a origem dos problemas de uso de memória.
- O tracemalloc só está disponível a partir do Python 3.4.

---

1 N. do T.: Em muitas linguagens de programação, como o Python, o C, o Perl, o PHP e o antigo BASIC, usa-se a palavra reservada print (em português, imprimir) para mostrar coisas na tela. Pode parecer estranho, para qualquer programador com menos de 40 anos de idade, usar a palavra "imprimir" para enviar coisas para o monitor em vez da impressora. A razão disso remonta aos primórdios da computação, em que não existia monitor de vídeo e todas as saídas do programa eram enviadas para uma espécie de máquina de escrever eletrônica chamada de teletipo. Print fazia sentido na época porque a saída do programa era realmente impressa. Com a evolução do hardware, os monitores de vídeo ficaram muito mais baratos e práticos, mas em algumas linguagens resolveu-se não mudar a sintaxe. Até hoje, muitos programadores falam "imprimir" para mostrar algo na tela. Neste livro, procuramos sempre traduzir o verbo print (quando não for a própria função) como "mostrar na tela".


Black Hat Python
Programação Python para hackers e pentesters
novatec
Justin Seitz
Apresentação de Charlie Miller

Black Hat Python

Seitz, Justin
9788575225578
200 páginas

Quando se trata de criar ferramentas eficazes e eficientes de hacking, o Python é a linguagem preferida da maioria dos analistas da área de segurança.

Mas como a mágica acontece?

Em Black Hat Python, o livro mais recente de Justin Seitz (autor do best-seller Gray Hat Python), você explorará o lado mais obscuro dos recursos do Python – fará a criação de sniffers de rede, manipulará pacotes, infectará máquinas virtuais, criará cavalos de Troia discretos e muito mais.

Você aprenderá a:
> Criar um cavalo de Troia para comando e controle usando o GitHub.
> Detectar sandboxing e automatizar tarefas comuns de malware, como fazer logging de teclas e capturar imagens de tela.
> Escalar privilégios do Windows por meio de um controle criativo de processo.

> Usar truques forenses de ataque à memória para obter hashes de senhas e injetar shellcode em uma máquina virtual.
> Estender o Burp Suite, que é uma ferramenta popular para web hacking.
> Explorar a automação do Windows COM para realizar um ataque do tipo man-in-the-browser.
> Obter dados de uma rede, principalmente de forma sub-reptícia.

Técnicas usadas por pessoas da área e desafios criativos ao longo de toda a obra mostrarão como estender os hacks e criar seus próprios exploits.

Quando se trata de segurança ofensiva, sua habilidade para criar ferramentas eficazes de forma imediata será indispensável.

Saiba como fazer isso em Black Hat Python.


JOVEM E
BEM-SUCEDIDO
Um guia para a realização
profissional e financeira
TRABALHO
DINHEIRO
ESTUDOS
novatec
Juliano Niederauer

Jovem e Bem-sucedido

Niederauer, Juliano

9788575225325

192 páginas

Jovem e Bem-sucedido é um verdadeiro guia para quem deseja alcançar a realização profissional e a financeira o mais rápido possível. Repleto de dicas e histórias interessantes vivenciadas pelo autor, o livro desmistifica uma série de crenças relativas aos estudos, ao trabalho e ao dinheiro.

Tem como objetivo orientar o leitor a planejar sua vida desde cedo, possibilitando que se torne bem-sucedido em pouco tempo e consiga manter essa realização no decorrer dos anos. As três perspectivas abordadas são:

ESTUDOS: mostra que os estudos vão muito além da escola ou faculdade. Aborda as melhores práticas de estudo e a aquisição dos conhecimentos ideais e nos momentos certos.

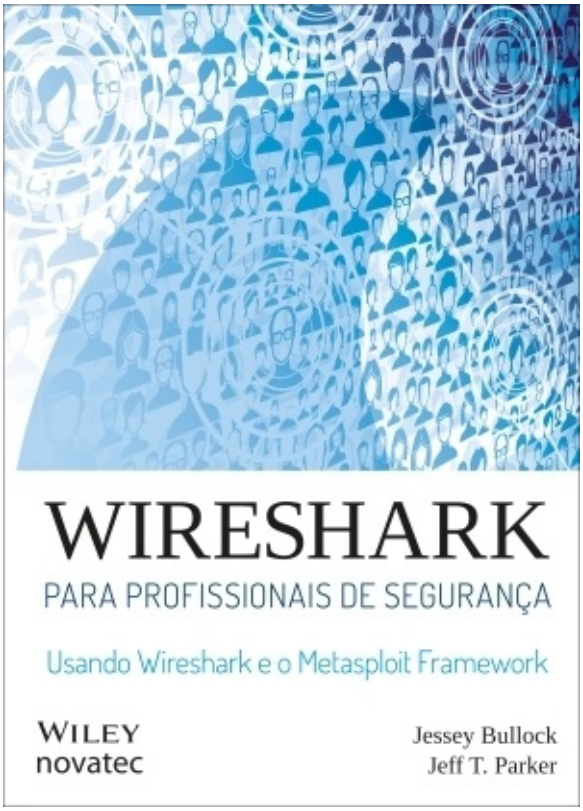
TRABALHO: explica como você pode se tornar um profissional moderno, identificando oportunidades e aumentando cada vez mais suas fontes de renda. Fornece ainda dicas valiosas para desenvolver as

habilidades mais valorizadas no mercado de trabalho.

DINHEIRO: explica como assumir o controle de suas finanças, para, então, começar a investir e multiplicar seu patrimônio. Apresenta estratégias de investimentos de acordo com o momento de vida de cada um, abordando as vantagens e desvantagens de cada tipo de investimento.

Jovem e Bem-sucedido apresenta ideias que o acompanharão a vida toda, realizando importantes mudanças no modo como você planeja estudar, trabalhar e lidar com o dinheiro.

Compre agora e leia


WIRESHARK
PARA PROFISSIONAIS DE SEGURANÇA
Usando Wireshark e o Metasploit Framework
WILEY
novatec
Jessey Bullock
Jeff T. Parker

Wireshark para profissionais de segurança

Bullock, Jessey

9788575225998

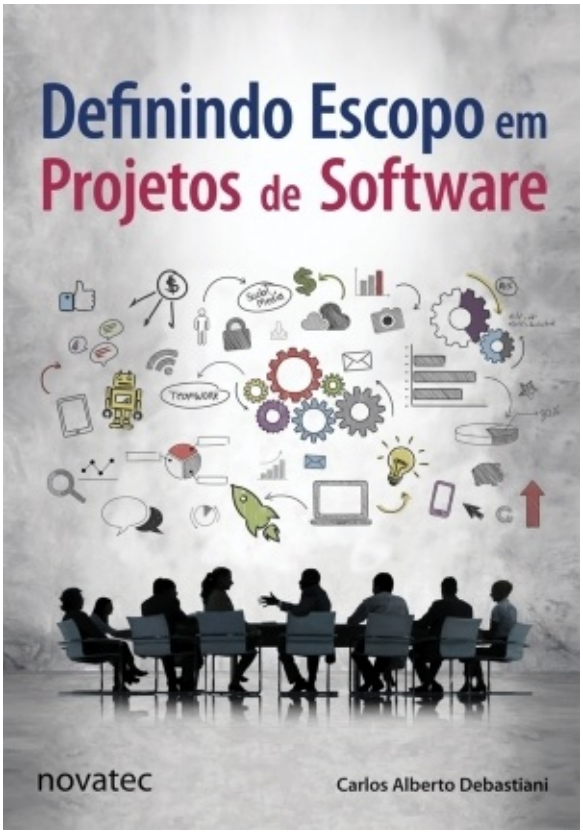
320 páginas

Um guia essencial para segurança de rede e para o Wireshark – um conjunto de ferramentas repleto de recursos

O analisador de protocolos de código aberto Wireshark é uma ferramenta de uso consagrado em várias áreas, incluindo o campo da segurança. O Wireshark disponibiliza um conjunto eficaz de recursos que permite inspecionar a sua rede em um nível microscópico. Os diversos recursos e o suporte a vários protocolos fazem do Wireshark uma ferramenta de segurança de valor inestimável, mas também o tornam difícil ou intimidador para os iniciantes que queiram conhecê-lo. Wireshark para profissionais de segurança é a resposta: ele ajudará você a tirar proveito do Wireshark e de ferramentas relacionadas a ele, por exemplo, a aplicação de linha de comando TShark, de modo rápido e eficiente. O conteúdo inclui uma introdução completa ao Metasploit, que é uma ferramenta de ataque eficaz, assim como da linguagem popular de scripting Lua.

Este guia extremamente prático oferece o insight necessário para você

aplicar o resultado de seu aprendizado na vida real com sucesso. Os exemplos mostram como o Wireshark é usado em uma rede de verdade, com o ambiente virtual Docker disponibilizado; além disso, princípios básicos de rede e de segurança são explicados em detalhes para ajudar você a entender o porquê, juntamente com o como. Ao usar a distribuição Kali Linux para testes de invasão, em conjunto com o laboratório virtual e as capturas de rede disponibilizadas, você poderá acompanhar os diversos exemplos ou até mesmo começar a pôr em prática imediatamente o seu conhecimento em um ambiente de rede seguro. A experiência prática torna-se mais valiosa ainda pela ênfase em uma aplicação coesa, ajudando você a explorar vulnerabilidades e a expandir todas as funcionalidades do Wireshark, estendendo-as ou integrando-as com outras ferramentas de segurança.


Definindo Escopo em
Projetos de Software
novatec
Carlos Alberto Debastiani

Definindo Escopo em Projetos de Software

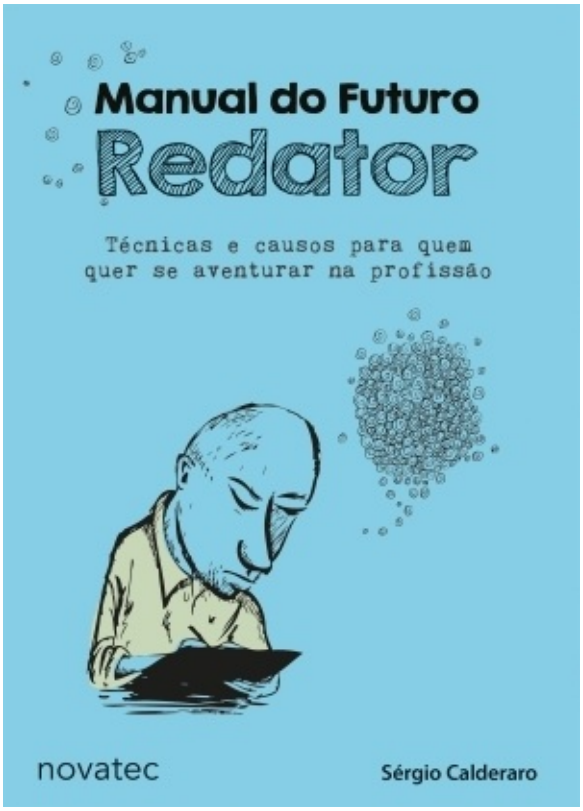
Debastiani, Carlos Alberto

9788575224960

144 páginas

Definindo Escopo em Projetos de Software é uma obra que pretende tratar, de forma clara e direta, a definição de escopo como o fator mais influente no sucesso dos projetos de desenvolvimento de sistemas, uma vez que exerce forte impacto sobre seus custos. Abrange diversas áreas do conhecimento ligadas ao tema, abordando desde questões teóricas como a normatização e a definição das características de engenharia de software, até questões práticas como métodos para coleta de requisitos e ferramentas para desenho e projeto de soluções sistêmicas.

Utilizando uma linguagem acessível, diversas ilustrações e citações de casos vividos em sua própria experiência profissional, o autor explora, de forma abrangente, os detalhes que envolvem a definição de escopo, desde a identificação das melhores fontes de informação e dos envolvidos na tomada de decisão, até as técnicas e ferramentas usadas no levantamento de requisitos, no projeto da solução e nos testes de aplicação.


Manual do Futuro
Redator
Técnicas e causos para quem
quer se aventurar na profissão
novatec
Sérgio Calderaro

Manual do Futuro Redator

Calderaro, Sérgio

9788575224908

120 páginas

Você estuda ou está pensando em estudar Publicidade, Jornalismo ou Letras? Gosta de ler e escrever e quer dicas de quem já passou por poucas e boas em agências de publicidade, redações de jornal e editoras? Quer conhecer causos curiosos de um profissional do texto que já deu aulas em universidades do Brasil e da Europa e trabalhou como assessor de Imprensa e Divulgação em uma das maiores embaixadas brasileiras do exterior?

O Manual do futuro redator traz tudo isso e muito mais. Em linguagem ágil e envolvente, mescla orientações técnicas a saborosas histórias do dia a dia de um profissional com duas décadas e meia de ofício. Esta obra inédita em sua abordagem pretende fazer com que você saiba onde está se metendo antes de decidir seu caminho. Daí pra frente, a decisão será sua. Vai encarar?