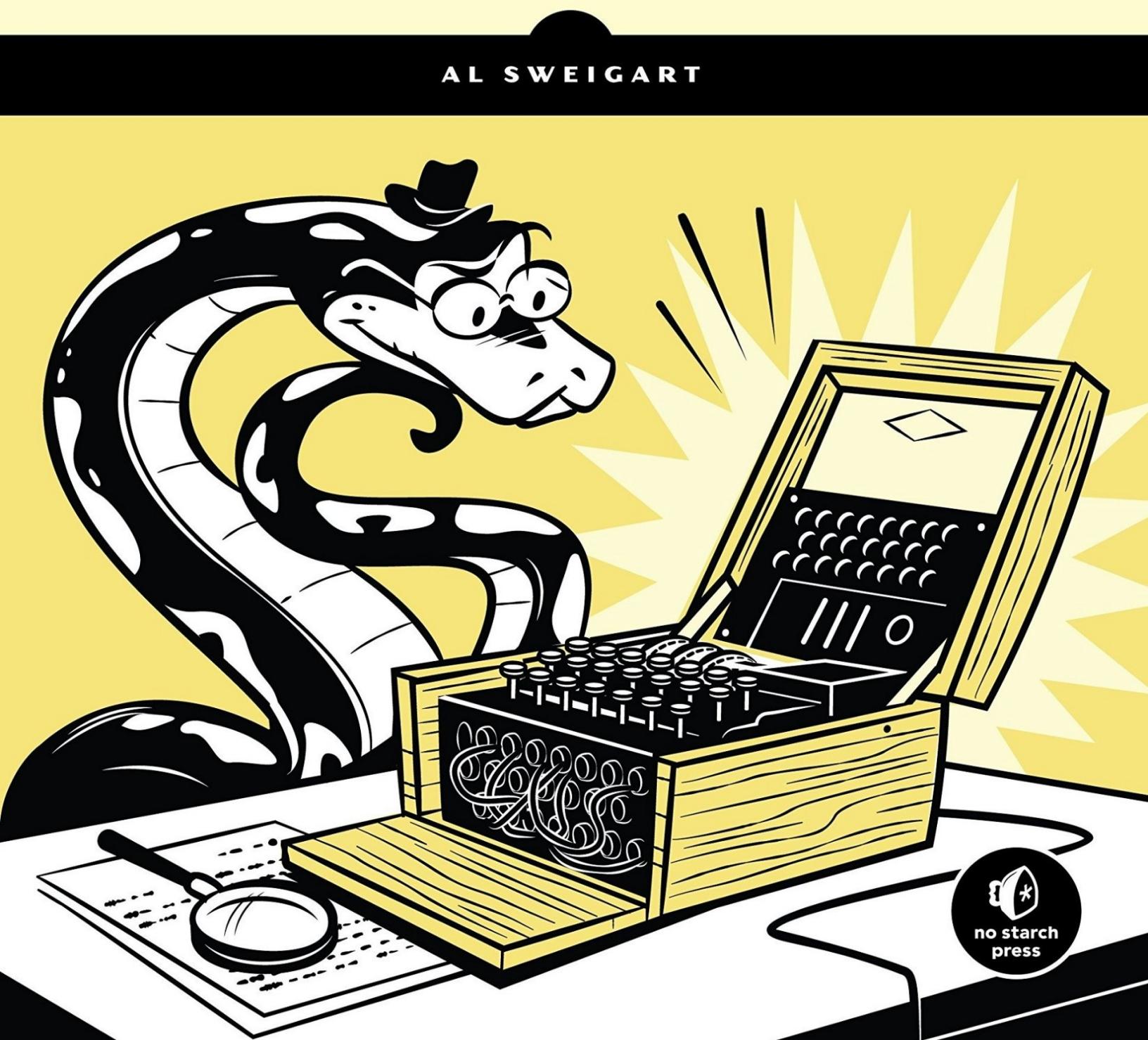


CRACKING CODES WITH PYTHON

AN INTRODUCTION TO
BUILDING AND BREAKING CIPHERS

AL SWEIGART



INTRODUÇÃO

"Eu não pude deixar de ouvir, provavelmente porque eu estava escutando".

-Anônimo



Se você pudesse viajar de volta ao início dos anos 90 com este livro, o conteúdo do [Capítulo 23](#) que implementasse parte da cifra RSA seria ilegal para ser exportado para fora dos Estados Unidos. Como as mensagens criptografadas com o RSA são impossíveis de serem hackeadas, a exportação de software de criptografia como o RSA foi considerada uma questão de segurança nacional e exigiu a aprovação do Departamento de Estado. De fato, a criptografia forte era regulada no mesmo nível dos tanques, mísseis e lança-chamas.

Em 1990, Daniel J. Bernstein, um estudante da Universidade da Califórnia, Berkeley, queria publicar um trabalho acadêmico que apresentasse o código-fonte de seu sistema de criptografia Snuffle. O governo dos EUA informou-o de que ele precisaria se tornar um revendedor autorizado de armas antes de poder postar seu código-fonte na Internet. O governo também lhe disse que lhe negaria uma licença de exportação se solicitasse uma, porque sua tecnologia era muito segura.

A Electronic Frontier Foundation, uma jovem organização digital de liberdades civis, representou Bernstein em *Bernstein vs. Estados Unidos*. Pela primeira vez, os tribunais decidiram que o código de software escrito era protegido pela Primeira Emenda e que as leis de controle de exportação sobre criptografia violavam os direitos da Primeira Emenda de Bernstein.

Agora, a criptografia forte está na base de uma grande parte da economia global, salvaguardando empresas e sites de comércio eletrônico usados por milhões de compradores da Internet todos os dias. As previsões da comunidade de inteligência de que o software de criptografia se tornaria uma grave ameaça à segurança nacional eram infundadas.

Mas, tão recentemente quanto nos anos 90, espalhar esse conhecimento

livremente (como este livro faz) teria te colocado na prisão por tráfico de armas. Para uma história mais detalhada da batalha legal pela liberdade de criptografia, leia o livro de Steven Levy, *Crypto: Como o Código Revolta Vence o Governo, Salvando a Privacidade na Era Digital* (Penguin, 2001).

Quem deveria ler esse livro?

Muitos livros ensinam os iniciantes a escrever mensagens secretas usando cifras. Alguns livros ensinam aos iniciantes como hackear cifras. Mas nenhum livro ensina aos iniciantes como programar computadores para hackear cifras. Este livro preenche essa lacuna.

Este livro é para aqueles que estão curiosos sobre criptografia, hacking ou criptografia. As cifras deste livro (com exceção da codificação de chave pública nos [capítulos 23 e 24](#)) têm séculos de existência, mas qualquer laptop tem o poder computacional de cortá-las. Nenhuma organização ou indivíduo moderno usa mais essas cifras, mas ao aprendê-las, você aprenderá sobre as bases nas quais a criptografia foi construída e como os hackers podem quebrar a criptografia fraca.

NOTA

As cifras que você aprenderá neste livro são divertidas de se brincar, mas não fornecem segurança verdadeira. Não use nenhum dos programas de criptografia deste livro para proteger seus arquivos reais. Como regra geral, você não deve confiar nas cifras que cria. Cifras do mundo real estão sujeitas a anos de análise por criptógrafos profissionais antes de serem colocadas em uso.

Este livro é também para pessoas que nunca programaram antes. Ele ensina conceitos básicos de programação usando a linguagem de programação Python, que é uma das melhores linguagens para iniciantes. Ele tem uma curva de aprendizado suave que os novatos de todas as idades podem dominar, mas também é uma linguagem poderosa usada por desenvolvedores profissionais de software. O Python é executado no Windows, no MacOS, no Linux e até no Raspberry Pi, e é gratuito para download e uso. (Veja “[Baixando e Instalando o Python](#)” na [página xxv](#) para instruções.)

Neste livro, usarei o termo *hacker* com frequência. A palavra tem duas definições. Um hacker pode ser uma pessoa que estuda um sistema (como as regras de uma cifra ou um software) para compreendê-lo tão bem que não é limitado pelas regras originais desse sistema e pode modificá-lo de maneiras

criativas. Um hacker também pode ser um criminoso que invade sistemas de computadores, viola a privacidade das pessoas e causa danos. Este livro usa o termo no primeiro sentido. Hackers são legais. Criminosos são apenas pessoas que pensam que estão sendo espertos ao quebrar coisas.

O que há neste livro?

Os primeiros capítulos apresentam conceitos básicos de Python e criptografia. Posteriormente, os capítulos geralmente alternam entre explicar um programa para uma cifra e, em seguida, explicar um programa que hackeia essa cifra. Cada capítulo também inclui questões práticas para ajudá-lo a rever o que aprendeu.

- **O Capítulo 1: Fazendo Ferramentas de Criptografia de Papel** abrange algumas ferramentas de papel simples, mostrando como a criptografia foi feita antes dos computadores.
- **Capítulo 2: Programação no Shell Interativo** explica como usar o shell interativo do Python para brincar com o código uma linha por vez.
- **Capítulo 3: Strings e programas de** escrita abrange a criação de programas completos e introduz o tipo de dados de string usado em todos os programas deste livro.
- **Capítulo 4: A cifra reversa** explica como escrever um programa simples para sua primeira cifra.
- **Capítulo 5: A Cifra César** cobre uma cifra básica inventada há milhares de anos.
- **Capítulo 6: Hacking the Cesar Cipher com Brute-Force** explica a técnica de hacking de força bruta e como usá-lo para descriptografar mensagens sem a chave de criptografia.
- **Capítulo 7: Criptografar com a Transposição A cifra** introduz a cifra de transposição e um programa que criptografa mensagens com ela.
- **Capítulo 8: Descriptografar com a Transposição A cifra** cobre a segunda metade da cifra de transposição: ser capaz de descriptografar mensagens com uma chave.
- **Capítulo 9: Programação de um programa para testar seu programa** apresenta a técnica de programação de testes de programas

com outros programas.

- **O Capítulo 10: Criptografando e descriptografando arquivos** explica como escrever programas que leem e gravam arquivos no disco rígido.
- **Capítulo 11: Detectando o inglês Programaticamente** descreve como fazer com que o computador detecte sentenças em inglês.
- **Capítulo 12: Hackeando a Transposição A codificação** combina os conceitos dos capítulos anteriores para hackear a cifra de transposição.
- **Capítulo 13: Um módulo aritmético modular para a cifra afim** explica os conceitos matemáticos por trás da cifra afim.
- **Capítulo 14: Programando a Cifra Afimea** cobre a gravação de um programa de criptografia de criptografia afim.
- **Capítulo 15: Hacking the Affine Cipher** explica como escrever um programa para hackear a cifra afim.
- **Capítulo 16: Programando a Cifra de Substituição Simples** cobre a gravação de um programa de criptografia de cifra de substituição simples.
- **O Capítulo 17: Hackeando a Cifra de Substituição Simples** explica como escrever um programa para hackear a cifra simples de substituição.
- **Capítulo 18: Programando a Cifra de Vigenère** explica um programa para a cifra de Vigenère, uma cifra de substituição mais complexa.
- **Capítulo 19: Análise de Frequência** explora a estrutura das palavras em inglês e como usá-lo para hackear a cifra de Vigenère.
- **Capítulo 20: Hacking the Vigenère Cipher** abrange um programa para hackear a cifra Vigenère.
- **Capítulo 21: A Cifra de Pad Único** explica a cifra do pad de uso único e porque é matematicamente impossível hackear.
- **O Capítulo 22: Localizando e gerando números primos** mostra como escrever um programa que determina rapidamente se um

número é primo.

- **Capítulo 23: Gerando Chaves para a Cifra de Chave Pública** descreve criptografia de chave pública e como escrever um programa que gera chaves públicas e privadas.
- **Capítulo 24: Programando a Cifra de Chave Pública** explica como escrever um programa para uma cifra de chave pública, que você não pode hackear usando apenas um laptop.
- O apêndice, **Debugging Python Code**, mostra como usar o depurador do IDLE para localizar e corrigir erros em seus programas.

Como usar este livro

Cracking Codes com Python é diferente de outros livros de programação porque se concentra no código-fonte de programas completos. Em vez de ensinar conceitos de programação e deixar para você descobrir como criar seus próprios programas, este livro mostra programas completos e explica como eles funcionam.

Em geral, você deve ler os capítulos deste livro em ordem. Os conceitos de programação se baseiam naqueles dos capítulos anteriores. No entanto, o Python é uma linguagem tão legível que, após os primeiros capítulos, você provavelmente pode avançar para capítulos posteriores e juntar o que o código faz. Se você pular na frente e se sentir perdido, volte aos capítulos anteriores.

Digitando o código fonte

Ao ler este livro, sugiro que você *digite manualmente o código-fonte deste livro no Python*. Fazer isso definitivamente ajudará você a entender melhor o código.

Ao digitar o código-fonte, não inclua os números de linha que aparecem no início de cada linha. Esses números não fazem parte dos programas reais e os usamos apenas para se referir a linhas específicas no código. Mas, além dos números de linha, insira o código exatamente como aparece, incluindo as letras maiúsculas e minúsculas.

Você também notará que algumas das linhas não começam na borda mais à esquerda da página, mas são recuadas por quatro, oito ou mais espaços. Certifique-se de inserir o número correto de espaços no início de cada linha para evitar erros.

Mas se você preferir não digitar o código, poderá fazer o download dos arquivos

de código fonte no site deste livro em <https://www.nostarch.com/crackingcodes/> .

Verificação de erros ortográficos

Embora inserir manualmente o código-fonte dos programas seja útil para o aprendizado do Python, você pode ocasionalmente criar erros de digitação que causam erros. Esses erros de digitação podem ser difíceis de detectar, especialmente quando seu código-fonte é muito longo.

Para verificar com rapidez e facilidade erros no código-fonte digitado, você pode copiar e colar o texto na ferramenta de comparação on-line no site do livro em <https://www.nostarch.com/crackingcodes/> . A ferramenta de comparação mostra as diferenças entre o código-fonte no livro e o seu.

Convenções de codificação neste livro

Este livro não foi projetado para ser um manual de referência; é um guia prático para iniciantes. Por esse motivo, o estilo de codificação às vezes vai contra as melhores práticas, mas essa é uma decisão consciente de tornar o código mais fácil de aprender. Este livro também ignora os conceitos teóricos da ciência da computação.

Os programadores veteranos podem apontar maneiras pelas quais o código neste livro poderia ser alterado para melhorar a eficiência, mas este livro está principalmente preocupado em fazer com que os programas funcionem com o mínimo de esforço.

Recursos online

O site deste livro (<https://www.nostarch.com/crackingcodes/>) inclui muitos recursos úteis, incluindo arquivos para download dos programas e exemplos de soluções para as questões práticas. Este livro cobre cifras clássicas completamente, mas como há sempre mais a aprender, inclui também sugestões para ler mais sobre muitos dos tópicos apresentados neste livro.

Fazendo o download e instalando o Python

Antes de começar a programar, você precisará instalar o *interpretador Python* , que é o software que executa as instruções que você irá escrever na linguagem Python. Eu vou me referir ao “interpretador Python” como “Python” a partir de agora.

Faça o download do Python para Windows, macOS e Ubuntu gratuitamente em <https://www.python.org/downloads/> . Se você baixar a versão mais recente, todos

os programas deste livro deverão funcionar.

NOTA

Certifique-se de baixar uma versão do Python 3 (como 3.6). Os programas neste livro são escritos para rodar no Python 3 e podem não ser executados corretamente, se forem, no Python 2.

Instruções para o Windows

No Windows, baixe o instalador do Python, que deve ter um nome de arquivo terminado em *.msi* e clique duas vezes nele. Siga as instruções que o instalador exibe na tela para instalar o Python, conforme listado aqui:

1. Selecione **Instalar agora** para iniciar a instalação.
2. Quando a instalação estiver concluída, clique em **Fechar**.

instruções macOS

No macOS, baixe o arquivo *.dmg* para sua versão do macOS no site e clique duas vezes nele. Siga as instruções que o instalador exibe na tela para instalar o Python, conforme listado aqui:

1. Quando o pacote DMG abrir em uma nova janela, clique duas vezes no arquivo *Python.mpkg*. Você pode ter que digitar a senha de administrador do seu computador.
2. Clique em **Continuar** até a seção Bem-vindo e clique em **Concordo** para aceitar a licença.
3. Selecione **HD Macintosh** (ou o nome do seu disco rígido) e clique em **Instalar**.

Instruções do Ubuntu

Se você estiver executando o Ubuntu, instale o Python no Ubuntu Software Center seguindo estas etapas:

1. Abra o Ubuntu Software Center.
2. Digite Python na caixa de pesquisa no canto superior direito da janela.
3. Selecione **IDLE (usando o Python 3.6)** ou qualquer que seja a versão mais recente.
4. Clique em **Instalar**.

Você pode ter que digitar a senha do administrador para concluir a instalação.

Baixando pyperclip.py

Quase todos os programas deste livro usam um módulo personalizado que escrevi chamado *pyperclip.py*. Este módulo fornece funções que permitem que seus programas copiem e colem texto na área de transferência. Ele não vem com o Python, então você precisa baixá-lo em <https://www.nostarch.com/crackingcodes/>.

Este arquivo deve estar na mesma pasta (também chamada de *diretório*) que os arquivos de programa do Python que você escreve. Caso contrário, você verá a seguinte mensagem de erro ao tentar executar seus programas:

ImportError: Nenhum módulo chamado pyperclip

Agora que você baixou e instalou o interpretador Python e o módulo *pyperclip.py*, vamos ver onde você vai escrever seus programas.

Iniciando o IDLE

Enquanto o interpretador Python é o software que executa seus programas em Python, o software do *ambiente de desenvolvimento interativo (IDLE)* é onde você vai escrever seus programas, da mesma forma que um processador de texto. O IDLE é instalado quando você instala o Python. Para iniciar o IDLE, siga estas etapas:

- No Windows 7 ou mais recente, clique no ícone Iniciar no canto inferior esquerdo da tela, digite **IDLE** na caixa de pesquisa e selecione **IDLE (Python 3.6 de 64 bits)**.
- No macOS, abra o Finder, clique em **Aplicativos**, clique em **Python 3.6** e, em seguida, clique no ícone **IDLE**.
- No Ubuntu, selecione **Aplicativos ▶ Acessórios ▶ Terminal** e, em seguida, insira `idle3`. (Você também poderá clicar em **Aplicativos** na parte superior da tela, selecionar **Programação** e, em seguida, clicar em **IDLE 3.**.)

Não importa qual sistema operacional você esteja executando, a janela IDLE deve se parecer com a [Figura 1](#). O texto do cabeçalho pode ser ligeiramente diferente dependendo da sua versão específica do Python.

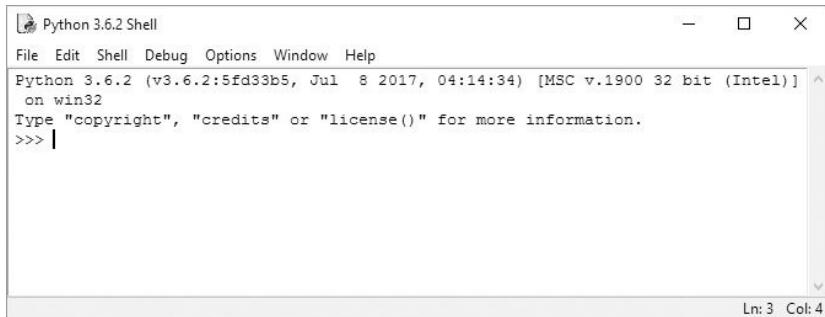


Figura 1: A janela IDLE

Esta janela é chamada de *shell interativo*. Um shell é um programa que permite digitar instruções no computador, da mesma forma que o Terminal no macOS ou o Prompt de Comando do Windows. Às vezes você vai querer executar pequenos trechos de código em vez de escrever um programa completo. O shell interativo do Python permite inserir instruções para o software do interpretador Python, que o computador lê e executa imediatamente.

Por exemplo, digite o seguinte no shell interativo ao lado do prompt >>> :

```
>>> print ('Olá, mundo!')
```

Pressione enter e o shell interativo deve exibir isso em resposta:

Olá Mundo!

Resumo

Antes da introdução dos computadores inaugurar a criptografia moderna, era impossível quebrar muitos códigos usando apenas lápis e papel. Embora a computação tenha tornado muitos dos códigos antigos e clássicos vulneráveis a ataques, eles ainda são divertidos de aprender. Escrever programas de criptoanálise que quebram essas cifras é uma ótima maneira de aprender a programar.

No [Capítulo 1](#), começaremos com algumas ferramentas básicas de criptografia para criptografar e descriptografar mensagens sem o auxílio de computadores.

Vamos pirar.

1

FABRICAÇÃO DE FERRAMENTAS DE CRIPTOGRAFIA DE PAPEL

"O gênio da criptografia está fora da garrafa."

- Jan Koum, fundador do WhatsApp



Antes de começarmos a escrever programas de codificação, vamos dar uma olhada no processo de criptografar e descriptografar apenas com lápis e papel. Isso ajudará você a entender como as cifras funcionam e a matemática usada para produzir suas mensagens secretas. Neste capítulo, você aprenderá o que entendemos por criptografia e como os códigos são diferentes das cifras. Então você vai usar uma simples cifra chamada cifra de César para criptografar e descriptografar mensagens usando papel e lápis.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- O que é criptografia?
- Códigos e cifras
- A cifra de César
- Rodas cifradas
- Fazendo criptografia com aritmética
- Criptografia dupla

O que é criptografia?

Historicamente, qualquer pessoa que tenha precisado compartilhar segredos com outras pessoas, como espiões, soldados, hackers, piratas, comerciantes, tiranos e ativistas políticos, tem confiado na criptografia para garantir que seus segredos permaneçam em segredo. *Criptografia* é a ciência do uso de códigos secretos. Para entender como é a criptografia, observe as duas partes do texto a seguir:

nyr N.vNwz5uNz5Ns6620Nz0N3z2v N yvNwz9vNz5N6!9Nyvr9 y0QNnvNwv tyNz Nw964N6!9N5vzxys690,N.vN2z5u- 3vNz Nr Ny64v,N.vN!644!5ztr vNz N 6N6 yv90,Nr5uNz Nsv!64v0N yvN7967v9 BN6wNr33Q N-m63 rz9v	INN2 Nuwy,N9,vNN!vNrBN3zyN4vN N6 Qvv0z6nvN,7N0yy4N 4 zzzNN vyN,NN99z0zz6wz0y3vv26 9 w296vyNNrrNyQst.560N94Nu5y rN5nz5vv5!6v63zNr5. N75sz6966NNvw6 zu0 wtNxs6t 49NrN3Ny9Nvzy!
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

O texto à esquerda é uma mensagem secreta que foi *criptografada* ou

transformada em um código secreto. É completamente ilegível para quem não sabe como *decifrá-* lo ou transformá-lo na mensagem original em inglês. A mensagem à direita é um jargão aleatório sem nenhum significado oculto. A criptografia mantém uma mensagem secreta de outras pessoas que não conseguem decifrá-la, mesmo se colocarem as mãos na mensagem criptografada. *Uma mensagem criptografada parece exatamente um absurdo aleatório.*

Um *criptógrafo* usa e estuda códigos secretos. Claro, essas mensagens secretas nem sempre permanecem em segredo. Um *criptoanalista*, também chamado de *quebrador de código* ou *hacker*, pode hackar códigos secretos e ler as mensagens criptografadas de outras pessoas. Este livro ensina como criptografar e descriptografar mensagens usando várias técnicas. Mas infelizmente (ou felizmente), o tipo de hacking que você aprenderá neste livro não é perigoso o suficiente para causar problemas com a lei.

Códigos vs. Cifras

Ao contrário das cifras, os *códigos* são feitos para serem comprehensíveis e publicamente disponíveis. Os códigos substituem as mensagens por símbolos que qualquer pessoa deve procurar para traduzir em uma mensagem.

No início do século XIX, um código conhecido surgiu do desenvolvimento do telégrafo elétrico, que permitia a comunicação quase instantânea entre os continentes por meio de fios. Enviar mensagens pelo telégrafo era muito mais rápido do que a alternativa anterior de enviar um cavaleiro carregando um saco de cartas. No entanto, o telégrafo não podia enviar diretamente cartas escritas desenhadas no papel. Em vez disso, ele poderia enviar apenas dois tipos de pulsos elétricos: um pulso curto chamado de "ponto" e um pulso longo chamado de "traço".

Para converter letras do alfabeto em pontos e traços, você precisa de um sistema de codificação para traduzir o inglês para pulsos elétricos. O processo de converter o inglês em pontos e traços para enviar um telégrafo é chamado de *codificação*, e o processo de traduzir pulsos elétricos para o inglês quando uma mensagem é recebida é chamado de *decodificação*. O código usado para codificar e decodificar mensagens sobre telégrafos (e mais tarde, rádio) foi chamado de *código Morse*, conforme mostrado na [Tabela 1-1](#). O código Morse foi desenvolvido por Samuel Morse e Alfred Vail.

Tabela 1-1: Codificação Internacional de Código Morse

Carta Codificação Carta Codificação Número Codificação

UMA	• —	N	— •	1	• — — —
B	• • • •	O	— — —	2	• • — —
C	— • — •	P	• — — •	3	• • • — —
D	— • •	Q	— — • —	4	• • • • —
E	•	R	• — •	5	• • • • •
F	• • — •	S	• • •	6	• • • • •
G	— — •	T	—	7	• — • • •
H	• • • •	você	• • —	8	— — — •
Eu	• •	V	• • • —	9	— — — —
J	• — — —	W	• — —	0	— — — — —
K	— • —	X	• • • —		
eu	• • • •	Y	— — —		
M	— —	Z	— — • •		

Ao tocar pontos e traços com um telégrafo de um botão, um operador de telégrafo poderia comunicar uma mensagem em inglês para alguém do outro lado do mundo quase que instantaneamente! (Para saber mais sobre o código Morse, visite <https://www.nostarch.com/crackingcodes/>.)

Em contraste com os códigos, uma *cifra* é um tipo específico de código destinado a manter as mensagens em segredo. Você pode usar uma cifra para

transformar um texto em inglês compreensível, chamado *plaintext*, em algo sem sentido que esconde uma mensagem secreta, chamada de *texto cifrado*. Uma cifra é um conjunto de regras para converter entre texto simples e texto cifrado. Essas regras geralmente usam uma chave secreta para criptografar ou descriptografar apenas os comunicadores. Neste livro, você aprenderá várias cifras e escreverá programas para usar essas cifras para criptografar e descriptografar o texto. Mas primeiro, vamos criptografar as mensagens manualmente usando ferramentas de papel simples.

A Cifra César

A primeira cifra que você vai aprender é a cifra de César, que tem o nome de Júlio César, que a usou há 2000 anos. A boa notícia é que é simples e fácil de aprender. A má notícia é que, por ser tão simples, também é fácil para um criptoanalista quebrar. No entanto, ainda é um exercício de aprendizagem útil.

A cifra de César funciona substituindo cada letra de uma mensagem por uma nova letra depois de mudar o alfabeto. Por exemplo, Júlio César substituiu as letras em suas mensagens trocando as letras do alfabeto por três e depois substituindo cada letra pelas letras de seu alfabeto.

Por exemplo, todo A na mensagem seria substituído por D, todo B seria E, e assim por diante. Quando César precisava trocar letras no final do alfabeto, como Y, ele envolvia o início do alfabeto e mudava três para B. Nesta seção, vamos criptografar uma mensagem manualmente usando a cifra de César.

A roda cifrada

Para facilitar a conversão de texto simples em texto cifrado usando a cifra de César, usaremos uma *roda de cifra*, também chamada de *disco de cifra*. A roda cifrada consiste em dois anéis de letras; cada anel é dividido em 26 slots (para um alfabeto de 26 letras). O anel externo representa o alfabeto de texto plano e o anel interno representa as letras correspondentes no texto cifrado. O anel interno também numera as letras de 0 a 25. Esses números representam a *chave de criptografia*, que nesse caso é o número de letras necessárias para mudar de A para a letra correspondente no anel interno. Como a mudança é circular, a mudança com uma chave maior que 25 faz com que os alfabetos se envolvam, de modo que a mudança de 26 seria a mesma que a mudança de 0, a mudança de 27 seria a mesma de 1 e assim por diante.

Você pode acessar uma roda de criptografia virtual on-line em

<https://www.nostarch.com/crackingcodes/>. A Figura 1-1 mostra o que parece. Para girar a roda, clique nela e move o cursor do mouse até que a configuração desejada esteja no lugar. Em seguida, clique no mouse novamente para parar a roda de girar.

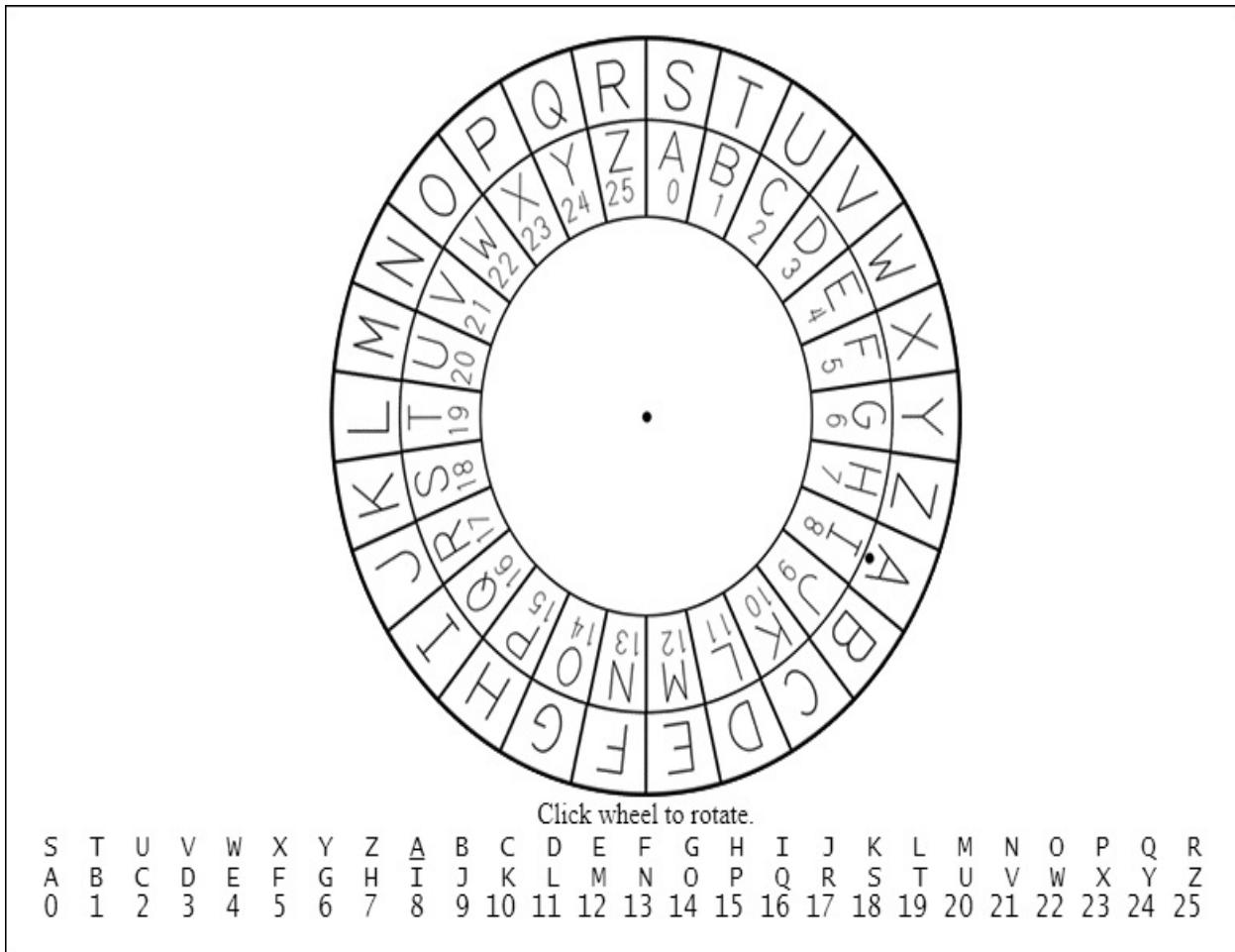


Figura 1-1: A roda de códigos online

Uma roda de criptografia de papel imprimível também está disponível no site do livro. Recorte os dois círculos e coloque-os um em cima do outro, colocando o menor no meio do maior. Insira um alfinete ou brad através do centro de ambos os círculos para que você possa girá-los no lugar.

Usando o papel ou a roda virtual, você pode criptografar manualmente as mensagens secretas.

Criptografando com a roda de codificação

Para começar a criptografar, escreva sua mensagem em inglês em um pedaço de papel. Para este exemplo, vamos criptografar a mensagem A SENHA SECRETA

É ROSEBUD. Em seguida, gire a roda interna da roda cifrada até que suas ranhuras coincidam com as ranhuras na roda externa. Observe o ponto ao lado da letra A na roda externa. Anote o número na roda interna ao lado deste ponto. Esta é a chave de criptografia.

Por exemplo, na [Figura 1-1](#), o círculo externo A está acima do número 8 do círculo interno. Usaremos essa chave de criptografia para criptografar a mensagem em nosso exemplo, conforme mostrado na [Figura 1-2](#).

T	H	E	S	E	C	R	E	T	P	A	S	S	W	O	R	D	I	S	R	O	S	E	B	U	D
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
B	P	M	A	M	K	Z	M	B	X	I	A	A	E	W	Z	L	Q	A	Z	W	A	M	J	C	L

Figura 1-2: Criptografando uma mensagem com uma chave de cifra de Caesar de 8

Para cada letra da mensagem, encontre-a no círculo externo e substitua-a pela letra correspondente no círculo interno. Neste exemplo, a primeira letra da mensagem é T (o primeiro T em “THE SECRET...”), então encontre o letra T no círculo externo e, em seguida, encontrar a letra correspondente no círculo interno, que é a letra B. Portanto, a mensagem secreta sempre substitui um T por um B. (Se você estivesse usando uma chave de criptografia diferente, cada T no texto simples seria substituído por uma letra diferente.) A próxima letra na mensagem é H, que se transforma em P. A letra E se transforma em M. Cada letra na roda externa sempre criptografa para a mesma letra na roda interna. Para poupar tempo, depois de procurar o primeiro T em “O SEGREDO...” e ver que ele criptografa para B, você pode substituir cada T na mensagem por B, então você só precisa procurar uma letra uma vez.

Depois de criptografar a mensagem inteira, a mensagem original, A SENHA SECRETA É ROSEBUD, torna-se BPM AMKZMB XIAAEWZL QA ZWAMJCL. Observe que os caracteres que não são letras, como os espaços, não são alterados.

Agora você pode enviar esta mensagem criptografada para alguém (ou guardá-la por si mesmo), e ninguém poderá lê-la a menos que você diga a ela a chave de criptografia secreta. Certifique-se de manter a chave de criptografia em segredo; o texto cifrado pode ser lido por qualquer um que saiba que a mensagem foi criptografada com a chave 8.

Decifrando com a roda de cifra

Para descriptografar um texto cifrado, inicie a partir do círculo interno da roda

de cifra e, em seguida, move-se para o círculo externo. Por exemplo, digamos que você receba o texto cifrado IWT CTL EPHHLDGS XH HLDGSUXHW. Você não seria capaz de decifrar a mensagem a menos que você soubesse a chave (ou a menos que você fosse um hacker inteligente). Felizmente, seu amigo já lhe disse que eles usam a chave 15 para suas mensagens. A roda de criptografia para essa chave é mostrada na [Figura 1-3](#).



Figura 1-3: Uma roda de codificação configurada para a chave 15

Agora você pode alinhar a letra A no círculo externo (aquele com o ponto abaixo dela) sobre a letra no círculo interno que tem o número 15 (que é a letra P). Em seguida, encontre a primeira letra na mensagem secreta no círculo interno, que é I, e observe a letra correspondente no círculo externo, que é T. A segunda letra no texto cifrado, W, descriptografa a letra H. Decrypt o resto das letras no texto cifrado de volta para o texto simples, e você receberá a mensagem A NOVA SENHA É ESPADA, conforme mostrado na [Figura 1-4](#).

I	W	T	C	T	L	E	P	H	H	L	D	G	S	X	H	H	L	D	G	S	U	X	H	W
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
T	H	E	N	E	W	P	A	S	S	W	O	R	D	I	S	S	W	O	R	D	F	I	S	H

Figura 1-4: Descriptografar uma mensagem com uma chave de cifra de César de 15

Se você usou uma chave incorreta, como 16, a mensagem descriptografada seria SGD MDV OZRRVNQC HR RVNQCEHRG, que é ilegível. A menos que a chave correta seja usada, a mensagem descriptografada não será comprehensível.

Criptografando e Decodificando com Aritmética

A roda de criptografia é uma ferramenta conveniente para criptografar e descriptografar com a cifra de César, mas você também pode criptografar e descriptografar usando a aritmética. Para fazer isso, escreva as letras do alfabeto de A a Z com os números de 0 a 25 em cada letra. Comece com 0 abaixo de A, 1 abaixo de B e assim por diante até que 25 fique abaixo de Z. A [Figura 1-5](#) mostra

como deve ficar.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figura 1-5: Numerando o alfabeto de 0 a 25

Você pode usar esse código de letras para números para representar letras. Este é um conceito poderoso, porque permite que você faça matemática em letras. Por exemplo, se você representar as letras CAT como os números 2, 0 e 19, poderá adicionar 3 para obter os números 5, 3 e 22. Esses novos números representam as letras FDW, como mostra a [Figura 1-5](#). Você acabou de adicionar 3 à palavra *gato*! Mais tarde, poderemos programar um computador para fazer isso para nós.

Para usar a aritmética para criptografar com a cifra de César, encontre o número abaixo da letra que deseja criptografar e adicione o número da chave a ele. A soma resultante é o número sob a letra criptografada. Por exemplo, vamos criptografar o HELLO. COMO VOCÊ ESTÁ? usando a tecla 13. (Você pode usar qualquer número de 1 a 25 para a tecla.) Primeiro, encontre o número em H, que é 7. Em seguida, adicione 13 a este número: $7 + 13 = 20$. Porque o número 20 é sob a letra U, a letra H criptografa para U.

Da mesma forma, para criptografar a letra E (4), adicione $4 + 13 = 17$. O número acima de 17 é R, então E é criptografado para R e assim por diante.

Este processo funciona bem até a letra O. O número sob O é 14. Mas 14 mais 13 é 27, e a lista de números só sobe para 25. Se a soma do número da letra e a chave for 26 ou mais, você precisa subtrair 26 dele. Neste caso, $27 - 26 = 1$. A letra acima do número 1 é B, então O criptografa para B usando a chave 13. Quando você criptografa cada letra da mensagem, o texto cifrado será URYYYB. UBJ NER LBH?

Para descriptografar o texto cifrado, subtraia a chave em vez de adicioná-la. O número da letra do texto cifrado B é 1. Subtraia 13 de 1 para obter -12 . Como nossa regra “subtrair 26” para criptografar, quando o resultado é menor que 0 ao descriptografar, precisamos adicionar 26. Como $-12 + 26 = 14$, a letra do texto cifrado B descriptografa para O.

NOTA

Se você não sabe como adicionar e subtrair com números negativos, pode ler sobre isso em <https://www.nostarch.com/crackingcodes/>.

Como você pode ver, você não precisa de uma roda de criptografia para usar a cifra de César. Tudo o que você precisa é de um lápis, um pedaço de papel e alguma aritmética simples!

Por que a criptografia dupla não funciona

Você pode pensar que criptografar uma mensagem duas vezes usando duas chaves diferentes dobraria a força da criptografia. Mas esse não é o caso da cifra de César (e da maioria das outras cifras). Na verdade, o resultado da criptografia dupla é o mesmo que você obteria após uma criptografia normal. Vamos tentar duplicar a criptografia de uma mensagem para ver o motivo.

Por exemplo, se você criptografar a palavra GATINHO usando a tecla 3, você adicionará 3 ao número da letra de texto sem formatação e o texto cifrado resultante será NLWWHQ. Se você, então, criptografar NLWWHQ, desta vez usando a chave 4, o texto cifrado resultante seria RPAALU, porque você está adicionando 4 ao número da letra de texto simples. Mas isso é o mesmo que criptografar a palavra GATINHO uma vez com uma chave de 7.

Para a maioria das cifras, criptografar mais de uma vez não fornece força adicional. De fato, se você criptografar algum texto simples com duas chaves que somam 26, o texto cifrado resultante será o mesmo que o texto original!

Resumo

A cifra de César e outras cifras semelhantes foram usadas para criptografar informações secretas por vários séculos. Mas se você quiser criptografar uma mensagem longa, digamos, um livro inteiro, pode levar dias ou semanas para criptografar tudo manualmente. É aqui que a programação pode ajudar. Um computador pode criptografar e descriptografar uma grande quantidade de texto em menos de um segundo!

Para usar um computador para criptografia, você precisa aprender a *programar* ou instruir o computador a fazer as mesmas etapas que acabamos de fazer usando um idioma que o computador possa entender. Felizmente, aprender uma linguagem de programação como o Python não é tão difícil quanto aprender uma língua estrangeira como o japonês ou o espanhol. Você também não precisa saber muita matemática além de adição, subtração e multiplicação. Tudo o que você precisa é de um computador e deste livro!

Vamos seguir para o [Capítulo 2](#), onde aprenderemos a usar o shell interativo do Python para explorar o código de uma linha por vez.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Criptografe as seguintes entradas do *Dicionário do Diabo* de Ambrose Bierce com as chaves fornecidas:
 1. Com a tecla 4: "AMBIDEXTROUS: Capaz de escolher com igual habilidade o bolso direito ou esquerdo".
 2. Com a chave 17: "GUILLOTINE: Uma máquina que faz um francês encolher os ombros com um bom motivo."
 3. Com a chave 21: "IMPIEDADE: Sua irreverência em relação à minha divindade".
2. Descriptografar os seguintes textos cifrados com as chaves fornecidas:
 1. Com a tecla 15: "ZXAI: P RDHIJBT HDBTIXBTH LDGC QN HRDIRWBTC XC PBTGXRP PCS PBTGXRPCH XC HRDIAPCS."
 2. Com a tecla 4: "MQTSWXSV: E VMWEP EWTMVERX XS TYFPMG LSRSVW".
 3. Criptografar a seguinte frase com a chave 0: "Este é um exemplo bobo".
 4. Aqui estão algumas palavras e suas criptografias. Qual chave foi usada para cada palavra?
 1. ROSEBUD - LIMYVOX
 2. YAMAMOTO - PRDRDFKF
 3. ASTRONOMIA - HZAYVUVTF
 5. O que esta frase criptografada com chave 8 descriptografar com chave 9? "UMMSVMAA: Cvkwuuwv xibqmvkm qv xtivvqvo e zmdmvom bpib qa ewzbp epqtm."

2

PROGRAMAÇÃO NA CAIXA INTERATIVA

“O mecanismo analítico não tem pretensões de originar nada. Pode fazer o que quisermos para fazer.

—Ada Lovelace, outubro de 1842



Antes de poder escrever programas de criptografia, você precisa aprender alguns conceitos básicos de programação. Esses conceitos incluem valores, operadores, expressões e variáveis.

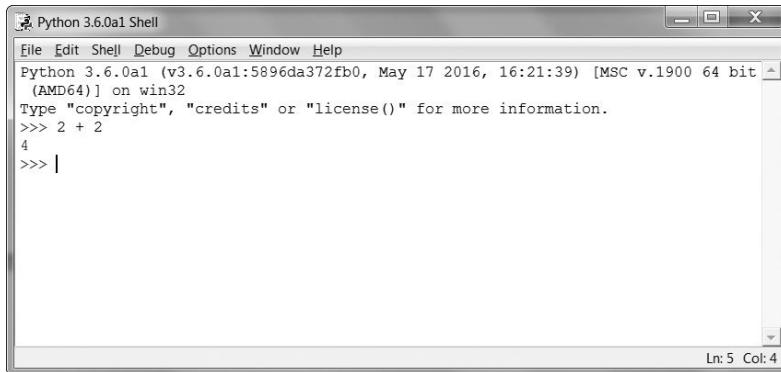
TÓPICOS ABORDADOS NESTE CAPÍTULO

- Operadores
- Valores
- Inteiros e números de ponto flutuante
- Expressões
- Avaliando expressões
- Armazenando valores em variáveis
- Sobrescrevendo variáveis

Vamos começar explorando como fazer uma matemática simples no shell interativo do Python. Certifique-se de ler este livro ao lado do seu computador para que você possa inserir os exemplos de código curto e ver o que eles fazem. Desenvolver memória muscular a partir de programas de digitação irá ajudá-lo a lembrar como o código Python é construído.

Algumas expressões matemáticas simples

Comece abrindo o IDLE (veja “ [Iniciando o IDLE](#) ” na [página xxvii](#)). Você verá o shell interativo e o cursor piscando próximo ao prompt `>>>` . O shell interativo pode funcionar como uma calculadora. Digite `2 + 2` no shell e pressione `enter` no seu teclado. (Em alguns teclados, essa é a tecla de `retorno` .) O computador deve responder exibindo o número `4` , conforme mostrado na [Figura 2-1](#) .



A screenshot of the Python 3.6.0a1 Shell window. The title bar says "Python 3.6.0a1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the Python interpreter's prompt: >>>. Below it, the user typed "2 + 2" and the interpreter responded with "4". At the bottom right of the window, there is a status bar with "Ln: 5 Col: 4".

Figura 2-1: Digite $2 + 2$ no shell.

No exemplo da [Figura 2-1](#), o sinal `+` diz ao computador para adicionar os números `2` e `2`, mas o Python também pode fazer outros cálculos, como subtrair números usando o sinal de menos (`-`), multiplicar números por um asterisco (`*`), ou dividir números com uma barra (`/`). Quando usados dessa maneira, os *operadores* `+`, `-`, `*` e `/` são chamados porque dizem ao computador para executar uma operação nos números que os cercam. [A Tabela 2-1](#) resume os operadores matemáticos do Python. Os `2`s (ou outros números) são chamados de *valores*.

Tabela 2-1: Operadores de matemática em Python

Operador Operação

<code>+</code>	Adição
<code>-</code>	Subtração
<code>*</code>	Multiplicação
<code>/</code>	Divisão

Por si só, $2 + 2$ não é um programa; é apenas uma única instrução. Programas são feitos de muitas dessas instruções.

Valores inteiros e valores de ponto flutuante

Na programação, números inteiros, como `4`, `0` e `99`, são chamados *inteiros*. Números com pontos decimais (`3.5`, `42.1` e `5.0`) são chamados de *números de ponto flutuante*. Em Python, o número `5` é um inteiro, mas se você o escreveu como `5.0`, seria um número de ponto flutuante.

Inteiros e pontos flutuantes são *tipos de dados*. O valor `42` é um valor do tipo de

dados inteiro ou *int* . O valor 7.5 é um valor do tipo de dados ponto flutuante ou *flutuante* .

Todo valor tem um tipo de dados. Você aprenderá sobre alguns outros tipos de dados (como strings no [Capítulo 3](#)), mas por enquanto lembre-se de que sempre que falamos sobre um valor, esse valor é de um certo tipo de dado. Geralmente é fácil identificar o tipo de dados apenas observando como o valor é escrito. Ints são números sem pontos decimais. Flutuadores são números com pontos decimais. Então, 42 é um int, mas 42.0 é um float.

Expressões

Você já viu o Python resolver um problema de matemática, mas o Python pode fazer muito mais. Tente digitar os seguintes problemas de matemática no shell, pressionando a tecla `enter` após cada um deles:

```
❶ >>> 2 + 2 + 2 + 2 + 2  
10  
>>> 8 * 6  
48  
❷ >>> 10-5 + 6  
11  
❸ >>> 2 + 2  
4
```

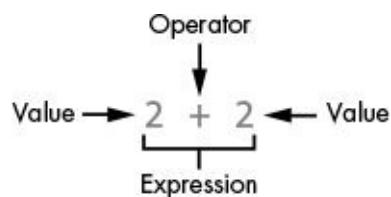


Figura 2-2: Uma expressão é composta de valores (como 2) e operadores (como +).

Esses problemas matemáticos são chamados de *expressões* . Computadores podem resolver milhões desses problemas em segundos. As expressões são compostas de valores (os números) conectados pelos operadores (os sinais de matemática), conforme mostrado na [Figura 2-2](#) . Você pode ter quantos números em uma expressão desejar ❶ , desde que eles estejam conectados por operadores; você pode até usar vários tipos de operadores em uma única expressão ❷ . Você também pode inserir qualquer número de espaços entre os inteiros e esses operadores ❸ . Mas não se esqueça de sempre iniciar uma

expressão no início da linha, sem espaços à frente, porque os espaços no início de uma linha alteram a maneira como o Python interpreta as instruções. Você aprenderá mais sobre espaços no início de uma linha em “ [Blocos](#) ” na [página 45](#) .

Ordem de operações

Você pode se lembrar da frase “ordem de operações” da sua aula de matemática. Por exemplo, a multiplicação é feita antes da adição. A expressão $2 + 4 * 3$ é avaliada como 14 porque a multiplicação é feita primeiro para avaliar $4 * 3$ e, em seguida, 2 é adicionado. Os parênteses podem fazer com que diferentes operadores sejam os primeiros. Na expressão $(2 + 4) * 3$, a adição é feita primeiro para avaliar $(2 + 4)$ e, em seguida, essa soma é multiplicada por 3. Os parênteses fazem a expressão avaliar em 18 em vez de 14. A ordem de operações (também chamada de *precedência*) dos operadores matemáticos de Python é semelhante à matemática. Operações dentro de parênteses são avaliadas primeiro; em seguida, os operadores * e / são avaliados da esquerda para a direita; e então os operadores + e - são avaliados da esquerda para a direita.

Avaliando Expressões

Quando um computador resolve a expressão $10 + 5$ e obtém o valor 15, dizemos que *avaliou* a expressão. Avaliar uma expressão reduz a expressão a um único valor, assim como resolver um problema matemático reduz o problema a um único número: a resposta.

As expressões $10 + 5$ e $10 + 3 + 2$ têm o mesmo valor, porque ambas são avaliadas como 15. Mesmo valores únicos são considerados expressões: a expressão 15 avalia o valor 15.

O Python continua a avaliar uma expressão até se tornar um valor único, como no seguinte:

$$\begin{array}{c} (5 - 1) * ((7 + 1) / (3 - 1)) \\ \downarrow \\ 4 * ((7 + 1) / (3 - 1)) \\ \downarrow \\ 4 * ((\underline{\quad 8 \quad}) / (\underline{\quad 3 - 1})) \\ \downarrow \\ 4 * ((\underline{\quad 8 \quad}) / (\underline{\quad 2 \quad})) \\ \downarrow \\ 4 * 4.0 \\ \hline 16.0 \end{array}$$

O Python avalia uma expressão que começa com os parênteses mais internos e mais à esquerda. Mesmo quando os parênteses estão aninhados uns nos outros, as partes das expressões dentro delas são avaliadas com as mesmas regras que qualquer outra expressão. Portanto, quando o Python encontra $((7 + 1) / (3 - 1))$, ele primeiro resolve a expressão nos parênteses internos mais à esquerda, $(7 + 1)$ e, em seguida, resolve a expressão à direita, $(3 - 1)$. Quando cada expressão nos parênteses internos é reduzida a um único valor, as expressões nos parênteses externos são avaliadas. Observe que a divisão é avaliada como um valor de ponto flutuante. Finalmente, quando não há mais expressões entre parênteses, o Python executa os cálculos restantes na ordem das operações.

Em uma expressão, você pode ter dois ou mais valores conectados por operadores, ou você pode ter apenas um valor, mas se você inserir um valor e um operador no shell interativo, receberá uma mensagem de erro:

```
>>> 5 +  
SyntaxError: sintaxe inválida
```

Este erro acontece porque $5 +$ não é uma expressão. Expressões com vários valores precisam que os operadores conectem esses valores e, na linguagem Python, o operador $+$ espera conectar dois valores. Um *erro de sintaxe* significa que o computador não entende a instrução que você deu porque você digitou incorretamente. Isso pode não parecer importante, mas a programação de computadores não significa apenas informar ao computador o que fazer - também é saber o modo correto de fornecer as instruções do computador que ele pode seguir.

ERROS ESTÃO OK!

É perfeitamente correto cometer erros! Você não vai quebrar o seu computador, digitando o código que causa erros. O Python simplesmente informará que ocorreu um erro e exibirá o prompt `>>>` novamente. Você pode continuar inserindo novo código no shell interativo.

Até você ganhar mais experiência em programação, as mensagens de erro podem não fazer muito sentido para você. No entanto, você sempre pode pesquisar no google o texto da mensagem de erro para encontrar páginas da web que explicam esse erro específico. Você também pode acessar <https://www.nostarch.com/crackingcodes/> para ver uma lista de mensagens de erro comuns do Python e seus significados.

Armazenando Valores com Variáveis

Os programas geralmente precisam salvar valores para usar posteriormente no programa. Você pode armazenar valores em *variáveis* usando o sinal = (chamado *operador de atribuição*). Por exemplo, para armazenar o valor 15 em uma variável chamada spam , insira spam = 15 no shell:

```
>>> spam = 15
```

Você pode pensar na variável como uma caixa com o valor 15 dentro dela (como mostrado na [Figura 2-3](#)). O nome da variável spam é o rótulo na caixa (para que possamos distinguir uma variável de outra), e o valor armazenado nela é como uma nota dentro da caixa.

Quando você pressiona enter , você não verá nada a não ser uma linha em branco em resposta. A menos que você veja uma mensagem de erro, você pode assumir que a instrução foi executada com sucesso. O próximo prompt >>> aparece para que você possa inserir a próxima instrução.

Esta instrução com o operador = atribuição (chamado de *declaração de atribuição*) cria a variável spam e armazena o valor 15 nela. Ao contrário das expressões, as *instruções* são instruções que não avaliam nenhum valor; em vez disso, eles apenas executam uma ação. É por isso que nenhum valor é exibido na próxima linha do shell.

Descobrir quais instruções são expressões e quais são instruções podem ser confusas. Apenas lembre-se de que, se uma instrução Python for avaliada como um único valor, será uma expressão. Se isso não acontecer, é uma afirmação.

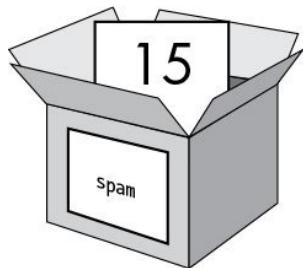


Figura 2-3: Variáveis são como caixas com nomes que podem conter valor.

Uma declaração de atribuição é escrita como uma variável, seguida pelo operador = , seguido por uma expressão, como mostrado na [Figura 2-4](#) . O valor que a expressão avalia é armazenado dentro da variável.

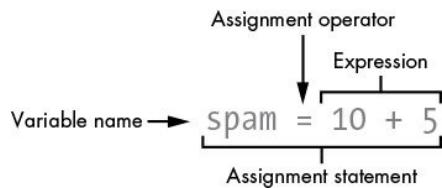


Figura 2-4: As partes de uma declaração de atribuição

Tenha em mente que as variáveis armazenam valores únicos, não as expressões que são atribuídas. Por exemplo, se você digitar a instrução `spam = 10 + 5`, a expressão $10 + 5$ será avaliada primeiro como 15 e, em seguida, o valor 15 será armazenado na variável `spam`, como podemos ver inserindo o nome da variável no shell:

```
>>> spam = 10 + 5
>>> spam
15
```

Uma variável por si só é uma expressão que avalia o valor armazenado na variável. Um valor por si só também é uma expressão que se avalia:

```
>>> 15
15
```

E aqui está uma reviravolta interessante. Se você agora digitar `spam + 5` no shell, você obterá o inteiro 20 :

```
>>> spam = 15
>>> spam + 5
20
```

Como você pode ver, as variáveis podem ser usadas em expressões da mesma maneira que os valores podem. Como o valor do `spam` é 15 , a expressão `spam + 5` avalia a expressão $15 + 5$, que então é avaliada como 20 .

Sobrescrevendo Variáveis

Você pode alterar o valor armazenado em uma variável, inserindo outra declaração de atribuição. Por exemplo, digite o seguinte:

```
>>> spam = 15
❶ >>> spam + 5
❷ 20
❸ >>> spam = 3
❹ >>> spam + 5
```

5 8

Na primeira vez que você inserir `spam + 5` ❶, a expressão será avaliada como 20 ❷ porque você armazenou o valor 15 dentro da variável `spam`. Mas quando você insere `spam = 3` ❸, o valor 15 é *sobreescrito* (ou seja, substituído) pelo valor 3, conforme mostrado na [Figura 2-5](#). Agora, quando você insere `spam + 5` ❹, a expressão é avaliada como 8 ❺ porque o `spam + 5` é avaliado como `3 + 5`. O valor antigo no `spam` é esquecido.

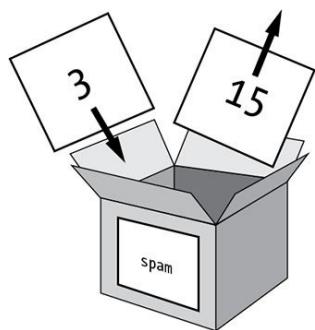


Figura 2-5: O valor 15 no spam é sobreposto pelo valor 3.

Você pode até usar o valor na variável `spam` para atribuir um novo valor ao `spam`:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam
20
```

A instrução de atribuição `spam = spam + 5` informa ao computador que “o novo valor da variável de `spam` é o valor atual de `spam` mais cinco”. A variável à esquerda do sinal `=` é atribuída ao valor da expressão à direita lado. Você pode continuar aumentando o valor do `spam` em 5 várias vezes:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
```

O valor do `spam` é alterado sempre que `spam = spam + 5` é executado. O valor armazenado no `spam` acaba sendo 30.

Nomes Variáveis

Embora o computador não se importe com o que você nomeia suas variáveis, você deveria. Dar nomes de variáveis que refletem o tipo de dados que eles contêm facilita a compreensão do que um programa faz. Você poderia dar nomes às suas variáveis como abrahamLincoln ou monkey, mesmo que seu programa não tivesse nada a ver com Abraham Lincoln ou com macacos - o computador ainda rodaria o programa (contanto que você usasse Abraham Lincoln ou macaco). Mas quando você retorna a um programa depois de não vê-lo por um longo tempo, você pode não se lembrar do que cada variável faz.

Um bom nome de variável descreve os dados que ele contém. Imagine que você se mudou para uma nova casa e rotulou todas as suas caixas móveis. Você nunca encontraria nada! Os nomes de variáveis spam , ovos , bacon e assim por diante (inspirados no esboço “Spam” do *Monty Python*) são usados como nomes genéricos para os exemplos deste livro e em grande parte da documentação do Python, mas em seus programas, um nome descritivo ajuda Torne seu código mais legível.

Nomes de variáveis (assim como tudo o mais em Python) diferenciam maiúsculas de minúsculas. *Caso sensível* significa que o mesmo nome de variável em um caso diferente é considerado uma variável totalmente diferente. Por exemplo, spam , spam , spam e sPAM são considerados quatro variáveis diferentes no Python. Cada um deles pode conter seus próprios valores separados e não pode ser usado de forma intercambiável.

Resumo

Então, quando vamos começar a fazer programas de criptografia? Em breve. Mas antes que você possa hackear cifras, você precisa aprender apenas alguns conceitos básicos de programação, então há mais um capítulo de programação que você precisa ler.

Neste capítulo, você aprendeu o básico de escrever instruções em Python no shell interativo. O Python precisa que você diga exatamente o que fazer da maneira esperada, porque os computadores só entendem instruções muito simples. Você aprendeu que o Python pode avaliar expressões (ou seja, reduzir a expressão a um único valor) e que expressões são valores (como 2 ou 5) combinados com operadores (como + ou -). Você também aprendeu que pode armazenar valores dentro de variáveis para que seu programa possa se lembrar deles para usar mais tarde.

O shell interativo é uma ferramenta útil para aprender o que as instruções do

Python fazem, pois permite que você as insira uma por vez e veja os resultados. No [Capítulo 3](#), você criará programas que contêm muitas instruções executadas em sequência, em vez de uma por vez. Vamos discutir alguns conceitos mais básicos e você vai escrever seu primeiro programa!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Qual é o operador para divisão, / ou \ ?
2. Qual dos seguintes é um valor inteiro e qual é um valor de ponto flutuante?

42

3,141592

3. Qual das seguintes linhas *não* são expressões?

4 x 10 + 2

3 * 7 + 1

2 +

42

2 + 2

spam = 42

4. Se você inserir as seguintes linhas de código no shell interativo, o que as linhas ❶ e ❷ imprimem?

spam = 20

❶ spam + 20

SPAM = 30

❶ spam

3

CORDAS E PROGRAMAS DE ESCRITA

“A única maneira de aprender uma nova linguagem de programação é escrevendo programas nela.”

- Brian Kernighan e Dennis Ritchie,

A linguagem de programação C



[O Capítulo 2](#) deu-lhe números inteiros e matemática suficientes por enquanto. O Python é mais do que apenas uma calculadora. Como a criptografia é toda sobre como lidar com valores de texto transformando texto simples em texto cifrado e vice-versa, você aprenderá a armazenar, combinar e exibir texto na tela neste capítulo. Você também fará seu primeiro programa, que cumprimenta o usuário com o texto “Hello, world!” E permite que o usuário insira seu nome.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Cordas
- Concatenação e replicação de strings
- Índices e fatias
- A função `print()`
- Escrevendo código fonte com IDLE
- Salvando e executando programas no IDLE
- Comentários
- A função `input()`

Trabalhando com texto usando valores de seqüência de caracteres

Em Python, trabalhamos com pequenos trechos de texto chamados valores de string (ou simplesmente *strings*). Todos os nossos programas de cifra e hacking lidam com valores de strings para transformar texto plano como 'Um se por terra, dois por espaço' em texto cifrado como ' b1rJvsJo ! Jyn1q, J702JvsJo! J63nprM' . O texto simples e o texto cifrado são representados em nosso programa como valores de string, e há muitas maneiras nas quais o código Python pode manipular esses valores.

Você pode armazenar valores de string dentro de variáveis, assim como você pode com valores inteiros e de ponto flutuante. Quando você digita uma string, coloque-a entre duas aspas simples (') para mostrar onde a string começa e

termina. Digite o seguinte no shell interativo:

```
>>> spam = 'olá'
```

As aspas simples não fazem parte do valor da cadeia. Python sabe que 'hello' é uma string e o spam é uma variável porque as strings são cercadas por aspas e os nomes das variáveis não.

Se você inserir spam no shell, verá o conteúdo da variável de spam (a string 'hello'):

```
>>> spam = 'olá'  
>>> spam  
'Olá'
```

Isso ocorre porque o Python avalia uma variável para o valor armazenado dentro dela: nesse caso, a string 'hello' . As cordas podem ter quase qualquer caractere de teclado nelas. Estes são todos exemplos de strings:

```
>>> 'olá'  
'Olá'  
>>> 'GATINHOS'  
'GATINHOS'  
>>> "  
"  
>>> '7 maçãs, 14 laranjas, 3 limões'  
'7 maçãs, 14 laranjas, 3 limões'  
>>> 'Qualquer coisa não pertencente a elefantes é irrelephant.'  
"Qualquer coisa que não seja de elefantes é irrelevant."  
>>> 'O * # wY% * & OcfsdYO * & gfC% YO * &% 3yc8r2'  
'O * # wY% * & OcfsdYO * & gfC% YO * &% 3yc8r2'
```

Observe que a string " tem zero caracteres; Não há nada entre aspas simples. Isso é conhecido como uma *string em branco* ou uma *string vazia* .

Concatenação de Cordas com o Operador +

Você pode adicionar dois valores de string para criar uma nova string usando o operador + . Isso é chamado de *concatenação de string* . Digite "Olá" + "mundo"! na casca:

```
>>> 'Olá,' + 'mundo!'  
'Olá Mundo!'
```

O Python concatena *exatamente* as strings que você diz para concatenar, por isso não colocará um espaço entre as strings quando você concatená-las. Se você quiser um espaço na string resultante, deve haver um espaço em uma das duas strings originais. Para colocar um espaço entre "Olá" e "mundo!" , você pode colocar um espaço no final da string 'Hello' e antes da segunda aspa simples, assim:

```
>>> 'Olá,' + 'mundo!'
'Olá Mundo!'
```

O operador + pode concatenar dois valores de string em um novo valor de string ('Hello,' + 'world!' Para 'Hello, world!'), Assim como pode adicionar dois valores inteiros para resultar em um novo valor inteiro (2 + 2 a 4). O Python sabe o que o operador + deve fazer por causa dos tipos de dados dos valores. Como você aprendeu no [Capítulo 2](#) , o tipo de dados de um valor nos diz (e o computador) que tipo de dados o valor é.

Você pode usar o operador + em uma expressão com duas ou mais cadeias de caracteres ou números inteiros, desde que os tipos de dados correspondam. Se você tentar usar o operador com uma string e um inteiro, você receberá um erro. Digite este código no shell interativo:

```
>>> 'Olá' + 42
Traceback (última chamada mais recente):
Arquivo "<stdin>", linha 1, em <module>
TypeError: deve ser str, não int
>>> 'Olá' + '42'
'Hello42'
```

A primeira linha de código causa um erro porque 'Hello' é uma string e 42 é um inteiro. Mas na segunda linha de código, '42' é uma string, então o Python a concatena.

Replicação de Cadeia com o Operador *

Você também pode usar o operador * em uma string e um inteiro para fazer a *replicação de string* . Isso replica (isto é, repete) uma string por muitas vezes o valor inteiro é. Digite o seguinte no shell interativo:

```
❶ >>> 'Olá' * 3
'Ola Ola Ola'
>>> spam = 'Abcdef'
```

```
❷ >>> spam = spam * 3  
>>> spam  
'AbcdefAbcdefAbcdef'
```

Para replicar uma string, digite a string, depois o operador * e, em seguida, o número de vezes que você deseja que a string se repita ❶ . Você também pode armazenar uma string, como fizemos com a variável spam , e depois replicar a variável ❷ . Você pode até mesmo armazenar uma string replicada de volta na mesma variável ou em uma nova variável.

Como você viu no [Capítulo 2](#) , o operador * pode trabalhar com dois valores inteiros para multiplicá-los. Mas não pode trabalhar com dois valores de string, o que causaria um erro, como este:

```
>>> 'Olá' * 'mundo!'
```

Traceback (última chamada mais recente):

Arquivo "<stdin>", linha 1, em <module>

TypeError: não é possível multiplicar a sequência por não int do tipo 'str'

A concatenação de strings e a replicação de strings mostram que os operadores no Python podem executar tarefas diferentes com base nos tipos de dados dos valores nos quais operam. O operador + pode fazer a adição ou a concatenação de strings. O operador * pode fazer multiplicação ou replicação de string.

Obtendo Caracteres de Strings Usando Índices

Seus programas de criptografia geralmente precisam obter um único caractere de uma string, que você pode realizar por meio da indexação. Com a *indexação* , você adiciona colchetes [e] ao final de um valor de string (ou uma variável contendo uma string) com um número entre eles para acessar um caractere. Esse número é chamado de *índice* e diz ao Python qual posição na string tem o caractere desejado. Os índices do Python começam em 0 , portanto, o índice do primeiro caractere em uma string é 0 . O índice 1 é para o segundo caractere, o índice 2 é para o terceiro caractere e assim por diante.

Digite o seguinte no shell interativo:

```
>>> spam = 'Olá'  
>>> spam [0]  
'O'  
>>> spam [1]  
'l'
```

```
>>> spam [2]  
'eu'  
  
string: ' H e l l o '  
indexes: 0 1 2 3 4
```

Figura 3-1: A string 'Hello' e seus índices

Observe que a expressão `spam [0]` é avaliada como o valor da string 'H' , porque H é o primeiro caractere na string 'Hello' e os índices começam em 0 , não 1 (veja a [Figura 3-1](#)).

Você pode usar a indexação com uma variável contendo um valor de string, como fizemos com o exemplo anterior, ou um valor de string sozinho, como este:

```
>>> 'Zophie' [2]  
'p'
```

A expressão `'Zophie' [2]` avalia o terceiro valor de string, que é um 'p' . Esta string 'p' é igual a qualquer outro valor de string e pode ser armazenada em uma variável. Digite o seguinte no shell interativo:

```
>>> ovos = 'Zophie' [2]  
>>> ovos  
'p'
```

Se você inserir um índice que seja muito grande para a sequência, o Python exibirá uma mensagem de erro "índice fora do intervalo" , como você pode ver no código a seguir:

```
>>> 'Olá' [10]  
Traceback (última chamada mais recente):  
Arquivo "<stdin>", linha 1, em <module>  
IndexError: índice de string fora do intervalo
```

Existem cinco caracteres na string 'Hello' , portanto, se você tentar usar o índice 10 , o Python exibirá um erro.

```
string: ' H e l l o '  
indexes: -5 -4 -3 -2 -1
```

Figura 3-2: A string 'Hello' e seus índices negativos

Índices negativos

Índices negativos começam no final de uma string e vão para trás. O índice

negativo -1 é o índice do *último* caractere em uma string. O índice -2 é o índice do segundo ao último caractere e assim por diante, conforme mostrado na [Figura 3-2](#).

Digite o seguinte no shell interativo:

```
>>> 'Olá' [- 1]  
'o'  
>>> 'Olá' [- 2]  
'eu'  
>>> 'Olá' [- 3]  
'eu'  
>>> 'Olá' [- 4]  
'e'  
>>> 'Olá' [- 5]  
'H'  
>>> 'Olá' [0]  
'H'
```

Observe que -5 e 0 são os índices para o mesmo caractere. Na maioria das vezes, seu código usa índices positivos, mas às vezes é mais fácil usar os negativos.

Obtendo vários caracteres de seqüências de caracteres usando fatias

Se você quiser obter mais de um caractere de uma string, use o fatiamento em vez da indexação. Uma *fatia* também usa os colchetes [e], mas tem dois índices inteiros em vez de um. Os dois índices são separados por dois pontos (:) e informam ao Python o índice do primeiro e último caracteres da fatia. Digite o seguinte no shell interativo:

```
>>> 'Howdy' [0: 3]  
'Como'
```

A cadeia que a fatia avalia começa no primeiro valor de índice e sobe, mas não inclui, o segundo valor de índice. O índice 0 do valor da string 'Howdy' é H e o índice 3 é d . Como uma fatia sobe mas não inclui o segundo índice, a fatia "Howdy" [0: 3] é avaliada como "How" .

Digite o seguinte no shell interativo:

```
>>> 'Olá, mundo!' [0: 5]  
'Olá'  
>>> 'Olá, mundo!' [7:13]
```

```
'mundo!'
>>> 'Olá, mundo!' [- 6: -1]
'mundo'
>>> 'Olá, mundo!' [7:13] [2]
'r'
```

Observe que a expressão 'Hello, world!' [7:13] [2] primeiro avalia a lista de segmentos como 'world!' [2] e depois avalia como 'r' .

Ao contrário dos índices, o fatiamento nunca gera um erro se você der um índice muito grande para a string. Ele apenas retornará a fatia correspondente mais ampla possível:

```
>>> 'Olá' [0: 999]
'Olá'
>>> 'Olá' [2: 999]
'llo'
>>> 'Olá' [1000: 2000]
"
```

A expressão 'Hello' [1000: 2000] retorna uma string em branco porque o índice 1000 é após o final da string, portanto, não há caracteres possíveis que essa fatia possa incluir. Embora nossos exemplos não mostrem isso, você também pode cortar strings armazenadas em variáveis.

Índices de fatia em branco

Se você omitir o primeiro índice de uma fatia, o Python usará automaticamente o índice 0 para o primeiro índice. As expressões 'Howdy' [0: 3] e 'Howdy' [: 3] avaliam a mesma string:

```
>>> 'Howdy' [: 3]
'Como'
>>> 'Howdy' [0: 3]
'Como'
```

Se você omitir o segundo índice, o Python usará automaticamente o restante da string a partir do primeiro índice:

```
>>> 'Howdy' [2:]
'wdy'
```

Você pode usar índices em branco de várias maneiras diferentes. Digite o seguinte no shell:

```
>>> myName = 'Zophie the Fat Cat'  
>>> myName [-7:]  
'Gato gordo'  
>>> myName [: 10]  
'Zophie the'  
>>> myName [7:]  
'o gato gordo'
```

Como você pode ver, você pode até usar índices negativos com um índice em branco. Como -7 é o índice inicial no primeiro exemplo, o Python conta sete caracteres para trás do final e usa isso como seu índice inicial. Em seguida, ele retorna tudo desse índice para o final da string devido ao segundo índice em branco.

Imprimindo valores com a função print ()

Vamos tentar outro tipo de instrução em Python: uma chamada de função print () . Digite o seguinte no shell interativo:

```
>>> print ('Olá!')  
Olá!  
>>> imprimir (42)  
42
```

Uma *função* (como print () neste exemplo) possui um código dentro dela que executa uma tarefa, como imprimir valores na tela. Muitas funções diferentes vêm com o Python e podem executar tarefas úteis para você. *Chamar* uma função significa executar o código dentro da função.

As instruções neste exemplo passam um valor para imprimir () entre parênteses e a função print () imprime o valor na tela. Os valores que são passados quando uma função é chamada são *argumentos* . Quando você escreve programas, você usará print () para fazer o texto aparecer na tela.

Você pode passar uma expressão para print () em vez de um único valor. Isso ocorre porque o valor que é realmente passado para print () é o valor avaliado dessa expressão. Digite esta expressão de concatenação de seqüência de caracteres no shell interativo:

```
>>> spam = 'Al'  
>>> print ('Olá', + spam)  
Olá Al
```

A expressão "Hello", + spam é avaliada como "Hello" e "Al" , que é avaliada como o valor da string "Hello, Al" . Este valor de string é o que é passado para a chamada print () .

Impressão de personagens de fuga

Você pode querer usar um caractere em um valor de string que confunda Python. Por exemplo, você pode querer usar um caractere de aspas simples como parte de uma string. Mas você receberia uma mensagem de erro porque o Python acha que a aspa simples é a citação que termina o valor da string e o texto após ele é um código Python incorreto, em vez do resto da string. Digite o seguinte no shell interativo para ver o erro em ação:

```
>>> print ('o gato de Al é chamado Zophie.')
SyntaxError: sintaxe inválida
```

Para usar uma aspa simples em uma string, você precisa usar um *caractere de escape* . Um caractere de escape é um caractere de barra invertida seguido por outro caractere - por exemplo, \ t , \ n ou \ ' . A barra diz ao Python que o caractere após a barra tem um significado especial. Digite o seguinte no shell interativo.

```
>>> print ('gato do Al é chamado Zophie.')
O gato de Al é chamado Zophie.
```

Agora o Python saberá que o apóstrofo é um caractere no valor da string, e não o código Python que marca o final da string.

[A Tabela 3-1](#) mostra alguns caracteres de escape que você pode usar no Python.

Tabela 3-1: Personagens de Escape

Personagem de fuga	Resultado impresso
\\	Barra invertida (\)
\'	Citação simples (')
\"	Aspas duplas (")

\n Nova linha

\t Aba

A barra invertida sempre precede um caractere de escape. Mesmo que você queira apenas uma barra invertida na sua string, não é possível adicionar uma barra invertida sozinha, porque o Python interpretará o próximo caractere como um caractere de escape. Por exemplo, esta linha de código não funcionaria corretamente:

```
>>> print ('É uma cor verde \ azul.')
É uma cor verde eal.
```

O 't' em 'teal' é identificado como um caractere de escape porque vem depois de uma barra invertida. O caractere de escape \ t simula o pressionamento da tecla tab no seu teclado.

Em vez disso, insira este código:

```
>>> print ('É uma cor verde \\ azul-petróleo.')
É uma cor verde \ azul.
```

Desta vez, a cadeia será impressa como você pretendia, porque colocar uma segunda barra invertida na cadeia torna a barra invertida o caractere de escape.

Cotações e Cotações Duplas

As strings nem sempre precisam estar entre duas aspas simples no Python. Você pode usar aspas duplas. Essas duas linhas imprimem a mesma coisa:

```
>>> print ('Olá, mundo!')
Olá Mundo!
>>> print ("Olá, mundo!")
Olá Mundo!
```

Mas você não pode misturar aspas simples e duplas. Esta linha dá um erro:

```
>>> print ('Olá, mundo! ')
SyntaxError: EOL durante a varredura de literal de string
```

Eu prefiro usar aspas simples porque elas são um pouco mais fáceis de digitar do que aspas duplas e o Python não se importa de nenhuma maneira.

Mas assim como você tem que usar o caractere de escape para ter uma aspa simples em uma string cercada por aspas simples, você precisa do caractere de

escape para ter aspas duplas em uma string entre aspas duplas. Por exemplo, veja estas duas linhas:

```
>>> print ('gato do Al é Zophie. Ela diz:' Miau '.')
O gato de Al é Zophie. Ela diz: "Miau".
>>> print ("Zophie disse," eu posso dizer coisas além de 'Meow' você sabe. "
Zophie disse: "Eu posso dizer coisas além de 'Meow' você sabe".
```

Você não precisa escapar de aspas duplas em cadeias de caracteres de aspas simples e não precisa escapar de aspas simples nas cadeias de caracteres de aspas duplas. O interpretador Python é esperto o suficiente para saber que, se uma string começar com um tipo de citação, o outro tipo de citação não significa que a string está terminando.

Escrevendo programas no editor de arquivos do IDLE

Até agora, você entrou nas instruções uma de cada vez no shell interativo. Mas quando você escreve programas, você vai inserir várias instruções e executá-las sem esperar pelo próximo. Está na hora de escrever seu primeiro programa!

O nome do programa de software que fornece o shell interativo é chamado de IDLE (Ambiente de Integração de Dados de Integração). Além do shell interativo, o IDLE também possui um *editor de arquivos*, que abriremos agora.

Na parte superior da janela do shell Python, selecione **Arquivo ▶ Nova Janela**. Uma nova janela em branco, o editor de arquivos, aparecerá para você inserir um programa, conforme mostrado na [Figura 3-3](#). O canto inferior direito da janela do editor de arquivos mostra em qual linha e coluna o cursor está atualmente.

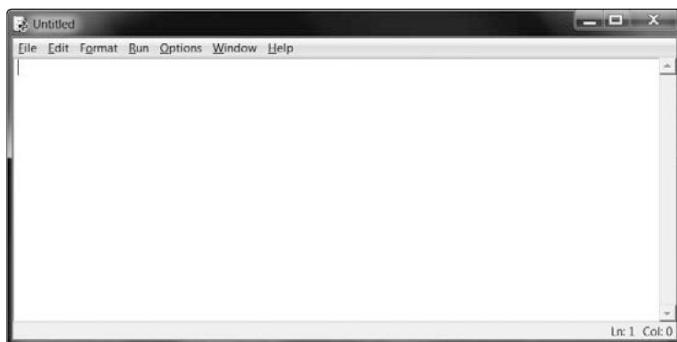


Figura 3-3: A janela do editor de arquivos com o cursor na linha 1, coluna 0

Você pode dizer a diferença entre a janela do editor de arquivos e a janela do shell interativo procurando o prompt `>>>`. O shell interativo sempre exibe o prompt e o editor de arquivo não.

Código fonte para o programa “Hello, World!”

Tradicionalmente, programadores que estão aprendendo uma nova linguagem fazem seu primeiro programa exibir o texto "Hello, world!" na tela. Vamos criar nosso próprio programa "Hello, world!", Digitando o texto na nova janela do editor de arquivos. Chamamos este texto de *código-fonte* do programa porque ele contém as instruções que o Python seguirá para determinar exatamente como o programa deve se comportar.

Você pode baixar o código-fonte "Hello, world!" De <https://www.nostarch.com/crackingcodes/>. Se você obtiver erros depois de inserir este código, compare-o ao código do livro usando a ferramenta de comparação online (consulte “[Verificando seu código-fonte com a ferramenta de comparação online](#)” a seguir). Lembre-se de que você não digita os números de linha; eles só aparecem neste livro para auxiliar a explicação.

ola.py

1. # Este programa diz olá e pede meu nome.
2. print ('Olá, mundo!')
3. print ('Qual é o seu nome?')
4. myName = input ()
5. print ('É bom conhecê-lo,' + myName)

O programa IDLE exibirá diferentes tipos de instruções em cores diferentes. Quando você terminar de digitar este código, a janela deve se parecer com a [Figura 3-4](#).

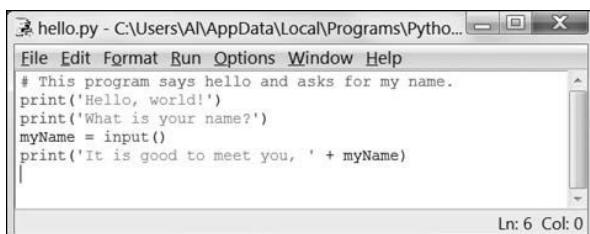


Figura 3-4: A janela do editor de arquivos ficará assim depois que você inserir o código.

Verificando seu código-fonte com a ferramenta online Diff

Mesmo que você possa copiar e colar ou baixar o código *hello.py* do site deste livro, você ainda deve digitar este programa manualmente. Fazer isso lhe dará mais familiaridade com o código no programa. No entanto, você pode cometer

alguns erros ao digitá-lo no editor de arquivos.

Para comparar o código que você digitou no código deste livro, use a ferramenta de comparação online mostrada na [Figura 3-5](#). Copie o texto do seu código e, em seguida, navegue até a ferramenta de comparação no site do livro em <https://www.nostarch.com/crackingcodes/>. Selecione o programa *hello.py* no menu suspenso. Cole seu código no campo de texto nesta página da Web e clique no botão **Comparar**. A ferramenta de comparação mostra as diferenças entre o seu código e o código deste livro. Esta é uma maneira fácil de encontrar erros de digitação causando erros no seu programa.

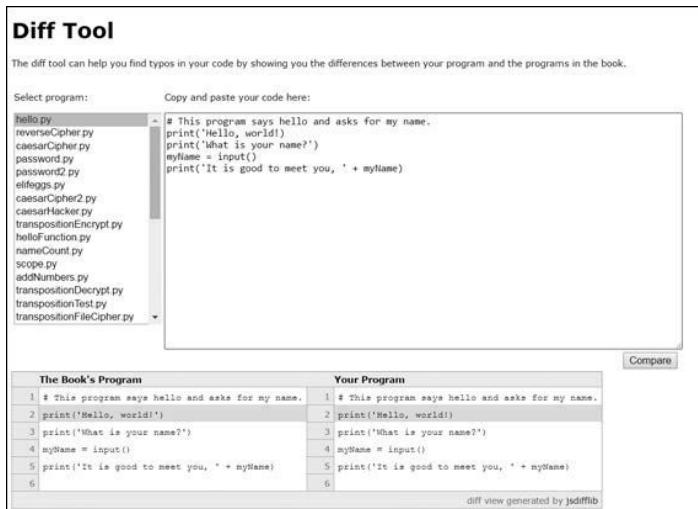


Figura 3-5: A ferramenta de comparação online

Usando o IDLE para acessar seu programa mais tarde

Quando você escreve programas, você pode salvá-los e voltar a eles mais tarde, especialmente depois de digitar um programa muito longo. O IDLE possui recursos para salvar e abrir programas, assim como um processador de texto possui recursos para salvar e reabrir seus documentos.

Salvando seu programa

Depois de inserir seu código-fonte, salve-o para não precisar redigitá-lo toda vez que quiser executá-lo. Escolha **Arquivo ▶ Salvar como** no menu na parte superior da janela do editor de arquivos. A caixa de diálogo Salvar como deve ser aberta, conforme mostrado na [Figura 3-6](#). Digite *hello.py* no campo **Nome do arquivo** e clique em **Salvar**.

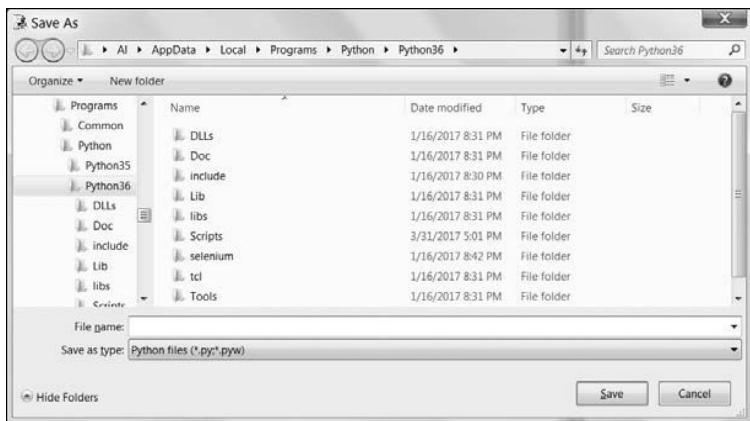


Figura 3-6: Salvando o programa

Você deve salvar seus programas sempre que os digitar, para não perder o trabalho se o computador travar ou se você acidentalmente sair do IDLE. Como atalho, você pode pressionar `ctrl -S` no Windows e no Linux ou `⌘ -S` no macOS para salvar seu arquivo.

Executando seu programa

Agora é hora de executar o seu programa. Selecione **Run ▶ Run Module** ou apenas pressione a tecla F5 no seu teclado. Seu programa deve ser executado na janela do shell que apareceu quando você começou a usar o IDLE. Lembre-se de que você deve pressionar F5 na janela do editor de arquivos, não na janela do shell interativo.

Quando o programa perguntar pelo seu nome, insira-o, conforme mostrado na [Figura 3-7](#).

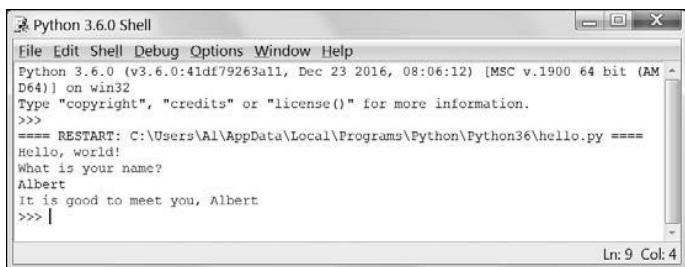


Figura 3-7: O shell interativo fica assim quando o programa “Hello, world!” É executado.

Agora, quando você pressiona `enter`, o programa deve cumprimentá-lo (o *usuário*, ou seja, aquele que usa o programa) pelo nome. Parabéns! Você escreveu seu primeiro programa. Você é agora um programador de computador iniciante. (Se desejar, você pode executar este programa novamente pressionando F5)

novamente.)

Se, em vez disso, você receber um erro assim, significa que você está executando o programa com o Python 2 em vez do Python 3:

Olá Mundo!

Qual é o seu nome?

Albert

Traceback (última chamada mais recente):

Arquivo "C:/Python27/hello.py", linha 4, em <module>

myName = input()

Arquivo "<string>", linha 1, em <module>

NameError: o nome 'Albert' não está definido

O erro é causado pela chamada da função `input()`, que se comporta de maneira diferente no Python 2 e 3. Antes de continuar, instale o Python 3 seguindo as instruções em “[Fazendo Download e Instalando o Python](#)” na [página xxv](#).

Abrindo os programas salvos

Feche o editor de arquivos clicando no X no canto superior. Para recarregar um programa salvo, escolha **Arquivo ▶ Abrir** no menu. Faça isso agora e, na janela exibida, escolha *hello.py*. Em seguida, clique no botão **Abrir**. Seu programa *hello.py* deve ser aberto na janela do editor de arquivos.

Como funciona o programa "Hello, World!"

Cada linha do programa "Hello, world!" É uma instrução que informa ao Python exatamente o que fazer. Um programa de computador é muito parecido com uma receita. Faça o primeiro passo primeiro, depois o segundo e assim por diante até chegar ao fim. Quando o programa segue as instruções passo-a-passo, chamamos de *execução do programa* ou apenas a *execução*.

Cada instrução é seguida em seqüência, começando da parte superior do programa e seguindo a lista de instruções. A execução começa na primeira linha de código e depois se move para baixo. Mas a execução também pode pular em vez de ir de cima para baixo; Você descobrirá como fazer isso no [Capítulo 4](#).

Vamos olhar para o programa "Hello, world!", Uma linha de cada vez, para ver o que está fazendo, começando com a linha 1.

Comentários

Qualquer texto após uma *marca de hash* (#) é um comentário:

1. # Este programa diz olá e pede meu nome.

Comentários não são para o computador, mas sim para você, o programador. O computador os ignora. Eles são usados para lembrá-lo do que o programa faz ou para dizer a outras pessoas que podem ver o código do seu código.

Os programadores costumam colocar um comentário no topo do código para dar um título ao programa. O programa IDLE exibe comentários em texto vermelho para ajudá-los a se destacar. Às vezes, os programadores colocam um # na frente de uma linha de código para pular temporariamente enquanto testam um programa. Isso é chamado de *comentar* código, e pode ser útil quando você está tentando descobrir por que um programa não funciona. Você pode remover o # mais tarde quando estiver pronto para colocar a linha de volta.

Imprimindo rotas para o usuário

As próximas duas linhas exibem instruções para o usuário com a função print () . Uma função é como um mini-programa dentro do seu programa. O grande benefício de usar funções é que precisamos apenas saber o que a função faz, e não como ela funciona. Por exemplo, você precisa saber que print () exibe texto na tela, mas você não precisa saber o código exato dentro da função que faz isso. Uma chamada de função é um pedaço de código que informa ao programa para executar o código dentro de uma função.

A linha 2 de *hello.py* é uma chamada para print () (com a string a ser impressa dentro dos parênteses). A linha 3 é outra chamada print () . Desta vez, o programa exibe "Qual é o seu nome?"

2. print ('Olá, mundo!')

3. print ('Qual é o seu nome?')

Adicionamos parênteses ao final dos nomes das funções para deixar claro que estamos nos referindo a uma função chamada print () , não uma variável chamada print . Os parênteses no final da função dizem ao Python que estamos usando uma função, assim como as aspas em torno do número '42' dizem ao Python que estamos usando a string '42' , não o inteiro 42 .

Tomando uma entrada do usuário

A linha 4 tem uma instrução de atribuição com uma variável (myName) e a nova entrada de chamada de função () :

4. myName = input ()

Quando `input()` é chamado, o programa espera que o usuário digite algum texto e pressione `enter`. A string de texto que o usuário digita (seu nome) se torna o valor da string armazenado em `myName`.

Como as expressões, as chamadas de função são avaliadas para um único valor. O valor para o qual a chamada avalia é chamado de *valor de retorno*. (Na verdade, também podemos usar a palavra "retorna" para significar a mesma coisa que "avalia" para chamadas de função.) Nesse caso, o valor de retorno de `input()` é a cadeia que o usuário inseriu, que deve ser sua nome. Se o usuário digitou `Albert`, a chamada `input()` é avaliada como (isto é, retorna) a string '`Albert`'.

Ao contrário de `print()`, a função `input()` não precisa de nenhum argumento, e é por isso que não há nada entre os parênteses.

A última linha do código em `hello.py` é outra chamada `print()`:

5. `print('É bom conhecê-lo,' + myName)`

Para a chamada `print()` da linha 5, usamos o operador plus (`+`) para concatenar a string '`É bom conhecê-lo`' e a string armazenada na variável `myName`, que é o nome que o usuário insere no programa. É assim que recebemos o programa para saudar o usuário pelo nome.

Terminando o programa

Quando o programa executa a última linha, ele para. Neste ponto, ele foi *finalizado* ou *encerrado* e todas as variáveis são esquecidas pelo computador, incluindo a cadeia armazenada em `myName`. Se você tentar executar o programa novamente e digitar um nome diferente, ele imprimirá esse nome.

Olá Mundo!

Qual é o seu nome?

Zophie

É bom conhecer você, Zophie

Lembre-se que o computador só faz exatamente o que você programa para fazer. Neste programa, ele pede o seu nome, permite que você digite uma string e, em seguida, diz olá e exibe a string digitada.

Mas os computadores são burros. O programa não se importa se você digitar seu nome, o nome de outra pessoa ou apenas algo bobo. Você pode digitar o que quiser e o computador tratará da mesma maneira:

Olá Mundo!
Qual é o seu nome?
cocô
É bom conhecer você, cocô

Resumo

Escrever programas é apenas saber falar a linguagem do computador. Você aprendeu um pouco sobre como fazer isso no [Capítulo 2](#) e agora reuniu várias instruções do Python para criar um programa completo que solicita o nome do usuário e cumprimenta esse usuário.

Neste capítulo, você aprendeu várias novas técnicas para manipular strings, como usar o operador + para concatenar strings. Você também pode usar indexação e fatiamento para criar uma nova string a partir de uma string diferente.

O restante dos programas deste livro será mais complexo e sofisticado, mas todos serão explicados linha por linha. Você sempre pode inserir instruções no shell interativo para ver o que elas fazem antes de colocá-las em um programa completo.

Em seguida, começaremos a escrever nosso primeiro programa de criptografia: a cifra reversa.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. Se você atribuir spam = 'Cats' , o que as seguintes linhas imprimem?

spam + spam + spam
spam * 3

2. O que as seguintes linhas imprimem?

```
print ("Querida Alice, \nComo você está? \nSinceramente, \nBob")  
print ('Hello' + 'Hello')
```

3. Se você atribuir spam = 'Quatro pontos e sete anos são oitenta e sete anos'. , o que cada uma das seguintes linhas imprimiria?

imprimir (spam [5])
imprimir (spam [-3])

```
imprimir (spam [0: 4] + spam [5])
imprimir (spam [-3: -1])
imprimir (spam [: 10])
imprimir (spam [-5:])
imprimir (spam [:])
```

4. Qual janela exibe o prompt >>> , o shell interativo ou o editor de arquivos?
5. O que a seguinte linha imprime?

```
#print ('Olá, mundo!')
```

4

A CIGA REVERSA

"Todo homem é cercado por um bairro de espiões voluntários."

- Jane Austen, abadia de Northanger



A criptografia inversa criptografa uma mensagem, imprimindo-a na ordem inversa. Então "Hello, world!" Criptografa para "! Dlrow, olleH". Para descriptografar ou receber a mensagem original, basta inverter a mensagem criptografada. As etapas de criptografia e descriptografia são as mesmas.

No entanto, essa cifra reversa é fraca, facilitando a descoberta do texto simples. Apenas olhando o texto cifrado, você pode descobrir que a mensagem está na ordem inversa.

.syas ti tahw tuo erugif llits ylbaborp nac uoy, assim como a neve, elpmaxe roF

Mas o código para o programa de criptografia reversa é fácil de explicar, então vamos usá-lo como nosso primeiro programa de criptografia.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A função len ()
- enquanto loops

- Tipo de dados booleano
- Operadores de comparação
- Condições
- Blocos

Código-fonte para o programa de codificação reversa

Em IDLE, clique em **File ▶ New Window** para criar uma nova janela do editor de arquivos. Digite o seguinte código, salve-o como *reverseCipher.py* e pressione F5 para executá-lo, mas lembre-se de não digitar os números antes de cada linha:

reverseCipher.py

```

1. Cifra reversa
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. message = 'Três podem manter um segredo, se dois deles estiverem mortos.'
5. traduzido = "
6
7. i = len (mensagem) - 1
8. enquanto i> = 0:
9.   traduzido = traduzido + mensagem [i]
10. i = i - 1
11
12. imprimir (tradução)
```

Exemplo de execução do programa de codificação reversa

Quando você executa o programa *reverseCipher.py*, a saída se parece com isto:

.ed ed era meht para fi, terces um peek nac eerhT

Para descriptografar esta mensagem, copie o arquivo .ed da era para os arquivos para ver o texto na área de transferência, destacando a mensagem e pressionando **ctrl -C** no Windows e no Linux ou **⌘ -C** no macOS. Em seguida, cole-o (usando **ctrl -V** no Windows e Linux ou **⌘ -V** no macOS) como o valor da string armazenado na mensagem na linha 4. Lembre-se de manter as aspas simples no início e no final da string. A nova linha 4 se parece com isso (com a alteração em negrito):

4. message = '.ed era meht para fi, terces a peek nac eerhT'

Agora, quando você executa o programa *reverseCipher.py* , a saída é descriptografada para a mensagem original:

Três podem manter um segredo, se dois deles estiverem mortos.

Configurando Comentários e Variáveis

As duas primeiras linhas em *reverseCipher.py* são comentários explicando o que é o programa e o site onde você pode encontrá-lo.

1. Cifra reversa

2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)

A parte Licenciada da BSD significa que este programa é gratuito para copiar e modificar por qualquer pessoa, desde que o programa retenha os créditos para o autor original (neste caso, o site do livro em

<https://www.nostarch.com/crackingcodes/> no segunda linha). Eu gosto de ter essa informação no arquivo, então se ele é copiado pela internet, uma pessoa que faz o download sempre sabe onde procurar pela fonte original. Eles também sabem que este programa é um software de código aberto e gratuito para distribuição a outros.

A linha 3 é apenas uma linha em branco e o Python a ignora. A linha 4 armazena a string que queremos criptografar em uma variável chamada message :

4. message = 'Três podem manter um segredo, se dois deles estiverem mortos.'

Sempre que quisermos criptografar ou descriptografar uma nova string, basta digitar a string diretamente no código da linha 4.

A variável traduzida na linha 5 é onde nosso programa armazenará a string invertida:

5. traduzido = "

No início do programa, a variável traduzida contém essa string em branco. (Lembre-se de que a string em branco é composta por dois caracteres de aspas simples e não por um caractere de aspas duplas.)

Encontrando o comprimento de uma corda

A linha 7 é uma instrução de atribuição que armazena um valor em uma variável denominada i :

7. $i = \text{len}(\text{mensagem}) - 1$

A expressão avaliada e armazenada na variável é $\text{len}(\text{mensagem}) - 1$. A primeira parte desta expressão, $\text{len}(\text{message})$, é uma chamada de função para a função $\text{len}()$, que aceita um argumento de string, assim como $\text{print}()$, e retorna um valor inteiro de quantos caracteres estão na string (isto é, o *comprimento* da string). Neste caso, passamos a variável `message` para $\text{len}()$, então $\text{len}(\text{message})$ retorna quantos caracteres estão no valor da string armazenado na `mensagem`.

Vamos experimentar com a função $\text{len}()$ no shell interativo. Digite o seguinte no shell interativo:

```
>>> len('Olá')  
5  
>>> len("")  
0  
>>> spam = 'Al'  
>>> len(spam)  
2  
>>> len('Olá,' + " + 'mundo!')  
13
```

A partir do valor de retorno de $\text{len}()$, sabemos que a string 'Hello' tem cinco caracteres e a string em branco possui zero caracteres. Se armazenarmos a string 'Al' em uma variável e depois passarmos a variável para $\text{len}()$, a função retornará 2. Se passarmos a expressão 'Olá,' + " + 'mundo!' para a função $\text{len}()$, retorna 13. A razão é que 'Olá,' + " + 'mundo!' avalia para o valor da string 'Hello, world!', que tem 13 caracteres nele. (O espaço e o ponto de exclamação contam como caracteres.)

Agora que você entende como a função $\text{len}()$ funciona, vamos voltar para a linha 7 do programa `reverseCipher.py`. A linha 7 encontra o índice do último caractere em `mensagem`, subtraindo 1 de $\text{len}(\text{mensagem})$. Ele tem que subtrair 1 porque os índices de, por exemplo, uma string de 5 caracteres como 'Hello' são de 0 a 4. Esse inteiro é então armazenado na variável `i`.

Apresentando o while Loop

A linha 8 é um tipo de instrução em Python chamada loop while ou while :

8. enquanto $i >= 0$:

Um loop while é composto de quatro partes (como mostrado na [Figura 4-1](#)).

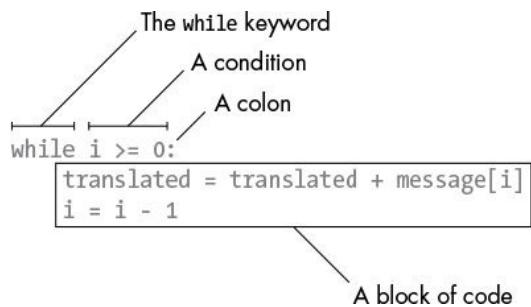


Figura 4-1: As partes de um loop while

Uma *condição* é uma expressão usada em uma instrução while . O bloco de código na instrução while será executado enquanto a condição for verdadeira.

Para entender os loops while , primeiro você precisa aprender sobre booleanos, operadores de comparação e blocos.

O tipo de dados booleano

O tipo de dados *booleano* tem apenas dois valores: verdadeiro ou falso . Esses valores booleanos, ou *bools* , *diferenciam* maiúsculas de minúsculas (você sempre precisa capitalizar o *T* e *F* , deixando o restante em minúsculas). Eles não são valores de string, então você não coloca aspas ao redor de True ou False .

Experimente alguns bools inserindo o seguinte no shell interativo:

```
>>> spam = True
>>> spam
Verdade
>>> spam = falso
>>> spam
Falso
```

Como um valor de qualquer outro tipo de dados, os bools podem ser armazenados em variáveis.

Operadores de Comparação

Na linha 8 do programa *reverseCipher.py* , observe a expressão após a palavra-chave while :

8. enquanto $i >= 0$:

A expressão que segue a palavra-chave while (a parte $i >= 0$) contém dois valores (o valor na variável *i* e o valor inteiro 0) conectados pelo sinal \geq ,

chamado de operador "maior que ou igual". O operador $>=$ é um *operador de comparação*.

Usamos operadores de comparação para comparar dois valores e avaliar para um valor Booleano Verdadeiro ou Falso. [A Tabela 4-1](#) lista os operadores de comparação.

Tabela 4-1: Operadores de Comparação

Sinal de operador	Nome do operador
<	Menos que
>	Melhor que
\leq	Menos que ou igual a
\geq	Melhor que ou igual a
$=$	Igual a
\neq	Não é igual a

Digite as seguintes expressões no shell interativo para ver o valor booleano que elas avaliam:

```
>>> 0 <6  
Verdade  
>>> 6 <0  
Falso  
>>> 50 <10,5  
Falso  
>>> 10,5 <11,3  
Verdade  
>>> 10 <10  
Falso
```

A expressão $0 <6$ retorna o valor booleano True porque o número 0 é menor que

o número 6 . Mas como 6 não é menor que 0 , a expressão $6 < 0$ é avaliada como False . A expressão $50 < 10,5$ é False porque 50 não é menor que 10,5 . A expressão $10 < 11,3$ é avaliada como Verdadeiro porque 10,5 é menor que 11,3 .

Olhe novamente para $10 < 10$. É falso porque o número 10 não é menor que o número 10 . Eles são exatamente os mesmos. (Se Alice tivesse a mesma altura que Bob, você não diria que Alice era mais baixa que Bob. Essa afirmação seria falsa.)

Insira algumas expressões usando os operadores \leq (menor que ou igual a) e \geq (maior que ou igual a):

```
>>> 10 <= 20
```

Verdade

```
>>> 10 <= 10
```

Verdade

```
>>> 10 >= 20
```

Falso

```
>>> 20 >= 20
```

Verdade

Observe que $10 \leq 10$ é True porque o operador verifica se 10 é menor ou igual a 10. Lembre-se que para os operadores “menor que ou igual a” e “maior que ou igual a”, o sinal < ou > sempre vem antes do sinal = .

Agora insira algumas expressões que usam os operadores == (igual a) e != (Não igual a) no shell para ver como elas funcionam:

```
>>> 10 == 10
```

Verdade

```
>>> 10 == 11
```

Falso

```
>>> 11 == 10
```

Falso

```
>>> 10 != 10
```

Falso

```
>>> 10 != 11
```

Verdade

Esses operadores funcionam como você esperaria para inteiros. A comparação de números inteiros que são iguais entre si com o operador == é avaliada como

True e valores desiguais como False . Quando você compara com o operador != , É o oposto.

As comparações de strings funcionam de maneira semelhante:

```
>>> 'Olá' == 'Olá'  
Verdade  
>>> 'Olá' == 'Adeus'  
Falso  
>>> 'Olá' == 'HELLO'  
Falso  
>>> 'Adeus' != 'Olá'  
Verdade
```

A capitalização é importante para o Python, portanto, os valores de string que não correspondem exatamente à capitalização não são a mesma string. Por exemplo, as strings 'Hello' e 'HELLO' não são iguais entre si, portanto, compará-las com == resultará em False .

Observe a diferença entre o operador de atribuição (=) e o operador de comparação "igual a" (==). O único sinal de igual (=) é usado para atribuir um valor a uma variável, e o sinal de igual duplo (==) é usado em expressões para verificar se dois valores são os mesmos. Se você está perguntando ao Python se duas coisas são iguais, use == . Se você estiver dizendo ao Python para definir uma variável como um valor, use = .

Em Python, os valores de string e inteiro são sempre considerados valores diferentes e nunca serão iguais entre si. Por exemplo, insira o seguinte no shell interativo:

```
>>> 42 == 'Olá'  
Falso  
>>> 42 == '42'  
Falso  
>>> 10 == 10,0  
Verdade
```

Mesmo parecendo iguais, o inteiro 42 e a string '42' não são considerados iguais porque uma string não é o mesmo que um número. Inteiros e números de ponto flutuante podem ser iguais entre si porque são ambos números.

Quando estiver trabalhando com operadores de comparação, lembre-se de que

toda expressão sempre é avaliada como True ou False .

Blocos

Um *bloco* é uma ou mais linhas de código agrupadas com a mesma quantidade mínima de *recuo* (ou seja, o número de espaços na frente da linha).

Um bloco começa quando uma linha é recuada por quatro espaços. Qualquer linha a seguir que também seja recuada por pelo menos quatro espaços faz parte do bloco. Quando uma linha é recuada com outros quatro espaços (para um total de oito espaços à frente da linha), um novo bloco começa dentro do primeiro bloco. Um bloco termina quando há uma linha de código com o mesmo recuo de antes do início do bloco.

Vamos ver alguns códigos imaginários (não importa qual seja o código, porque vamos nos concentrar apenas no recuo de cada linha). Os espaços recuados são substituídos por pontos cinzentos para facilitar a contagem.

```
1. codecodecode      # 0 spaces of indentation
2. ....codecodecode # 4 spaces of indentation
3. ....codecodecode # 4 spaces of indentation
4. .........codecodecode # 8 spaces of indentation
5. ....codecodecode   # 4 spaces of indentation
6.
7. ....codecodecode # 4 spaces of indentation
8. codecodecode     # 0 spaces of indentation
```

Você pode ver que a linha 1 não tem recuo; ou seja, há espaços zero na frente da linha de código. Mas a linha 2 tem quatro espaços de recuo. Como essa é uma quantidade maior de recuo que a linha anterior, sabemos que um novo bloco já começou. A linha 3 também tem quatro espaços de recuo, então sabemos que o bloco continua na linha 3.

A linha 4 tem ainda mais indentação (oito espaços), então um novo bloco foi iniciado. Este bloco está dentro do outro bloco. No Python, você pode ter blocos dentro de blocos.

Na linha 5, a quantidade de recuo diminuiu para quatro, por isso sabemos que o bloco na linha anterior terminou. A linha 4 é a única linha nesse bloco. Como a linha 5 tem a mesma quantidade de indentação que o bloco nas linhas 2 e 3, ela ainda faz parte do bloco externo original, mesmo que não faça parte do bloco na linha 4.

A linha 6 é uma linha em branco, então nós apenas a ignoramos; isso não afeta os blocos.

A linha 7 tem quatro espaços de indentação, então sabemos que o bloco que começou na linha 2 continuou na linha 7.

A linha 8 tem zero espaços de recuo, o que é menos recuo que a linha anterior. Esta diminuição no recuo nos diz que o bloco anterior, o bloco que começou na linha 2, terminou.

Este código mostra dois blocos. O primeiro bloco vai da linha 2 à linha 7. O segundo bloco consiste apenas na linha 4 (e está dentro do outro bloco).

NOTA

Os blocos nem sempre precisam ser delineados por quatro espaços. Os blocos podem usar qualquer número de espaços, mas a convenção é usar quatro por recuo.

A declaração while Loop

Vamos dar uma olhada na declaração full while começando na linha 8 de *reverseCipher.py* :

8. enquanto $i >= 0$:
9. traduzido = traduzido + mensagem [i]
10. $i = i - 1$
- 11
12. imprimir (tradução)

Uma instrução while diz ao Python para verificar primeiro o que a condição avalia, que na linha 8 é $i >= 0$. Você pode pensar na instrução while enquanto $i >= 0$: como significando “Enquanto a variável i é maior ou igual a zero, continue executando o código no bloco a seguir.” Se a condição for avaliada como True , a execução do programa entrará no bloco seguindo a declaração while .

Observando o recuo, você pode ver que esse bloco é composto pelas linhas 9 e 10. Quando ele atinge a parte inferior do bloco, a execução do programa volta para a instrução while na linha 8 e verifica a condição novamente. Se ainda for True , a execução vai para o início do bloco e executa o código no bloco novamente.

Se a condição da instrução while for avaliada como False , a execução do programa pula o código dentro do bloco a seguir e salta para a primeira linha após o bloco (que é a linha 12).

“Crescendo” uma corda

Tenha em mente que na linha 7, a variável i é primeiro definida para o comprimento da mensagem menos 1, e o loop while na linha 8 continua executando as linhas dentro do bloco seguinte até que a condição $i \geq 0$ seja False :

7. `i = len(mensagem) - 1`
8. `enquanto i >= 0:`
9. `traduzido = traduzido + mensagem [i]`
10. `i = i - 1`
- 11
12. `imprimir (tradução)`

A linha 9 é uma instrução de atribuição que armazena um valor na variável traduzida . O valor armazenado é o valor atual da tradução concatenada com o caractere no índice i na mensagem . Como resultado, o valor da string armazenado na tradução “aumenta” um caractere por vez até se tornar a string totalmente criptografada.

A linha 10 também é uma declaração de atribuição. Ele pega o valor inteiro atual em i e subtrai 1 dele (isso é chamado de *decrementar* a variável). Em seguida, ele armazena esse valor como o novo valor de i .

A próxima linha é 12, mas como essa linha tem menos recuo, o Python sabe que o bloco da instrução while terminou. Então, em vez de seguir para a linha 12, a execução do programa volta para a linha 8, onde a condição do loop while é verificada novamente. Se a condição for True , as linhas dentro do bloco (linhas 9 e 10) serão executadas novamente. Isso continua acontecendo até que a condição seja False (isto é, quando i é menor que 0), nesse caso a execução do programa vai para a primeira linha após o bloco (linha 12).

Vamos pensar sobre o comportamento desse loop para entender quantas vezes ele executa o código no bloco. A variável i começa com o valor do último índice da mensagem , e a variável traduzida começa como um espaço em branco corda. Então, dentro do loop, o valor da mensagem [i] (que é o último caractere na string da mensagem , porque eu terei o valor do último índice) é adicionado ao final da string traduzida .

Então o valor em i é decrementado (isto é, reduzido) por 1 , significando que a mensagem [i] será o segundo a último caractere. Então, enquanto i como um índice continua se movendo da parte de trás da string na mensagem para a frente, a mensagem da string [i] é adicionada ao final da tradução . É assim que a

tradução acaba mantendo o reverso da string na mensagem . Quando i finalmente é definido como -1 , o que acontece quando atingimos o índice 0 da mensagem, a condição do loop while é False , e a execução salta para a linha 12:

12. imprimir (tradução)

No final do programa, na linha 12, nós imprimimos o conteúdo da variável traduzida (isto é, a string '.ed ed era meht fo fi f, põe um peek nac eerht') na tela. Isso mostra ao usuário como é a string invertida.

Se você ainda tiver problemas para entender como o código no loop while reverte a string, tente adicionar a nova linha (mostrada em negrito) ao bloco do loop:

8. enquanto $i >= 0$:

9. traduzido = traduzido + mensagem [i]

10. print ('i é', i, ', mensagem [i] é', mensagem [i], ', traduzida é', traduzido)

11. $i = i - 1$

12

13. imprimir (tradução)

A linha 10 imprime os valores de i , message [i] e traduzidos juntamente com os rótulos de string cada vez que a execução passa pelo loop (ou seja, em cada *iteração* do loop). Desta vez, não estamos usando a concatenação de strings, mas algo novo. As vírgulas informam à função print () que estamos imprimindo seis coisas separadas, então a função adiciona um espaço entre elas. Agora, quando você executa o programa, você pode ver como a variável traduzida "cresce". A saída é assim:

eu é 48, mensagem [i] é. traduzido é.

i é 47, mensagem [i] é d, traduzido é .d

eu é 46, a mensagem [i] é uma, traduzida é .da

eu tenho 45 anos, a mensagem [i] é e, traduzida é .dae

i é 44, mensagem [i] é d, traduzida é .daed

i é 43, a mensagem [i] é, traduzida é .daed

i é 42, mensagem [i] é e, traduzida é .daed e

i é 41, mensagem [i] é r, traduzida é .ed er

eu tenho 40 anos, a mensagem [i] é uma era traduzida .daed

tenho 39 anos, a mensagem [i] é, traduzida é era .daed

eu tenho 38 anos, a mensagem [i] é m, traduzida é .daed era m

tenho 37 anos, mensagem [i] é e, traduzido é .daed era eu
eu tenho 36 anos, a mensagem [i] é h, traduzida é .daed era meh
i é 35, mensagem [i] é t, traduzida é .daed era meht
i é 34, a mensagem [i] é, traduzida é .daed era meht
i é 33, mensagem [i] é f, traduzida é .daed era meht f
eu tenho 32 anos, a mensagem [i] é o, traduzida é .daed era meht fo
tenho 31 anos, a mensagem [i] é, traduzida é .daed era meht fo
Eu tenho 30 anos, mensagem [i] é o, traduzido é.
eu tenho 29 anos, a mensagem [i] é w, traduzida é .daed era meht fo ow ow
Eu tenho 28 anos, a mensagem [i] é t, traduzida é.
eu tenho 27 anos, a mensagem [i] é, traduzida é .daed era meht para fo
eu tenho 26 anos, a mensagem [i] é f, a tradução é a .daed era meht fo f
tenho 25 anos, a mensagem [i] é i, traduzida é .daed era meht fo owt fi
eu tenho 24 anos, a mensagem [i] é, traduzida é .daed era meht fo owt fi
eu tenho 23 anos, a mensagem [i] é, a tradução é a .daed era meht fo fi fi,
tenho 22 anos, a mensagem [i] é t, traduzida é a época .daed era meht fi fi, t
eu tenho 21 anos, a mensagem [i] é e, traduzi é a era .daed meht fo fi fi, te
eu tenho 20 anos, a mensagem [i] é r, traduzida é .daed era meht para fi, ter
eu tenho 19 anos, a mensagem [i] é c, traduzida é a era .eded meht fi fi, terc
eu tenho 18 anos, a mensagem [i] é e, traduzido é .daed era meht fo fi, tercei
eu tenho 17 anos, a mensagem [i] é s, traduzida é .daed era meht fo fi fi, terces
eu tenho 16 anos, a mensagem [i] é, traduzida é .daed era meht fo fi fi, terces
eu tenho 15 anos, a mensagem [i] é uma, traduzida é .daed era meht fi fi, terces a
eu tenho 14 anos, a mensagem [i] é, traduzida é .daed era meht fi fi, terces a
eu tenho 13 anos, a mensagem [i] é p, a tradução é a .daed era meht fi fi, terces
ap
i é 12, mensagem [i] é e, traduzido é .daed era meht fi fi, terces a pe
eu é 11, mensagem [i] é e, traduzido é .daed era meht para fi, terces um xixi
eu é 10, mensagem [i] é k, traduzida é .daed era meht para fi, terces uma espiada
eu tenho 9 anos, a mensagem [i] é, traduzi é a era .daed meht para fi, terces uma
espiada
i é 8, mensagem [i] é n, traduzido é .daed era meht fo fi fi, terces a peek n
eu é 7, a mensagem [i] é um, traduzido é.
eu é 6, mensagem [i] é c, traduzido é .daed era meht fi fi, terces a peek nac
eu é 5, mensagem [i] é, traduzido é .daed era meht para fi, terces um peek nac
i é 4, mensagem [i] é e, traduzido é .daed era meht fi fi, terces a peek nac e
eu é 3, a mensagem [i] é e, traduzido é .daed era meht para fi, terces um peek nac

ee

i é 2, mensagem [i] é r, traduzido é .daed era meht para fi, terces a peek nac eer
eu é 1, mensagem [i] é h, traduzido é .daed era meht fo fi fi, terces a peek nac eerh

i é 0, mensagem [i] é T, traduzida é .daed era meht fo fi fi, terces a peek nac eerhT

A linha de saída, "i é 48, mensagem [i] é., Traduzida é". , mostra o que as expressões i , message [i] e traduzidas avaliam após a mensagem de string [i] ter sido adicionada ao final da tradução, mas antes de i ser decrementada. Você pode ver que na primeira vez que a execução do programa passa pelo loop, i é configurado para 48 , então message [i] (isto é, message [48]) é a string '.'. A variável traduzida começou como uma string em branco, mas quando a mensagem [i] foi adicionada ao final dela na linha 9, ela se tornou o valor da string '.'.

Na próxima iteração do loop, a saída é "i é 47, mensagem [i] é d, traduzido é .d" . Você pode ver que eu fui decrementado de 48 para 47 , então agora a mensagem [i] é a mensagem [47] , que é a string 'd' . (Esse é o segundo 'd' in 'dead'). Este 'd' é adicionado ao final da tradução , então traduzido agora é o valor '.d' .

Agora você pode ver como a string da variável traduzida é “crescida” lentamente de uma string em branco para a mensagem invertida.

Aprimorando o programa com um prompt de entrada ()

Os programas neste livro são todos projetados para que as cadeias de caracteres que estão sendo criptografadas ou descriptografadas sejam digitadas diretamente no código-fonte como instruções de atribuição. Isso é conveniente enquanto desenvolvemos os programas, mas você não deve esperar que os usuários se sintam à vontade para modificar o código-fonte. Para tornar os programas mais fáceis de usar e compartilhar, você pode modificar as instruções de atribuição para que eles chamem a função input () . Você também pode passar uma string para input () para exibir um prompt para que o usuário insira uma string para criptografar. Por exemplo, altere a linha 4 em *reverseCipher.py* para isto:

```
4. message = input ('Digite a mensagem:')
```

Quando você executa o programa, ele imprime a solicitação na tela e aguarda que o usuário insira uma mensagem. A mensagem que o usuário digita será o

valor da string armazenado na variável de mensagem . Quando você executa o programa agora, você pode colocar qualquer string que quiser e obter uma saída como esta:

Digite a mensagem: Olá, mundo!

```
! dlrow, olleH
```

Resumo

Acabamos de concluir nosso segundo programa, que manipula uma string em uma nova string usando técnicas do [Capítulo 3](#) , como indexação e concatenação. Uma parte importante do programa era a função len () , que recebe um argumento de string e retorna um inteiro de quantos caracteres estão na string.

Você também aprendeu sobre o tipo de dados booleano, que tem apenas dois valores, True e False . Operadores de comparação == , != , < , > , <= E > = podem comparar dois valores e avaliar um valor booleano.

Condições são expressões que usam operadores de comparação e avaliam um tipo de dados booleano. Eles são usados em loops while , que executarão o código no bloco após a instrução while até que a condição seja avaliada como False . Um bloco é composto de linhas com o mesmo nível de recuo, incluindo quaisquer blocos dentro delas.

Agora que você aprendeu a manipular texto, pode começar a criar programas com os quais o usuário possa executar e interagir. Isso é importante porque o texto é a principal maneira pela qual o usuário e o computador se comunicam entre si.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. O que o seguinte trecho de código imprime na tela?

```
print(len('Hello') + len('Hello'))
```

2. O que esse código imprime?

```
i = 0
while i <3:
    print('Hello')
```

i = i + 1

3. Como sobre esse código?

i = 0

spam = 'Olá'

while i <5:

spam = spam + spam [i]

i = i + 1

print (spam)

4. E isto?

i = 0

while i <4:

while i <6:

i = i + 2

print (i)

5

O CESAR DE CAESAR

"O BIG BROTHER ESTÁ OLHANDO PARA VOCÊ."

- George Orwell , mil novacentos e oitenta e quatro



No [Capítulo 1](#) , usamos uma roda de códigos e um gráfico de letras e números para implementar a cifra de César. Neste capítulo, vamos implementar a cifra de César em um programa de computador.

A cifra reversa que fizemos no [Capítulo 4](#) sempre criptografa da mesma maneira. Mas a cifra de César usa chaves, que criptografam a mensagem de forma diferente, dependendo de qual chave é usada. As chaves para a cifra de César são os inteiros de 0 a 25 . Mesmo que um criptoanalista saiba que a cifra de César foi usada, só isso não lhes dá informações suficientes para quebrar a cifra. Eles também devem conhecer a chave.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A declaração de importação
- Constantes
- para laços
- if, else e elif statements
- O em e não em operadores
- O método da string find ()

Código-fonte para o programa Cipher Cipher

Digite o seguinte código no editor de arquivos e salve-o como *caesarCipher.py* . Em seguida, faça o download do módulo *pyperclip.py* em <https://www.nostarch.com/crackingcodes/> e coloque-o no mesmo diretório (ou seja, na mesma pasta) que o arquivo *caesarCipher.py* . Este módulo será importado pelo *caesarCipher.py* ; discutiremos isso com mais detalhes em “[Importando Módulos e Configurando Variáveis](#) ” na [página 56](#) .

Quando terminar de configurar os arquivos, pressione F5 para executar o programa. Se você encontrar algum erro ou problema com seu código, poderá compará-lo ao código do livro usando a ferramenta de comparação on-line em <https://www.nostarch.com/crackingcodes/> .

caesarCipher.py

```
1. Cifra César
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip
5
6. # A string a ser criptografada / descriptografada:
7. message = 'Esta é a minha mensagem secreta'.
8
9. # A chave de criptografia / descriptografia:
10. chave = 13
11
12. # Se o programa criptografa ou descriptografa:
13. mode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.
14
```

15. # Todos os símbolos possíveis que podem ser criptografados:
16. SÍMBOLOS =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
67890!
17
18. # Armazene o formulário criptografado / decriptografado da mensagem:
19. traduzido = "
20
21. para símbolo na mensagem:
22. Nota: Somente símbolos na string SYMBOLS podem ser
criptografado / descriptografado.
23. se símbolo em SÍMBOLOS:
24. symbolIndex = SYMBOLS.find (símbolo)
25
26. # Execute criptografia / descriptografia:
27. if mode == 'encriptar':
28. translatedIndex = symbolIndex + chave
29. modo elif == 'decifrar':
30. translatedIndex = symbolIndex - chave
31
32. # Manipular a envolvente, se necessário:
33. if translatedIndex >= len (SÍMBOLOS):
34. translatedIndex = traduzidoIndex - len (SÍMBOLOS)
35. elif translatedIndex <0:
36. translatedIndex = traduzidoIndex + len (SÍMBOLOS)
37
38. traduzido = traduzido + SYMBOLS [translatedIndex]
39. else:
40. # Anexar o símbolo sem criptografar / descriptografar:
41. translated = traduzido + símbolo
42.
43. # Mostra a string traduzida:
44. imprimir (tradução)
45. pyperclip.copy (tradução)

Exemplo de execução do programa Cipher Cipher

Quando você executa o programa *caesarCipher.py* , a saída é assim:

guv6Jv6Jz! J6rp5r7Jzr66ntrM

A saída é a string "Esta é a minha mensagem secreta". criptografado com a cifra de César usando uma chave de 13 . O programa de codificação Caesar que você acabou de executar copia automaticamente essa string criptografada para a área de transferência para que você possa colá-la em um arquivo de email ou de texto. Como resultado, você pode enviar facilmente a saída criptografada do programa para outra pessoa.

Você pode ver a seguinte mensagem de erro ao executar o programa:

Traceback (última chamada mais recente):

Arquivo "C: \ caesarCipher.py", linha 4, em <module>

importar pyperclip

ImportError: Nenhum módulo chamado pyperclip

Se assim for, provavelmente você não baixou o módulo *pyperclip.py* na pasta correta. Se você confirmar que *pyperclip.py* está na pasta com *caesarCipher.py* mas ainda assim não consegue fazer com que o módulo funcione, apenas comente o código nas linhas 4 e 45 (que têm o texto *pyperclip* nelas) do *caesarCipher.py* programa colocando um # na frente deles. Isso faz com que Python ignore o código isso depende do módulo *pyperclip.py* e deve permitir que o programa seja executado com sucesso. Observe que, se você comentar esse código, o texto criptografado ou descriptografado não será copiado para a área de transferência no final do programa. Você também pode comentar o código *pyperclip* dos programas em capítulos futuros, que também removerão a funcionalidade de copiar para a área de transferência desses programas.

Para descriptografar a mensagem, basta colar o texto de saída como o novo valor armazenado na variável de mensagem na linha 7. Em seguida, altere a instrução de atribuição na linha 13 para armazenar a sequência 'decifrar' no modo de variável:

6. # A string a ser criptografada / descriptografada:

7. message = 'guv6Jv6Jz! J6rp5r7Jzr66ntrM'

8

9. # A chave de criptografia / descriptografia:

10. chave = 13

11

12. # Se o programa criptografa ou descriptografa:

13. mode = 'decrypt' # Configure como 'encrypt' ou 'decrypt'.

Quando você executa o programa agora, a saída se parece com isso:

Esta é minha mensagem secreta.

Importando Módulos e Configurando Variáveis

Embora o Python inclua muitas funções internas, algumas funções existem em programas separados chamados módulos. *Módulos* são programas em Python que contêm funções adicionais que seu programa pode usar. Nós importamos módulos com a declaração de importação apropriadamente nomeada, que consiste na palavra-chave import seguida do nome do módulo.

A linha 4 contém uma declaração de importação :

1. Cifra César
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar pyperclip

Neste caso, estamos importando um módulo chamado pyperclip para que possamos chamar a função pyperclip.copy () posteriormente neste programa. A função pyperclip.copy () copiará automaticamente as strings para a área de transferência do seu computador, para que você possa colá-las convenientemente em outros programas.

As próximas linhas no *caesarCipher.py* definem três variáveis:

6. # A string a ser criptografada / descriptografada:
7. message = 'Esta é a minha mensagem secreta'.
- 8
9. # A chave de criptografia / descriptografia:
10. chave = 13
- 11
12. # Se o programa criptografa ou descriptografa:
13. mode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.

A variável de mensagem armazena a cadeia a ser criptografada ou descriptografada, e a variável de chave armazena o inteiro da chave de criptografia. A variável mode armazena a string 'encrypt' , que faz o código mais tarde no programa criptografar a string na mensagem , ou 'decrypt' , que faz o programa decriptar em vez de criptografar.

Constantes e Variáveis

Constantes são variáveis cujos valores não devem ser alterados quando o programa é executado. Por exemplo, o programa de cifra de César precisa de uma string que contenha todos os caracteres possíveis que possam ser criptografados com essa cifra de César. Como essa string não deve mudar, nós a armazenamos na variável constante chamada SYMBOLS na linha 16:

15. # Todos os símbolos possíveis que podem ser criptografados:

16. SÍMBOLOS =

```
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345  
67890!'
```

Símbolo é um termo comum usado na criptografia para um único caractere que uma cifra pode criptografar ou descriptografar. Um *conjunto de símbolos* é todo símbolo possível que uma cifra é configurada para criptografar ou descriptografar. Porque usaremos o conjunto de símbolos muitas vezes neste programa, e porque não queremos digitar o valor completo da string toda vez que ele aparecer no programa (podemos criar erros de digitação, o que causaria erros), usamos uma constante variável para armazenar o conjunto de símbolos. Entramos o código para o valor da string uma vez e o colocamos na constante SYMBOLS .

Observe que SYMBOLS está em todas as letras maiúsculas, que é a convenção de nomenclatura para constantes. Embora pudéssemos alterar SYMBOLS como qualquer outra variável, o nome todo em letras maiúsculas lembra ao programador não escrever código que o faça.

Tal como acontece com todas as convenções, não *temos* que seguir este. Mas isso torna mais fácil para outros programadores entender como essas variáveis são usadas. (Pode até ajudá-lo quando você está olhando para o seu próprio código mais tarde.)

Na linha 19, o programa armazena uma string em branco em uma variável chamada traduzida que armazenará posteriormente a mensagem criptografada ou descriptografada:

18. # Armazene o formulário criptografado / descriptografado da mensagem:
19. traduzido = "

Assim como no código reverso no [Capítulo 5](#), até o final do programa, a variável traduzida conterá a mensagem completamente criptografada (ou descriptografada). Mas por enquanto começa como uma string em branco.

A declaração Loop for

Na linha 21, usamos um tipo de loop chamado loop for :

21. para símbolo na mensagem:

Lembre-se de que um loop while fará um loop enquanto uma determinada condição for True . O loop for tem um propósito ligeiramente diferente e não possui uma condição como o loop while. Em vez disso, faz um loop sobre uma string ou um grupo de valores. [A Figura 5-1](#) mostra as seis partes de um loop for

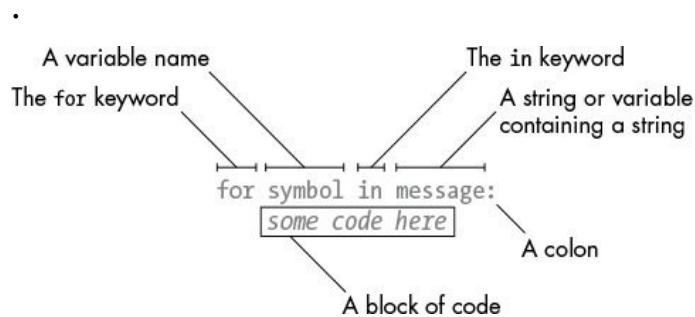


Figura 5-1: As seis partes de uma declaração de loop for

Cada vez que a execução do programa passa pelo loop (ou seja, em cada iteração pelo loop), a variável na instrução for (que na linha 21 é símbolo) assume o valor do próximo caractere na variável que contém uma string (que neste caso é mensagem). A instrução for é semelhante a uma instrução de atribuição porque a variável é criada e atribuída a um valor, exceto que a instrução for percorre diferentes valores para atribuir a variável.

Um exemplo para o laço

Por exemplo, digite o seguinte no shell interativo. Observe que depois de digitar a primeira linha, o prompt >>> desaparecerá (representado em nosso código como ...) porque o shell está esperando um bloco de código após os dois pontos da instrução. No shell interativo, o bloco terminará quando você inserir uma linha em branco:

```
>>> para carta em 'Howdy':
```

```
... print ('A letra é' + letra)
```

```
...
```

```
A carta é H
```

```
A carta é o
```

```
A letra é w
```

```
A letra é d
```

A letra é y

Este código faz um loop sobre cada caractere na string 'Howdy' . Quando isso acontece, a letra variável assume o valor de cada caractere em 'Howdy', um de cada vez, em ordem. Para ver isso em ação, escrevemos código no loop que imprime o valor da letra para cada iteração.

Um loop while Equivalente de um loop for

O loop for é muito semelhante ao loop while, mas quando você precisa apenas iterar sobre caracteres em uma string, usar um loop for é mais eficiente. Você poderia fazer um loop while agir como um loop for , escrevendo um pouco mais de código:

```
❶ >>> i = 0
❷ >>> while i <len ('Howdy'):
❸ ... letra = 'Howdy' [i]
Print ... print ('A letra é' + letra)
❹ ... i = i + 1
```

```
...
A carta é H
A carta é o
A letra é w
A letra é d
A letra é y
```

Observe que esse loop while funciona da mesma forma que o loop for, mas não é tão curto e simples quanto o loop for . Primeiro, definimos uma nova variável i para 0 antes da instrução while ❶ . Esta declaração tem uma condição que será avaliada como True , desde que a variável i seja menor que o comprimento da string 'Howdy' ❷ . Como eu é um número inteiro e só monitora a posição atual na string, precisamos declarar uma variável de letra separada para conter o caractere na string na posição i ❸ . Então, podemos imprimir o valor atual da letra para obter a mesma saída que o loop for ❹ . Quando o código terminar a execução, precisamos incrementar i adicionando 1 a ele para passar para a próxima posição ❺ .

Para entender as linhas 23 e 24 em *caesarCipher.py* , você precisa aprender sobre as instruções if , elif e else , os operadores in e not in e o método string find () . Vamos ver isso nas próximas seções.

A declaração if

A linha 23 na cifra de César tem outro tipo de instrução em Python - a declaração if :

23. se símbolo em SÍMBOLOS:

Você pode ler uma instrução if como “Se esta condição for True , execute o código no bloco a seguir. Caso contrário, se for Falso , pule o bloco. ”Uma instrução if é formatada usando a palavra-chave se seguida por uma condição, seguida por dois-pontos (:). O código a ser executado é recuado em um bloco, da mesma maneira que com loops.

Um exemplo se declaração

Vamos tentar um exemplo de uma declaração if . Abra uma nova janela do editor de arquivos, insira o código a seguir e salve-o como *checkPw.py* :

checkPw.py

```
print ('Digite sua senha'.)
❶ typedPassword = input ()
❷ se typedPassword == 'swordfish':
❸ print ('Acesso Concedido')
❹ print ('Feito')
```

Quando você executa este programa, ele exibe o texto Digite sua senha. e permite que o usuário digite uma senha. A senha é então armazenada na variável typedPassword ❶ . Em seguida, a instrução if verifica se a senha é igual à string 'swordfish' ❷ . Se estiver, a execução se move dentro do bloco após a instrução if para exibir o texto Access Granted to the user ❸ ; caso contrário, se typedPassword não for igual a 'swordfish' , a execução ignorará o bloco if statement. De qualquer forma, a execução continua no código após o bloco if para exibir Done ❹ .

A outra declaração

Frequentemente, queremos testar uma condição e executar um bloco de código se a condição for True e outro bloco de código, se for False . Podemos usar uma instrução else após o bloco de uma instrução if , e o bloco de código da outra instrução será executado se a condição da instrução if for False . Para uma instrução else , você apenas escreve a palavra-chave else e um ponto-e-vírgula (:). Ele não precisa de uma condição porque será executado se a condição da

instrução if não for verdadeira. Você pode ler o código como: "Se esta condição for True , execute este bloco, ou então, se for falso , execute este outro bloco."

Modifique o programa *checkPw.py* para se parecer com o seguinte (as novas linhas estão em negrito):

checkPw.py

```
print ('Digite sua senha'.)
typedPassword = input ()
❶ se typedPassword == 'swordfish':
    print ('Acesso Concedido')
outro:
❷ print ('Acesso negado')
❸ print ('Feito')
```

Esta versão do programa funciona quase da mesma forma que a versão anterior. O texto Acesso Concedido ainda será exibido se a condição da declaração if for True ❶ . Mas agora, se o usuário digitar algo diferente de espadarte , a condição da declaração if será False , fazendo com que a execução insira o bloco da instrução else e exiba Access Denied Access . De qualquer maneira, a execução continuará e exibirá Concluído .

A declaração elif

Outra instrução, chamada de instrução elif , também pode ser emparelhada com if . Como uma declaração if , ela tem uma condição. Como uma declaração else , ela segue uma instrução if (ou outra elif) e é executada se a condição da instrução if (ou elif) anterior for False . Você pode ler if , elif e outras instruções como, “Se esta condição for True , execute este bloco. Ou então, verifique se esta próxima condição é verdadeira . Ou então, apenas execute este último bloco. ”Qualquer número de instruções elif pode seguir uma instrução if . Modifique o programa *checkPw.py* novamente para torná-lo semelhante ao seguinte:

checkPw.py

```
print ('Digite sua senha'.)
typedPassword = input ()
❶ se typedPassword == 'swordfish':
❷ print ('Acesso Concedido')
❸ elif typedPassword == 'mary':
    print ('Dica: a senha é um peixe.')
```

```
❸ elif typedPassword == '12345':  
    print ('Essa é uma senha realmente óbvia'.)  
outro:  
    print ('Acesso negado')  
    print ('Feito')
```

Esse código contém quatro blocos para as instruções if , elif e else . Se o usuário digitar 12345 , então typedPassword == 'swordfish' será avaliado como False ❶ , então o primeiro bloco com print ('Access Granted') ❷ será ignorado. Em seguida, a execução verifica a condição typedPassword == 'mary' , que também é avaliada como False ❸ , de modo que o segundo bloco também é ignorado. A condição typedPassword == '12345' é True ❹ , portanto a execução entra no bloco após esta instrução elif para executar a impressão de código ('Essa é uma senha realmente óbvia') e pula todas as instruções elif e else restantes. *Observe que apenas um desses blocos será executado.*

Você pode ter zero ou mais instruções elif seguindo uma instrução if . Você pode ter zero ou uma, mas não várias instruções else , e a instrução else sempre vem por último, porque ela só é executada se nenhuma das condições for avaliada como True . A primeira instrução com uma condição True tem seu bloco executado. O restante das condições (mesmo se elas também forem verdadeiras) não são verificadas.

O em e não em operadores

A linha 23 em *caesarCipher.py* também usa o operador in :

23. se símbolo em SÍMBOLOS:

Um operador in pode conectar duas strings e ele será avaliado como True se a primeira string estiver dentro da segunda string ou avaliada como False, se não for. O em operador também pode ser emparelhado com não , o que fará o oposto. Digite o seguinte no shell interativo:

```
>>> 'olá' em 'olá mundo!'  
Verdade  
>>> 'olá' não no 'olá mundo!'  
Falso  
>>> 'ello' em 'olá mundo!'  
Verdade  
❶ >>> 'OLÁ' em 'olá mundo!'
```

Falso

❷ >>> " em 'Olá'

Verdade

Observe que os operadores in e not in diferenciam maiúsculas de minúsculas ❶ . Além disso, uma string em branco é sempre considerada em qualquer outra string ❷ .

Expressões usando os operadores in e not in são úteis para usar como condições de instruções if para executar algum código se uma string existir dentro de outra string.

Retornando a *caesarCipher.py* , a linha 23 verifica se a string em symbol (que é o loop for na linha 21 definida como um único caractere da string da mensagem) está na string SYMBOLS (o conjunto de símbolos de todos os caracteres que podem ser criptografados ou descriptografados) por este programa de cifra). Se o símbolo estiver em SYMBOLS , a execução entrará no bloco que segue começando na linha 24. Se não estiver, a execução pula esse bloco e, em vez disso, entra no bloco seguindo a instrução else da linha 39. O programa de codificação precisa executar um código diferente, dependendo se o símbolo está no conjunto de símbolos.

O método da string find ()

A linha 24 encontra o índice na string SYMBOLS, em que symbol é:

24. symbolIndex = SYMBOLS.find (símbolo)

Este código inclui uma chamada de método. Os métodos são exatamente como funções, exceto que são anexados a um valor com um período (ou na linha 24, uma variável contendo um valor). O nome desse método é find () e está sendo chamado no valor da string armazenado em SYMBOLS .

A maioria dos tipos de dados (como strings) tem métodos. O método find () recebe um argumento de string e retorna o índice inteiro de onde o argumento aparece na string do método. Digite o seguinte no shell interativo:

```
>>> 'hello'.find (' e ')
```

```
1
```

```
>>> 'hello'.find (' o ')
```

```
4
```

```
>>> spam = 'olá'
```

```
>>> spam.find ('h')
❶ 0
```

Você pode usar o método `find ()` em uma string ou em uma variável contendo um valor de string. Lembre-se que a indexação em Python começa com 0 , então quando o índice retornado por `find ()` é para o primeiro caractere na string, um 0 é retornado ❶ .

Se o argumento string não puder ser encontrado, o método `find ()` retornará o inteiro -1 . Digite o seguinte no shell interativo:

```
>>> 'hello'.find ('x')
-1
❶ >>> 'hello'.find ('H')
-1
```

Observe que o método `find ()` também diferencia maiúsculas de minúsculas ❶ .

A string que você passa como argumento para `find ()` pode ser mais de um caractere. O inteiro que `find ()` retorna será o índice do primeiro caractere onde o argumento é encontrado. Digite o seguinte no shell interativo:

```
>>> 'hello'.find ('ello')
1
>>> 'hello'.find ('lo')
3
>>> 'hello hello'.find ('e')
1
```

O método da string `find ()` é como uma versão mais específica do uso do operador `in` . Ele não apenas informa se uma string existe em outra string, mas também informa onde.

Criptografando e Descriptografando Símbolos

Agora que você entende `if` , `elif` e `else` statements; o operador `in` ; e o método de string `find ()` , será mais fácil entender como funciona o resto do programa de cifra de César.

O programa de criptografia só pode criptografar ou descriptografar símbolos que estão no conjunto de símbolos:

23. se símbolo em SÍMBOLOS:
24. `symbolIndex = SYMBOLS.find (símbolo)`

Portanto, antes de executar o código na linha 24, o programa deve descobrir se o símbolo está no conjunto de símbolos. Então ele pode encontrar o índice em SYMBOLS onde o símbolo está localizado. O índice retornado pela chamada `find()` é armazenado em `symbolIndex`.

Agora que temos o índice do símbolo atual armazenado em `symbolIndex`, podemos fazer a matemática de criptografia ou descriptografia nele. A cifra de César adiciona o número da chave ao índice do símbolo para criptografá-lo ou subtrai o número da chave do índice do símbolo para descriptografá-lo. Este valor é armazenado em `translatedIndex` porque será o índice em SYMBOLS do símbolo traduzido.

caesarCipher.py

```
26. # Execute criptografia / descriptografia:  
27. if mode == 'encriptar':  
28.     translatedIndex = symbolIndex + chave  
29.     modo elif == 'decifrar':  
30.         translatedIndex = symbolIndex - chave
```

A variável `mode` contém uma string que informa ao programa se deve ser criptografado ou descriptografado. Se esta string for 'encrypt' , a condição para a instrução `if` da linha 27 será True e a linha 28 será executada para adicionar a chave a `symbolIndex` (e o bloco após a instrução `elif` será ignorado). Caso contrário, se o modo for 'decifrado' , a linha 30 será executada para subtrair a chave .

Manipulando o Wraparound

Quando estávamos implementando a cifra de César com papel e lápis no [Capítulo 1](#) , às vezes adicionar ou subtrair a chave resultaria em um número maior ou igual ao tamanho do conjunto de símbolos ou menor que zero. Nesses casos, temos que adicionar ou subtrair o tamanho do conjunto de símbolos para que ele “envolva” ou retorne ao início ou ao final do conjunto de símbolos. Podemos usar o código `len(SYMBOLS)` para fazer isso, que retorna 66 , o comprimento da string SYMBOLS . As linhas 33 a 36 lidam com esse embrulho no programa de criptografia.

```
32. # Manipular a envolvente, se necessário:  
33. if translatedIndex >= len (SÍMBOLOS):  
34.     translatedIndex = traduzidoIndex - len (SÍMBOLOS)
```

35. elif translatedIndex <0:

36. translatedIndex = traduzidoIndex + len (SÍMBOLOS)

Se translatedIndex for maior ou igual a 66 , a condição na linha 33 é True e a linha 34 é executada (e a instrução elif na linha 35 é ignorada). A subtração do comprimento de SYMBOLS do translatedIndex aponta o índice da variável de volta para o início da string SYMBOLS . Caso contrário, o Python verificará se o translateIndex é menor que 0 . Se essa condição for True , a linha 36 será executada e o translateIndex será redefinido até o final da string SYMBOLS .

Você pode estar se perguntando por que não usamos apenas o valor inteiro 66 diretamente em vez de len (SYMBOLS) . Usando len (SYMBOLS) em vez de 66 , podemos adicionar ou remover símbolos de SYMBOLS e o resto do código ainda funcionará.

Agora que você tem o índice do símbolo traduzido em translatedIndex , SYMBOLS [translatedIndex] será avaliado como o símbolo traduzido. A linha 38 adiciona esse símbolo criptografado / decriptografado ao final da string traduzida usando concatenação de string:

38. traduzido = traduzido + SYMBOLS [translatedIndex]

Eventualmente, a string traduzida será toda a mensagem codificada ou decodificada.

Manipulando Símbolos Fora do Conjunto de Símbolos

A cadeia de mensagens pode conter caracteres que não estão na sequência SYMBOLS . Esses caracteres estão fora do conjunto de símbolos do programa de criptografia e não podem ser criptografados ou descriptografados. Em vez disso, eles serão anexados à string traduzida como está, o que acontece nas linhas 39 a 41:

39. else:

40. # Anexar o símbolo sem criptografar / descriptografar:

41. translated = traduzido + símbolo

A instrução else na linha 39 tem quatro espaços de recuo. Se você olhar o recuo das linhas acima, verá que ele está emparelhado com a instrução if na linha 23. Embora haja muito código entre essa instrução if e else , tudo pertence ao mesmo bloco de código.

Se a condição da instrução 23 da linha 23 fosse False , o bloco seria ignorado, e

a execução do programa entraria no bloco da instrução else começando na linha 41. Esse outro bloco tem apenas uma linha. Adiciona a string de símbolo inalterada ao final da tradução . Como resultado, os símbolos fora do conjunto de símbolos, como '%' ou '(', são adicionados à string traduzida sem serem criptografados ou descriptografados.

Exibindo e copiando a string traduzida

A linha 43 não possui recuo, o que significa que é a primeira linha após o bloco que iniciou na linha 21 (o bloco do loop for). No momento em que a execução do programa atinge a linha 44, ele passou por cada caractere na sequência de mensagens , criptografou (ou descriptografou) os caracteres e os adicionou à tradução :

43. # Mostra a string traduzida:
44. imprimir (tradução)
45. pyperclip.copy (tradução)

A linha 44 chama a função print () para exibir a string traduzida na tela. Observe que esta é a única chamada de print () em todo o programa. O computador faz muito trabalho para criptografar todas as letras da mensagem , manipular o contorno e manipular os caracteres que não são letras. Mas o usuário não precisa ver isso. O usuário só precisa ver a string final na tradução .

A linha 45 chama copy () , que recebe um argumento de string e o copia para a área de transferência. Como copy () é uma função no módulo pyperclip , devemos informar ao Python isso colocando pyperclip. na frente do nome da função. Se digitarmos copy (translated) em vez de pyperclip.copy (traduzido) , o Python nos fornecerá uma mensagem de erro porque não conseguirá localizar a função.

O Python também apresentará uma mensagem de erro se você esquecer a linha pyperclip de importação (linha 4) antes de tentar chamar pyperclip.copy () .

Esse é todo o programa de cifra de César. Quando você executá-lo, observe como seu computador pode executar o programa inteiro e criptografar a seqüência de caracteres em menos de um segundo. Mesmo se você digitar uma string muito longa para armazenar na mensagem variável, seu computador pode criptografar ou descriptografar a mensagem em um segundo ou dois. Compare isso com os vários minutos necessários para fazer isso com uma roda de criptografia. O programa copia automaticamente o texto criptografado para a

área de transferência para que o usuário possa simplesmente colá-lo em um email para enviar a alguém.

Criptografando Outros Símbolos

Um problema com a cifra de César que implementamos é que ela não pode criptografar caracteres fora de seu conjunto de símbolos. Por exemplo, se você criptografar a string 'Certifique-se de trazer o \$!!!'. com a chave 20 , a mensagem será criptografada em 'VyQ? A! yQ.9Qv! 381Q.2yQ \$\$\$ T' . Esta mensagem criptografada não esconde que você está se referindo a \$!!!. No entanto, podemos modificar o programa para criptografar outros símbolos.

Ao alterar a string armazenada em SYMBOLS para incluir mais caracteres, o programa os criptografará também, porque na linha 23, o símbolo de condição em SYMBOLS será True . O valor de symbolIndex será o índice de símbolo nesta nova variável constante SYMBOLS maior. O “wraparound” precisará adicionar ou subtrair o número de caracteres nesta nova string, mas isso já foi tratado porque usamos len (SYMBOLS) em vez de digitar 66 diretamente no código (é por isso que programamos dessa forma).

Por exemplo, você poderia expandir a linha 16 para ser:

SÍMBOLOS =

```
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345  
67890!?. ` ~@ # $% ^ & * () _ + - = [] {} |;: <>, / '
```

Tenha em mente que uma mensagem deve ser criptografada e descriptografada com o mesmo conjunto de símbolos para funcionar.

Resumo

Você aprendeu vários conceitos de programação e leu vários capítulos para chegar a esse ponto, mas agora você tem um programa que implementa uma criptografia secreta. E mais importante, você entende como esse código funciona.

Módulos são programas em Python que contêm funções úteis. Para usar essas funções, você deve primeiro importá-las usando uma instrução de importação . Para chamar funções em um módulo importado, coloque o nome do módulo e um ponto antes do nome da função, assim: module . função () .

Variáveis constantes são escritas em letras maiúsculas por convenção. Essas variáveis não devem ter seus valores alterados (embora nada impeça o

programador de escrever código). Constantes são úteis porque dão um “nome” a valores específicos em seu programa.

Métodos são funções anexadas a um valor de um determinado tipo de dados. O método de string find () retorna um inteiro da posição do argumento string passado para ele dentro da string em que é chamado.

Você aprendeu sobre várias novas maneiras de manipular quais linhas de código são executadas e quantas vezes cada linha é executada. Um loop for itera sobre todos os caracteres em um valor de string, definindo uma variável para cada caractere em cada iteração. As instruções if , elif e else executam blocos de código com base no fato de uma condição ser True ou False .

Os operadores in e not in verificam se uma string é ou não está em outra string e avaliam como True ou False de acordo.

Saber como programar lhe dá a capacidade de escrever um processo como criptografar ou descriptografar com a cifra de César em uma linguagem que um computador possa entender. E uma vez que o computador entenda como executar o processo, ele pode fazê-lo muito mais rápido do que qualquer ser humano e sem erros (a menos que haja erros em sua programação). Embora essa seja uma habilidade incrivelmente útil, a cifra de César pode ser facilmente quebrada por alguém que sabe programar. No [Capítulo 6](#) , você usará as habilidades que aprendeu para escrever um hacker de cifra de César para poder ler o texto cifrado que outras pessoas criptografaram. Vamos seguir em frente e aprender a invadir a criptografia.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. Usando *caesarCipher.py* , criptografe as seguintes sentenças com as chaves fornecidas:
 1. " Você pode mostrar que o preto é branco por discussão ', disse Filby,' mas você nunca vai me convencer .' Com a chave 8
 2. '1234567890' com chave 21
2. Usando *caesarCipher.py* , descriptografe os seguintes textos cifrados com as chaves fornecidas:
 1. 'Kv? Uqwpfu? Rncwukdng? GpqwijB' com chave 2

2. 'XCBSw88S18A1S 2SB41SE .8zSEwAS50D5A5x81V' com chave 22
3. Qual instrução Python importaria um módulo chamado *watermelon.py* ?
4. O que as seguintes partes do código exibem na tela?
 1. spam = 'foo'
para i em spam:
spam = spam + i
imprimir (spam)
 2. se 10 <5:
print ('Hello')
elif Falso:
print ('Alice')
elif 5! = 5:
print ('Bob')
outro:
print ('Adeus')
 3. print ('f' não está em 'foo')
 4. print ('foo' em 'f')
 5. print ('hello'.find (' oo '))

6

HACKING O CESAR DE CAESAR COM FORÇA-BRUTELA

“Estudiosos árabes. . . inventou criptoanálise, a ciência de desembaralhar uma mensagem sem o conhecimento da chave.”

- Simon Singh, *The Code Book*



Podemos cortar a cifra de César usando uma técnica criptoanalítica chamada *força bruta*. Um *ataque de força bruta* tenta todas as chaves de descriptografia possíveis para uma cifra. Nada impede um criptoanalista de adivinhar uma chave, descriptografar o texto cifrado com essa chave, examinar a saída e, em seguida, passar para a próxima chave se não encontrar a mensagem secreta. Como a técnica de força bruta é tão eficaz contra a cifra de César, você não deveria usar a cifra de César para criptografar informações secretas.

Idealmente, o texto cifrado nunca cairia nas mãos de ninguém. Mas o *princípio de Kerckhoffs* (em homenagem ao criptógrafo do século 19, Auguste Kerckhoffs), afirma que uma cifra ainda deve ser segura, mesmo que todos saibam como a cifra funciona e que outra pessoa tenha o texto cifrado. Esse princípio foi reafirmado pelo matemático do século 20 Claude Shannon como a *máxima de Shannon*: "O inimigo conhece o sistema". A parte da cifra que mantém a mensagem em segredo é a chave, e para a cifra de César essa informação é muito fácil de encontrar.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Princípio de Kerckhoff e a máxima de Shannon
- A técnica de força bruta
- A função range()
- Formatação de string (interpolação de string)

Código fonte do programa Cesar Cipher Hacker

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como *caesarHacker.py*. Em seguida, faça o download do módulo *pyperclip.py*, se ainda não o fez (<https://www.nostarch.com/crackingcodes/>), e coloque-o no mesmo diretório (ou seja, na mesma pasta) que o arquivo *caesarCipher.py*. Este módulo será importado pelo *caesarCipher.py*.

Quando terminar de configurar os arquivos, pressione F5 para executar o programa. Se você encontrar algum erro ou problema com seu código, poderá compará-lo ao código do livro usando a ferramenta de comparação on-line em <https://www.nostarch.com/crackingcodes/>.

caesarHacker.py

1. # César Cifra Hacker

```
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. message = 'guv6Jv6Jz! J6rp5r7Jzr66ntrM'
5. SÍMBOLOS =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
67890!
6
7. # Loop através de todas as chaves possíveis:
8. para chave no intervalo (len (SYMBOLS)):
9. # É importante definir a tradução para a string em branco para que o
10. O valor da iteração # anterior da tradução é limpo:
11. traduzido = "
12
13. # O resto do programa é quase o mesmo que o programa César:
14
15. # Loop através de cada símbolo na mensagem:
16. para o símbolo na mensagem:
17. se símbolo em SÍMBOLOS:
18. symbolIndex = SYMBOLS.find (símbolo)
19. translatedIndex = symbolIndex - chave
20
21. # Lide com o embrulho:
22. if translatedIndex <0:
23.     translatedIndex = translatedIndex + len (SÍMBOLOS)
24
25. # Anexar o símbolo descriptografado:
26. traduzido = traduzido + SYMBOLS [translatedIndex]
27
28. mais:
29. # Anexar o símbolo sem criptografar / descriptografar:
30. traduzido = traduzido + símbolo
31
32. # Exibe todas as descodificações possíveis:
33. print ('Chave %# s:% s' (chave, traduzida))
```

Observe que muito deste código é o mesmo que o código no programa de cifra original de César. Isso ocorre porque o programa hacker Cipher usa os mesmos passos para descriptografar a mensagem.

Execução de Amostra do Programa Craser Cipher Hacker

O programa hacker de cifra Caesar imprime a seguinte saída quando você a executa. Ele quebra o texto cifrado guv6Jv6Jz! J6rp5r7Jzr66ntrM descriptografando o texto cifrado com todas as 66 chaves possíveis:

Chave 0: guv6Jv6Jz! J6rp5r7Jzr66ntrM

Legenda 1: ftu5Iu5Iy I5qo4q6Iyq55msqL

Chave 2: est4Ht4Hx0H4pn3p5Hxp44lrpK

Chave 3: drs3Gs3Gw9G3om2o4Gwo33kqoJ

Legenda # 4: cqr2Fr2Fv8F2nl1n3Fvn22jpnI

--recorte--

Legenda # 11: Vjku? Ku? O1? Ugetgv? OguucigB

Legenda # 12: Uijt! Jt! Nz! Tfdsfu! NfttbhfA

Chave # 13: Esta é minha mensagem secreta.

Legenda # 14: Sghr0hr0lx0rdbqds0ldrrZfd?

Chave 15: Rfgq9gq9kw9qcapcr9kcqqYec!

--recorte--

Legenda # 61: lz1 O1 O5CO wu0w! O5w sywR

Legenda # 62: kyz0Nz0N4BN0vt9v N4v00rxvQ

Chave # 63: jxy9My9M3AM9us8u0M3u99qwuP

Chave # 64: iwx8Lx8L2.L8tr7t9L2t88pvtO

Chave 65: hvw7Kw7K1? K7sq6s8K1s77ousN

Como a saída descriptografada da chave 13 é em inglês simples, sabemos que a chave de criptografia original deve ter sido 13 .

Configurando Variáveis

O programa hacker criará uma variável de mensagem que armazena a string de texto cifrado que o programa tenta descriptografar. A variável constante SYMBOLS contém todos os caracteres que a cifra pode criptografar:

1. # César Cifra Hacker

2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)

3

4. message = 'guv6Jv6Jz! J6rp5r7Jzr66ntrM'

5. SYMBOLS =

'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890!'

O valor para SYMBOLS precisa ser o mesmo que o valor para SYMBOLS usado no programa de cifra de César que criptografou o texto cifrado que estamos tentando hackear; caso contrário, o programa hacker não funcionará. Note que existe um único espaço entre 0 e ! no valor da string.

Looping com a função range ()

A linha 8 é um loop for que não itera em um valor de string, mas, em vez disso, itera sobre o valor de retorno de uma chamada para a função range () :

7. # Loop através de todas as chaves possíveis:

8. para chave no intervalo (len (SYMBOLS)):

A função range () recebe um argumento inteiro e retorna um valor do tipo de dado do intervalo . Os valores do intervalo podem ser usados em loops para fazer um loop em um número específico de vezes, de acordo com o inteiro que você atribuiu à função. Vamos tentar um exemplo. Digite o seguinte no shell interativo:

```
>>> para i na faixa (3):
```

```
... print ('Olá')
```

```
...
```

```
Olá
```

```
Olá
```

```
Olá
```

O loop for fará um loop três vezes porque passamos o inteiro 3 para range () .

Mais especificamente, o valor do intervalo retornado da chamada da função range () irá definir a variável do loop for para os inteiros de 0 a (mas não incluindo) o argumento passado para range () . Por exemplo, insira o seguinte no shell interativo:

```
>>> para i na faixa (6):
```

```
... imprimir (i)
```

```
...
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

Este código define a variável `i` para os valores de 0 a (mas não incluindo) 6 , que é semelhante ao que a linha 8 em `caesarHacker.py` faz. A linha 8 define a variável de chave com os valores de 0 a (mas não incluindo) 66 . Em vez de codificar o valor 66 diretamente em nosso programa, usamos o valor de retorno de `len (SYMBOLS)` para que o programa ainda funcione se modificarmos `SYMBOLS` .

A primeira vez que a execução do programa passa por esse loop, a chave é definida como 0 e o texto cifrado na mensagem é descriptografado com a chave 0 . (Claro que, se 0 não é a chave real, a mensagem apenas “descriptografa” para absurdo.) O código dentro do loop for das linhas 9 a 31, que explicaremos a seguir, é semelhante ao programa original de cifra de César e a descriptografia. Na próxima iteração da linha 8 para loop, a tecla é definida como 1 para a descriptografia.

Apesar de não usá-lo neste programa, você também pode passar dois argumentos inteiros para a função `range ()` em vez de apenas um. O primeiro argumento é onde o intervalo deve começar, e o segundo argumento é onde o intervalo deve parar (até, mas não incluindo o segundo argumento). Os argumentos são separados por uma vírgula:

```
>>> para i na faixa (2, 6):
... imprimir (i)
...
2
3
4
5
```

A variável `i` levará o valor de 2 (incluindo 2) até o valor 6 (mas não incluindo 6)

.

Descriptografar a mensagem

O código de descriptografia nas próximas linhas adiciona o texto descriptografado ao final da string na tradução . Na linha 11, traduzido é definido como uma string em branco:

7. # Loop através de todas as chaves possíveis:
8. para chave no intervalo (`len (SYMBOLS)`):
9. # É importante definir a tradução para a string em branco para que o

10. O valor da iteração # anterior da tradução é limpo:

11. traduzido = "

É importante que redefinamos a tradução para uma string em branco no início desse loop; caso contrário, o texto que foi descriptografado com o A chave atual será adicionada ao texto descriptografado na tradução da última iteração no loop.

As linhas 16 a 30 são quase iguais ao código no programa de cifra de César no [Capítulo 5](#), mas são um pouco mais simples porque esse código só precisa descriptografar:

13. # O resto do programa é quase o mesmo que o programa César:

14

15. # Loop através de cada símbolo na mensagem:

16. para o símbolo na mensagem:

17. se símbolo em SÍMBOLOS:

18. symbolIndex = SYMBOLS.find (símbolo)

Na linha 16, percorremos todos os símbolos da string de texto cifrado armazenados na mensagem . Em cada iteração desse loop, a linha 17 verifica se o símbolo existe na variável constante SYMBOLS e, em caso afirmativo, descriptografa-o. A chamada do método find () da linha 18 localiza o índice onde o símbolo está em SYMBOLS e o armazena em uma variável chamada symbolIndex .

Então subtraímos a chave de symbolIndex na linha 19 para descriptografar:

19. translatedIndex = symbolIndex - chave

20

21. # Lide com o embrulho:

22. if translatedIndex <0:

23. translatedIndex = translatedIndex + len (SÍMBOLOS)

Essa operação de subtração pode fazer com que o translateIndex se torne menor que zero e nos obrigue a “envolver” a constante SYMBOLS quando encontrarmos a posição do caractere em SYMBOLS para descriptografar. A linha 22 verifica esse caso e a linha 23 adiciona 66 (que é o que len (SYMBOLS) retorna) se translateIndex for menor que 0 .

Agora que o translateIndex foi modificado, SYMBOLS [translatedIndex] irá avaliar o símbolo descriptografado. A linha 26 adiciona este símbolo ao final da string armazenada na tradução :

```
25. # Anexar o símbolo descriptografado:  
26. traduzido = traduzido + SYMBOLS [translatedIndex]  
27  
28. mais:  
29. # Anexar o símbolo sem criptografar / descriptografar:  
30. traduzido = traduzido + símbolo
```

A linha 30 apenas adiciona o símbolo não modificado ao final da tradução se o valor não foi encontrado no conjunto SYMBOL .

Usando formatação de seqüência de caracteres para exibir a chave e mensagens descriptografadas

Embora a linha 33 seja a única chamada de função print () em nosso programa de hackers Cesar cipher, ela executará várias linhas porque é chamada uma vez por iteração do loop for na linha 8:

```
32. # Exibe todas as descodificações possíveis:  
33. print ('Chave #% s:% s' % (chave, traduzida))
```

O argumento para a chamada de função print () é um valor de string que usa *formatação de string* (também chamada de *interpolação de string*). A formatação de string com o texto % s coloca uma string dentro de outra. O primeiro % s na string é substituído pelo primeiro valor entre parênteses no final da string.

Digite o seguinte no shell interativo:

```
>>> 'Olá% s!' % ('mundo')  
'Olá Mundo!'  
>>> 'Olá' + 'mundo' + '!'  
'Olá Mundo!'  
>>> '% s comeu% s que comeram% s' % ('cachorro', 'gato', 'rato')  
"O cachorro comeu o gato que comeu o rato."
```

Neste exemplo, primeiro a string 'world' é inserida na string 'Hello% s!' no lugar do % s . Funciona como se você tivesse concatenado a parte da string antes do % s com a string interpolada e a parte da string após o % s . Quando você interpola várias strings, elas substituem cada % s em ordem.

A formatação de strings geralmente é mais fácil de digitar do que a concatenação de strings usando o operador + , especialmente para strings grandes. E,

diferentemente da concatenação de strings, você pode inserir valores não-string, como inteiros, na string. Digite o seguinte no shell interativo:

```
>>> '% s teve% s tortas' % ('Alice', 42)
"Alice tinha 42 tortas".
>>> 'Alice' + 'tinha' + 42 + 'tortas'
Traceback (última chamada mais recente):
Arquivo "<stdin>", linha 1, em <module>
TypeError: Não é possível converter o objeto 'int' para str implicitamente
```

O inteiro 42 é inserido na string sem nenhum problema quando você usa a interpolação, mas quando você tenta concatenar o inteiro, isso causa um erro.

A linha 33 de *caesarHacker.py* usa a formatação de string para criar uma string que tenha os valores nas variáveis chave e traduzida . Como a chave armazena um valor inteiro, usamos a formatação de string para colocá-la em um valor de string que é passado para print () .

Resumo

A fraqueza crítica da cifra de César é que não há muitas chaves possíveis que podem ser usadas para criptografar. Qualquer computador pode facilmente descriptografar com todas as 66 chaves possíveis, e leva um criptoanalista apenas alguns segundos para examinar as mensagens descriptografadas para encontrar a palavra em inglês. Para tornar nossas mensagens mais seguras, precisamos de uma criptografia que tenha mais chaves em potencial. A cifra de transposição discutida no [Capítulo 7](#) pode fornecer essa segurança para nós.

PERGUNTA DE PRÁTICA

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. Quebre o seguinte texto cifrado, descriptografando uma linha por vez, porque cada linha tem uma chave diferente. Lembre-se de escapar de qualquer caractere de aspas:

qeFIP? eGSeECNNS,
5coOMXXcoPSZIWoQI,
avnl1olyD4l'y1Dohww6DhzDjhuDil,

z.GM?.cEQc. 70c.7KcKMKHA9AGFK,

MFYp2pPJJUpZSIJWpRdpMFY,
ZqH8sl5HtqHTH4s3lyvH5zH5spH4t pHqHlH3l5K

Zfbi,! Tif! Xpvme! Qspcbcmz! Fbu! NfA

7

CRIANDO COM A CIPHER DE TRANSPOSIÇÃO

"Argumentar que você não se importa com o direito à privacidade porque não tem nada a esconder não é diferente de dizer que não se importa com a liberdade de expressão porque não tem nada a dizer."

- Edward Snowden, 2015



A cifra de César não é segura; Não é preciso muito para um computador usar força bruta através de todas as 66 teclas possíveis. A cifra de transposição, por outro lado, é mais difícil para a força bruta porque o número de chaves possíveis depende do tamanho da mensagem. Existem muitos tipos diferentes de cifras de transposição, incluindo a cifra da cerca, a cifra da rota, a cifra de transposição de Myszkowski e a cifra de transposição interrompida. Este capítulo cobre uma simples cifra de transposição chamada de *cifra de transposição colunar*.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Criando funções com instruções def
- Argumentos e Parâmetros
- Variáveis em escopos globais e locais
- funções main ()
- O tipo de dados da lista
- Semelhanças em listas e strings
- Listas de listas
- Operadores de atribuição aumentada (+ =, - =, * =, / =)

- O método de string join ()
- Retornar valores e a declaração de retorno
- A variável __name__

Como funciona a cifra de transposição

Em vez de substituir caracteres por outros caracteres, a cifra de transposição reorganiza os símbolos da mensagem em uma ordem que torna a mensagem original ilegível. Como cada chave cria uma ordem ou *permutação diferente* dos caracteres, um criptoanalista não sabe como reorganizar o texto cifrado de volta na mensagem original.

As etapas para criptografar com a cifra de transposição são as seguintes:

1. Conte o número de caracteres na mensagem e na chave.
2. Desenhe uma linha de um número de caixas igual à chave (por exemplo, 8 caixas para uma chave de 8).
3. Comece a preencher as caixas da esquerda para a direita, inserindo um caractere por caixa.
4. Quando você ficar sem caixas, mas ainda tiver mais caracteres, adicione outra linha de caixas.
5. Quando você alcança o último caractere, sombreie as caixas não usadas na última linha.
6. Começando do canto superior esquerdo e indo para baixo de cada coluna, escreva os caracteres. Quando você chegar ao final de uma coluna, vá para a próxima coluna à direita. Ignore todas as caixas sombreadas. Este será o texto cifrado.

Para ver como essas etapas funcionam na prática, vamos criptografar uma mensagem à mão e depois traduzir o processo em um programa.

Criptografando uma Mensagem à Mão

Antes de começarmos a escrever código, vamos criptografar a mensagem “O senso comum não é tão comum”, usando lápis e papel. Incluindo os espaços e pontuação, esta mensagem tem 30 caracteres. Para este exemplo, você usará o número 8 como chave. O intervalo de possíveis chaves para esse tipo de criptografia é de 2 a metade do tamanho da mensagem, que é 15. Mas quanto mais longa a mensagem, mais chaves são possíveis. Criptografar um livro inteiro

usando a cifra de transposição colunar permitiria milhares de chaves possíveis.

O primeiro passo é desenhar oito caixas seguidas para corresponder ao número da chave, como mostra a [Figura 7-1](#) .

--	--	--	--	--	--	--	--

Figura 7-1: O número de caixas na primeira linha deve corresponder ao número da chave.

O segundo passo é começar a escrever a mensagem que você deseja criptografar nas caixas, colocando um caractere em cada caixa, como mostra a [Figura 7-2](#) . Lembre-se de que espaços também são caracteres (indicados aqui com ▪).

C	o	m	m	o	n	▪	s
---	---	---	---	---	---	---	---

Figura 7-2: Preencha um caractere por caixa, incluindo espaços.

Você tem apenas oito caixas, mas há 30 caracteres na mensagem. Quando você ficar sem caixas, desenhe outra linha de oito caixas abaixo da primeira linha. Continue criando novas linhas até ter escrito a mensagem inteira, conforme mostrado na [Figura 7-3](#) .

1 st	2 nd	3 rd	4 th	5 th	6 th	7 th	8 th
C	o	m	m	o	n	▪	s
e	n	s	e	▪	i	s	▪
n	o	t	▪	s	o	▪	c
o	m	m	o	n	▪		

Figura 7-3: Adicione mais linhas até que toda a mensagem seja preenchida.

Sobreie nas duas caixas na última linha como um lembrete para ignorá-las. O texto cifrado consiste nas letras lidas da caixa do canto superior esquerdo, descendo a coluna. C , e , n e o são da primeira coluna, conforme rotulado no diagrama. Quando você chegar à última linha de uma coluna, vá para a linha superior da próxima coluna à direita. Os próximos caracteres são o , n , o , m . Ignore as caixas sombreadas.

O texto cifrado é “Cenoonommstmme oo snnio. ssc ”, que é suficientemente embaralhado para impedir que alguém descubra a mensagem original olhando para ela.

Criando o programa de criptografia

Para criar um programa de criptografia, você precisa traduzir essas etapas de papel e lápis em código Python. Vamos ver novamente como criptografar a string "O senso comum não é tão comum". usando a tecla 8 . Para o Python, a posição de um caractere dentro de uma string é seu índice numerado, portanto, adicione o índice de cada letra da string às caixas no diagrama de criptografia original, como mostra a [Figura 7-4](#) . (Lembre-se que os índices começam com 0 , não 1.)

1st	2nd	3rd	4th	5th	6th	7th	8th
C 0	o 1	m 2	m 3	o 4	n 5	■ 6	s 7
e 8	n 9	s 10	e 11	■ 12	i 13	s 14	■ 15
n 16	o 17	t 18	■ 19	s 20	o 21	■ 22	c 23
o 24	m 25	m 26	o 27	n 28	■ 29		

Figura 7-4: Adicione o número do índice a cada caixa, começando com 0.

Essas caixas mostram que a primeira coluna tem os caracteres nos índices 0 , 8 , 16 e 24 (que são 'C' , 'e' , 'n' e 'o'). A próxima coluna tem os caracteres nos índices 1 , 9 , 17 e 25 (que são 'o' , 'n' , 'o' e 'm'). Observe o padrão emergente: a enésima coluna tem todos os caracteres da string nos índices $0 + (n - 1)$, $8 + (n - 1)$, $16 + (n - 1)$ e $24 + (n - 1)$, conforme mostrado na [Figura 7-5](#) .

1st	2nd	3rd	4th	5th	6th	7th	8th
C 0+0=0	o 1+0=1	m 2+0=2	m 3+0=3	o 4+0=4	n 5+0=5	■ 6+0=6	s 7+0=7
e 0+8=8	n 1+8=9	s 2+8=10	e 3+8=11	■ 4+8=12	i 5+8=13	s 6+8=14	■ 7+8=15
n 0+16=16	o 1+16=17	t 2+16=18	■ 3+16=19	s 4+16=20	o 5+16=21	■ 6+16=22	c 7+16=23
o 0+24=24	m 1+24=25	m 2+24=26	o 3+24=27	n 4+24=28	■ 5+24=29		

Figura 7-5: O índice de cada caixa segue um padrão previsível.

Há uma exceção para a última linha nas colunas 7 e 8, porque $24 + (7 - 1)$ e $24 + (8 - 1)$ seriam maiores que 29, que é o maior índice da string. Nesses casos, você adiciona apenas 0, 8 e 16 a n (e pula 24).

O que há de tão especial nos números 0, 8, 16 e 24? Estes são os números que você recebe quando, a partir de 0, você adiciona a chave (que neste exemplo é

8). Então, $0 + 8$ é 8 , $8 + 8$ é 16 , $16 + 8$ é 24 . O resultado de $24 + 8$ seria 32 , mas como 32 é maior que o comprimento da mensagem, você irá parar em 24 .

Para a sequência da coluna n , comece no índice ($n - 1$) e continue adicionando 8 (a chave) para obter o próximo índice. Continue adicionando 8 , desde que o índice seja menor que 30 (o tamanho da mensagem), e então passe para a próxima coluna.

Se você imaginar que cada coluna é uma string, o resultado seria uma lista de oito strings, assim: 'Ceno' , 'onom' , 'mstm' , 'eu o' , 'o sn' , 'nio' . 's' , 's c' . Se você concatenar as cordas juntas em ordem, o resultado seria o texto cifrado: 'Cenoonommstmme oo snnio. ss c' . Você aprenderá sobre um conceito chamado *listas* mais adiante no capítulo que permitirá que você faça exatamente isso.

Código-fonte para o programa de criptografia de criptografia de transposição

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File** . Digite o seguinte código no editor de arquivos e salve-o como *transpositionEncrypt.py* . Lembre-se de colocar o módulo *pyperclip.py* no mesmo diretório que o arquivo *transpositionEncrypt.py* . Em seguida, pressione F5 para executar o programa.

*transposição
Encrypt.py*

```
1. # Criptografia de criptografia de transposição
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip
5
6. def main():
7.     myMessage = 'O senso comum não é tão comum.'
8.     myKey = 8
9
10.    ciphertext = encryptMessage (myKey, myMessage)
11
12.    # Imprime a string criptografada em texto cifrado para a tela, com
13.    # a | (caractere "pipe") depois, caso haja espaços no
14.    # o final da mensagem criptografada:
```

```
15. print (texto cifrado + '|')
16
17. # Copie a string criptografada em texto cifrado para a área de transferência:
18. pyperclip.copy (texto cifrado)
19
20
21. def encryptMessage (chave, mensagem):
22. # Cada string no texto cifrado representa uma coluna na grade:
23. ciphertext = [""] * chave
24
25. # Loop através de cada coluna no texto cifrado:
26. para coluna no intervalo (chave):
27. currentIndex = coluna
28.
29. # Manter o loop até currentIndex ultrapassar o tamanho da mensagem:
30. while currentIndex <len (mensagem):
31. # Coloque o caractere em currentIndex em mensagem no
32. # fim da coluna atual na lista de texto cifrado:
33. texto cifrado [coluna] += mensagem [currentIndex]
34
35. # Mover currentIndex sobre:
36. currentIndex += chave
37
38. # Converta a lista de texto cifrado em um único valor de string e retorne-o:
39. return " .join (texto cifrado)
40.
41.
42. # Se transpositionEncrypt.py for executado (em vez de importado como um
módulo), ligue
43. # a função main ():
44. se __name__ == '__main__':
45. main ()
```

Execução de amostra do programa de criptografia de criptografia de transposição

Quando você executa o programa *transpositionEncrypt.py* , ele produz esta saída:

Cenoonommstmme oo snnio. ssc |

O caractere de pipe vertical (|) marca o final do texto cifrado no caso de existirem espaços no final. Este texto cifrado (sem o caractere pipe no final) também é copiado para a área de transferência, para que você possa colá-lo em um email para alguém. Se você deseja criptografar uma mensagem diferente ou usar uma chave diferente, altere o valor atribuído às variáveis myMessage e myKey nas linhas 7 e 8. Em seguida, execute o programa novamente.

Criando suas próprias funções com instruções de defesa

Depois de importar o módulo pyperclip , você usará uma instrução def para criar uma função personalizada, main () , na linha 6.

1. # Criptografia de criptografia de transposição
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar pyperclip
6. def main ():
7. myMessage = 'O senso comum não é tão comum.'
8. myKey = 8

A declaração def significa que você está criando ou *definindo* uma nova função que pode ser chamada posteriormente no programa. O bloco de código após o def statement é o código que será executado quando a função for chamada.

Quando você *chama* esta função, a execução se move dentro do bloco de código seguindo a instrução def da função.

Como você aprendeu no [Capítulo 3](#) , em alguns casos, as funções aceitarão *argumentos* , que são valores que a função pode usar com seu código. Por exemplo, print () pode pegar um valor de string como um argumento entre seus parênteses. Quando você define uma função que recebe argumentos, você coloca um nome de variável entre seus parênteses em sua instrução def . Essas variáveis são chamadas de *parâmetros* . A função main () definida aqui não possui parâmetros, portanto, não recebe argumentos quando é chamada. Se você tentar chamar uma função com muitos ou poucos argumentos para o número de parâmetros que a função possui, o Python exibirá uma mensagem de erro.

Definindo uma Função que Leva Argumentos com Parâmetros

Vamos criar uma função com um parâmetro e, em seguida, chamá-lo com um argumento. Abra uma nova janela do editor de arquivos e insira o seguinte

código:

Olá
Function.py

```
❶ def oi (nome):  
❷     print ('Olá,' + nome)  
❸     print ('Start.')  
❹     ola ('Alice')  
❺     print ('Chame a função novamente:')  
❻     ola ('Bob')  
❼     print ('Feito')
```

Salve este programa como *helloFunction.py* e execute-o pressionando F5. A saída é assim:

Começar.
Olá Alice
Chame a função novamente:
Olá, bob
Feito.

Quando o programa *helloFunction.py* é executado, a execução começa no topo. A instrução `def` define a função `hello ()` com um parâmetro, que é o nome da variável. A execução pula o bloco após a instrução `def` **❷** porque o bloco só é executado quando a função é chamada. Em seguida, ele executa `print ('Start.')`. Which , e é por isso que 'Start'. é a primeira string impressa quando você executa o programa.

A próxima linha após impressão ('Start.') É a primeira chamada de função para `hello ()` . A execução do programa pula para a primeira linha no bloco `function` da função `hello ()` . A string 'Alice' é passada como argumento e é atribuída ao nome do parâmetro. Essa chamada de função imprime a string 'Hello, Alice' na tela.

Quando a execução do programa atinge a parte inferior do bloco da declaração `def` , a execução volta para a linha com a chamada de função **❹** e continua executando o código a partir daí, então 'Chame a função novamente:' é impressa **❺** .

Em seguida é uma segunda chamada para `hello ()` **❻** . A execução do programa volta para a definição da função `hello ()` exec e executa o código novamente,

exibindo 'Hello, Bob' na tela. Em seguida, a função retorna e a execução vai para a próxima linha, que é a instrução print ('Done.') 7 , e a executa. Esta é a última linha do programa, portanto, o programa sai.

Alterações nos parâmetros existem somente dentro da função

Digite o seguinte código no shell interativo. Este código define e, em seguida, chama uma função chamada func () . Observe que o shell interativo requer que você insira uma linha em branco após o param = 42 para fechar o bloco da instrução def :

```
>>> def func (param):  
    param = 42
```

```
>>> spam = 'Olá'  
>>> func (spam)  
>>> imprimir (spam)  
Olá
```

A função func () usa um parâmetro chamado param e define seu valor como 42 . O código fora da função cria uma variável de spam e a define como um valor de string, e então a função é chamada de spam e o spam é impresso.

Quando você executa este programa, a chamada print () na última linha imprime "Hello" , não 42 . Quando func () é chamado com spam como argumento, apenas o valor dentro do spam é copiado e atribuído ao param . Qualquer alteração feita no param dentro da função *não* alterará o valor na variável spam . (Há uma exceção a essa regra quando você está passando uma lista ou um valor de dicionário, mas isso é explicado em “ [Listar variáveis usar referências](#) ” na [página 119](#)).

Toda vez que uma função é chamada, um *escopo local* é criado. Variáveis criadas durante uma chamada de função existem neste escopo local e são chamadas de *variáveis locais* . Os parâmetros sempre existem em um escopo local (eles são criados e recebem um valor quando a função é chamada). Pense em um *escopo* como um contêiner no qual as variáveis existem. Quando a função retorna, o escopo local é destruído e as variáveis locais contidas no escopo são esquecidas.

Variáveis criadas fora de cada função existem no *escopo global* e são chamadas de *variáveis globais* . Quando o programa sai, o escopo global é destruído e

todas as variáveis no programa são esquecidas. (Todas as variáveis nos programas de codificação reversa e cifra de César nos [Capítulos 5 e 6](#), respectivamente, eram globais.)

Uma variável deve ser local ou global; não pode ser ambos. Duas variáveis diferentes podem ter o mesmo nome, desde que estejam em escopos diferentes. Eles ainda são considerados duas variáveis diferentes, semelhantes a como a Main Street em San Francisco é uma rua diferente da Main Street em Birmingham.

A ideia importante a entender é que o valor do argumento que é “passado” para uma chamada de função é *copiado* para o parâmetro. Portanto, mesmo que o parâmetro seja alterado, a variável que forneceu o valor do argumento não é alterada.

Definindo a função main ()

Nas linhas 6 a 8 em *transpositionEncrypt.py*, você pode ver que definimos uma função main () que irá definir valores para as variáveis myMessage e myKey quando chamado:

```
6. def main ():  
7.     myMessage = 'O senso comum não é tão comum.'  
8.     myKey = 8
```

O restante dos programas neste livro também terá uma função chamada main () que é chamada no início de cada programa. A razão pela qual temos uma função main () é explicada no final deste capítulo, mas por enquanto só sabemos que main () é sempre chamado logo depois que os programas neste livro são executados.

As linhas 7 e 8 são as duas primeiras linhas no bloco de código que define main (). Nessas linhas, as variáveis myMessage e myKey armazenam a mensagem de texto sem formatação para criptografar e a chave usada para fazer a criptografia. A linha 9 é uma linha em branco, mas ainda faz parte do bloco e separa as linhas 7 e 8 da linha 10 para tornar o código mais legível. A linha 10 atribui o texto cifrado variável como a mensagem criptografada chamando uma função que recebe dois argumentos:

```
10. ciphertext = encryptMessage (myKey, myMessage)
```

O código que faz a criptografia real está na função encryptMessage () definida posteriormente na linha 21. Essa função recebe dois argumentos: um valor

inteiro para a chave e um valor de sequência para a mensagem criptografar. Neste caso, passamos as variáveis myMessage e myKey , que acabamos de definir nas linhas 7 e 8. Ao passar vários argumentos para uma chamada de função, separe os argumentos com uma vírgula.

O valor de retorno de encryptMessage () é um valor de string do texto cifrado criptografado. Esta string é armazenada em texto cifrado .

A mensagem de texto cifrado é impressa na tela na linha 15 e copiada para a área de transferência na linha 18:

```
12. # Imprime a string criptografada em texto cifrado para a tela, com  
13. # a | (caractere "pipe") depois, caso haja espaços no  
14. # o final da mensagem criptografada:  
15. print (texto cifrado + '|')  
16  
17. # Copie a string criptografada em texto cifrado para a área de transferência:  
18. pyperclip.copy (texto cifrado)
```

O programa imprime um caractere de pipe (|) no final da mensagem para que o usuário possa ver qualquer caractere de espaço vazio no final do texto cifrado.

A linha 18 é a última linha da função main () . Após a execução, a execução do programa retorna para a linha após a linha que o chamou.

Passando a chave e mensagem como argumentos

As variáveis key e message entre os parênteses na linha 21 são parâmetros:

```
21. def encryptMessage (chave, mensagem):
```

Quando a função encryptMessage () é chamada na linha 10, dois valores de argumento são passados (os valores em myKey e myMessage). Esses valores são atribuídos à chave de parâmetros e à mensagem quando a execução se move para o topo da função.

Você pode se perguntar por que você tem os parâmetros key e message , já que você já tem as variáveis myKey e myMessage na função main () . Precisamos de variáveis diferentes porque myKey e myMessage estão no escopo local da função main () e não podem ser usadas fora do main () .

O tipo de dados da lista

A linha 23 no programa *transpositionEncrypt.py* usa um tipo de dados chamado

de *lista* :

22. # Cada string no texto cifrado representa uma coluna na grade:
23. ciphertext = [""] * chave

Antes de prosseguirmos, você precisa entender como as listas funcionam e o que você pode fazer com elas. Um valor de lista pode conter outros valores.

Semelhante a como as cadeias começam e terminam com aspas, um valor de lista começa com um colchete aberto [e termina com um colchete fechado]. Os valores armazenados na lista estão entre os colchetes. Se mais de um valor estiver na lista, os valores serão separados por vírgulas.

Para ver uma lista em ação, insira o seguinte no shell interativo:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animais
['aardvark', 'tamanduá', 'antílope', 'albert']
```

A variável *animals* armazena um valor de lista e, nesse valor de lista, há quatro valores de string. Os valores individuais dentro de uma lista também são chamados de *itens* ou *elementos*. As listas são ideais para usar quando você precisa armazenar vários valores em uma variável.

Muitas das operações que você pode fazer com strings também funcionam com listas. Por exemplo, a indexação e o fatiamento funcionam em valores de lista da mesma maneira que eles trabalham em valores de string. Em vez de caracteres individuais em uma string, o índice se refere a um item em uma lista. Digite o seguinte no shell interativo:

```
>>> animals = ['aardvark', 'tamanduá', 'albert']
❶ >>> animais [0]
'aardvark'
>>> animais [1]
'tamanduá'
>>> animais [2]
'albert'
❷ >>> animais [1: 3]
['tamanduá', 'albert']
```

Tenha em mente que o primeiro índice é 0 , não 1 ❶ . Semelhante ao modo como o uso de fatias com uma string fornece uma nova string que faz parte da string original, o uso de fatias com uma lista fornece uma lista que faz parte da

lista original. E lembre-se de que, se uma fatia tiver um segundo índice, a fatia só subirá , mas não incluirá o item no segundo índice ❷ .

Um loop for também pode iterar sobre os valores em uma lista, assim como pode iterar sobre os caracteres em uma string. O valor armazenado na variável do loop for é um valor único da lista. Digite o seguinte no shell interativo:

```
>>> para spam em ['aardvark', 'tamanduá', 'albert']:  
... print ('Para o jantar estamos cozinhando' + spam)
```

...

```
Para o jantar estamos cozinhando aardvark  
Para o jantar estamos cozinhando tamanduá  
Para o jantar nós estamos cozinhando albert
```

Cada vez que o loop é iterado, a variável spam recebe um novo valor da lista, começando com o índice 0 da lista até o final da lista.

Reatribuindo os itens nas listas

Você também pode modificar os itens dentro de uma lista usando o índice da lista com uma instrução de atribuição normal. Digite o seguinte no shell interativo:

```
>>> animals = ['aardvark', 'tamanduá', 'albert']  
❶ >>> animais [2] = 9999  
>>> animais  
❷ ['aardvark', 'tamanduá', 9999]
```

Para modificar o terceiro membro da lista de animais , usamos o índice para obter o terceiro valor com animais [2] e, em seguida, usamos uma instrução de atribuição para alterar seu valor de 'albert' para o valor 9999 ❶ . Quando verificamos o conteúdo da lista novamente, 'albert' não está mais na lista ❷ .

REAGENDO PERSONAGENS EM CORDAS

Embora você possa reatribuir itens em uma lista, não é possível reatribuir um caractere em um valor de sequência. Digite o seguinte código no shell interativo:

```
>>> 'Olá, mundo!' [6] = 'X'
```

Você verá o seguinte erro:

Traceback (última chamada mais recente):
Arquivo <pyshell # 0>, linha 1, em <module>

'Olá, mundo!' [6] = 'X'

TypeError: o objeto 'str' não suporta a atribuição de itens

A razão pela qual você vê esse erro é que o Python não permite usar instruções de atribuição no valor de índice de uma cadeia de caracteres. Em vez disso, para alterar um caractere em uma string, você precisa criar uma nova string usando fatias. Digite o seguinte no shell interativo:

```
>>> spam = 'Olá, mundo!'
>>> spam = spam [: 6] + 'X' + spam [7:]
>>> spam
'Olá Xorld!'
```

Você primeiro pegaria uma fatia que começa no início da string e vai até o personagem para mudar. Então você poderia concatenar isso com a string do novo caractere e uma fatia do caractere após o novo caractere até o final da string. Isso resulta na seqüência original com apenas um caractere alterado.

Listas de Listas

Valores de lista podem conter outras listas. Digite o seguinte no shell interativo:

```
>>> spam = [['dog', 'cat'], [1, 2, 3]]
>>> spam [0]
['cachorro gato']
>>> spam [0] [0]
'cão'
>>> spam [0] [1]
'gato'
>>> spam [1] [0]
1
>>> spam [1] [1]
2
```

O valor do spam [0] é avaliado na lista ['dog', 'cat'] , que possui seus próprios índices. Os colchetes de índice duplo usados para spam [0] [0] indicam que estamos pegando o primeiro item da primeira lista: spam [0] é avaliado como ['dog', 'cat'] e ['dog', 'cat '] [0] avalia para ' cachorro ' .

Usando len () e o operador in com listas

Você usou len () para indicar o número de caracteres em uma string (isto é, o comprimento da string). A função len () também funciona em valores de lista e

retorna um número inteiro de itens em uma lista.

Digite o seguinte no shell interativo:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> len (animais)
4
```

Da mesma forma, você usou os operadores in e não in para indicar se existe uma string dentro de outro valor de string. O operador in também funciona para verificar se existe um valor em uma lista, e o operador não em verifica se um valor não existe em uma lista. Digite o seguinte no shell interativo:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'tamanduá' em animais
Verdade
>>> 'tamanduá' não em animais
Falso
❶ >>> 'anteat' em animais
Falso
❷ >>> 'anteater' em animais [1]
Verdade
>>> 'delicioso spam' em animais
Falso
```

Por que a expressão em ❶ retorna False enquanto a expressão em ❷ retorna True ? Lembre-se de que animals é um valor de lista, enquanto animals [1] avalia o valor da string 'anteater' . A expressão em ❶ é avaliada como False porque a string 'anteat' não existe na lista de animais . No entanto, a expressão em ❷ é avaliada como True porque animals [1] é a string 'tamanduá' e 'anteat' existe nessa sequência.

Semelhante a como um conjunto de aspas vazias representa um valor de string em branco, um conjunto de colchetes vazios representa uma lista em branco.

Digite o seguinte no shell interativo:

```
>>> animals = []
>>> len (animais)
0
```

A lista de animais está vazia, então seu comprimento é 0 .

Listar concatenação e replicação com os operadores + e *

Você sabe que os operadores + e * podem concatenar e replicar cadeias de caracteres; os mesmos operadores também podem concatenar e replicar listas. Digite o seguinte no shell interativo.

```
>>> ['olá'] + ['mundo']
['Olá Mundo']
>>> ['olá'] * 5
['Ola Ola Ola Ola Ola']
```

Isso é o suficiente sobre as semelhanças entre strings e listas. Lembre-se de que a maioria das operações que você pode fazer com valores de string também funciona com valores de lista.

O Algoritmo de Criptografia de Transposição

Vamos usar listas em nosso algoritmo de criptografia para criar nosso texto cifrado. Vamos retornar ao código no programa *transpositionEncrypt.py*. Na linha 23, que vimos anteriormente, a variável de texto cifrado é uma lista de valores de cadeia vazia:

```
22. # Cada string no texto cifrado representa uma coluna na grade:
23. ciphertext = [""] * chave
```

Cada string na variável de texto cifrado representa uma coluna da grade da cifra de transposição. Como o número de colunas é igual à chave, você pode usar a replicação de lista para multiplicar uma lista com um valor de sequência em branco pelo valor na chave. É assim que a linha 23 é avaliada em uma lista com o número correto de strings em branco. Os valores da string serão atribuídos a todos os caracteres que entram em uma coluna da grade. O resultado será uma lista de valores de string que representam cada coluna, conforme discutido anteriormente no capítulo. Como os índices de lista começam com 0, você precisará rotular cada coluna a partir de 0. Assim, o texto cifrado [0] é a coluna mais à esquerda, o texto cifrado [1] é a coluna à direita e assim por diante.

Para ver como isso funcionaria, vamos examinar novamente a grade a partir do exemplo “Senso comum não é tão comum.” A partir do início deste capítulo (com números de colunas correspondentes aos índices da lista adicionados ao topo), como mostrado na [Figura 7 -6](#).

0	1	2	3	4	5	6	7
C	o	m	m	o	n	■	s
e	n	s	e	■	i	s	■
n	o	t	■	s	o	■	c
o	m	m	o	n	.		

Figura 7-6: A grade de mensagens de exemplo com índices de lista para cada coluna

Se atribuirmos manualmente os valores de string à variável de texto cifrado para essa grade, seria assim:

```
>>> texto_cifrado = ['Ceno', 'onom', 'mstm', 'eu o', 'o sn', 'nio', 's', 's c']
```

```
>>> texto_cifrado [0]
```

```
'Ceno'
```

A próxima etapa adiciona texto a cada string no texto cifrado , como acabamos de fazer no exemplo do manual, exceto que desta vez adicionamos algum código para fazer o computador fazer isso programaticamente:

25. # Loop através de cada coluna no texto cifrado:

26. para coluna no intervalo (chave):

27. currentIndex = coluna

O loop for na linha 26 itera uma vez para cada coluna e a variável de coluna tem o valor inteiro a ser usado para o índice para o texto cifrado . Na primeira iteração através do loop for , a variável de coluna é definida como 0 ; na segunda iteração, é definido como 1 ; depois 2 ; e assim por diante. Temos o índice para os valores de string no texto cifrado que queremos acessar mais tarde usando a expressão ciphertext [column] .

Enquanto isso, a variável currentIndex mantém o índice da string de mensagem que o programa examina em cada iteração do loop for . Em cada iteração através do loop, a linha 27 configura currentIndex para o mesmo valor da coluna . Em seguida, criaremos o texto cifrado ao concatenar a mensagem embaralhada em conjunto, um caractere por vez.

Operadores de atribuição aumentada

Até agora, quando concatenamos ou adicionamos valores um ao outro, usamos o operador + para adicionar o novo valor à variável. Muitas vezes, quando você

está atribuindo um novo valor a uma variável, você quer que ela seja baseada no valor atual da variável, então você usa a variável como a parte da expressão que é avaliada e atribuída à variável, como neste exemplo no shell interativo:

```
>>> spam = 40
>>> spam = spam + 2
>>> imprimir (spam)
42
```

Existem outras maneiras de manipular valores em variáveis com base no valor atual da variável. Por exemplo, você pode fazer isso usando *operadores de atribuição aumentados*. A instrução `spam += 2`, que usa o operador de atribuição `+=` aumentada, faz o *mesmo que* `spam = spam + 2`. É um pouco mais curto para digitar. O operador `+=` trabalha com inteiros para fazer adição, strings para fazer concatenação de strings e listas para fazer a concatenação de listas. [A Tabela 7-1](#) mostra os operadores de designação aumentada e suas instruções de atribuição equivalentes.

Tabela 7-1: Operadores de atribuição aumentada

Atribuição aumentada	Atribuição normal equivalente
<code>spam += 42</code>	<code>spam = spam + 42</code>
<code>spam -= 42</code>	<code>spam = spam - 42</code>
<code>spam *= 42</code>	<code>spam = spam * 42</code>
<code>spam /= 42</code>	<code>spam = spam / 42</code>

Usaremos operadores de atribuição aumentados para concatenar caracteres ao nosso texto cifrado.

Movendo currentIndex Através da Mensagem

A variável `currentIndex` mantém o índice do próximo caractere na sequência de mensagens que será concatenada às listas de texto cifrado. A chave é adicionada ao `currentIndex` em cada iteração da linha 30 enquanto o loop aponta para diferentes caracteres na mensagem e, a cada iteração da linha 26 para loop, `currentIndex` é definido como o valor na variável da coluna .

Para embaralhar a string na variável de mensagem , precisamos pegar o primeiro caractere da mensagem , que é 'C' , e colocá-lo na primeira string de texto cifrado . Então, nós pulamos oito caracteres para a mensagem (porque a chave é igual a 8) e concatenamos esse caractere, que é 'e' , para a primeira string do texto cifrado. Continuaríamos pulando caracteres de acordo com a chave e concatenando cada caractere até chegarmos ao final da mensagem. Isso criaria a string 'Ceno' , que é a primeira coluna do texto cifrado. Então, faríamos isso de novo, mas começamos no segundo caractere da mensagem para criar a segunda coluna.

Dentro do loop for que inicia na linha 26 é um loop while que inicia na linha 30. Esse loop while localiza e concatena o caractere correto na mensagem para criar cada coluna. Faz um loop enquanto currentIndex é menor que o tamanho da mensagem :

```
29. # Manter o loop até currentIndex ultrapassar o tamanho da mensagem:  
30. while currentIndex < len (mensagem):  
31. # Coloque o caractere em currentIndex em mensagem no  
32. # fim da coluna atual na lista de texto cifrado:  
33. texto cifrado [coluna] += mensagem [currentIndex]  
34  
35. # Mover currentIndex sobre:  
36. currentIndex += chave
```

Para cada coluna, o loop while percorre a variável de mensagem original e seleciona caracteres em intervalos de chave , adicionando a chave ao currentIndex . Na linha 27 para a primeira iteração do loop for , currentIndex foi configurado para o valor da coluna , que inicia em 0 .

Como você pode ver na [Figura 7-7](#) , a mensagem [currentIndex] é o primeiro caractere da mensagem em sua primeira iteração. O caractere na mensagem [currentIndex] é concatenado ao texto cifrado [coluna] para iniciar a primeira coluna na linha 33. A linha 36 adiciona o valor na chave (que é 8) ao currentIndex cada vez que passa pelo loop. A primeira vez é a mensagem [0] , a segunda mensagem de tempo [8] , a terceira mensagem de tempo [16] e a quarta mensagem de tempo [24] .

1st	2nd	3rd	4th
c o m m o n s e n s e i s n o t s o c o m m o n .			
0 1 2 3 4 5 6 7 8 9 1 0 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 0 1 2 2 2 3 4 5 6 7 8 9			

Figura 7-7: setas apontando para qual mensagem [currentIndex] se refere durante a primeira iteração do loop for quando a coluna é definida como 0

Embora o valor em currentIndex seja menor que o comprimento da sequência de mensagens , você deseja continuar concatenando o caractere em message [currentIndex] para o final da sequência no índice da coluna em texto cifrado . Quando currentIndex é maior que o tamanho da mensagem , a execução sai do loop while e retorna ao loop for . Como não há código no bloco for após o loop while, o loop for itera, a coluna é definida como 1 e currentIndex inicia com o mesmo valor da coluna .

Agora, quando a linha 36 adiciona 8 ao currentIndex em cada iteração da linha 30 durante o loop, os índices serão 1 , 9 , 17 e 25 , conforme mostrado na [Figura 7-8](#) .

1st 5th	2nd 6th	3rd 7th	4th 8th
C o m m o n s e n s e i s n o t s o c o m m o n .			
0 1 2 3 4 5 6 7 8 9 1 0 1 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 0 1 2 2 2 3 4 5 6 7 8 9			

Figura 7-8: setas apontando para qual mensagem [currentIndex] se refere durante a segunda iteração do loop for quando a coluna é definida como 1

Como mensagem [1] , mensagem [9] , mensagem [17] e mensagem [25] são concatenadas ao final do texto cifrado [1] , elas formam a cadeia 'onom' . Esta é a segunda coluna da grade.

Quando o loop for terminou de percorrer o resto das colunas, o valor no texto cifrado é ['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', 'S' é c '] . Depois de termos a lista de colunas de strings, precisamos juntá-las para fazer uma string que é o texto cifrado inteiro: 'Cenoonommstmme oo snnio. ss c ' .

O método de string join ()

O método join () é usado na linha 39 para unir as strings de coluna individuais do texto cifrado em uma string. O método join () é chamado em um valor de string e recebe uma lista de strings. Ele retorna uma string que tem todos os

membros na lista unida pela string que join () é chamada. (Esta é uma string em branco se você quiser apenas unir as strings.) Digite o seguinte no shell interativo:

```
>>> ovos = ['cachorros', 'gatos', 'alce']  
❶ >>> ''.join(ovos)  
'dogscatsmooose'  
❷ >>> ',' .join(ovos)  
'cachorros, gatos, alces '  
❸ >>> 'XYZ'.join(ovos)  
'dogsXYZcatsXYZmooose'
```

Quando você chama join () em uma string vazia e se junta à lista eggs ❶ , você obtém as strings da lista concatenadas sem nenhuma string entre elas. Em alguns casos, você pode querer separar cada membro em uma lista para torná-lo mais legível, o que fizemos em ❷ chamando join () na string ','. Isso insere a string ',' entre cada membro da lista. Você pode inserir qualquer string que quiser entre os membros da lista, como você pode ver em ❸ .

Valores de retorno e declarações de retorno

Uma chamada de função (ou método) sempre é avaliada como um valor. Este é o valor *retornado* pela função ou chamada de método, também chamado de *valor de retorno* da função. Quando você cria suas próprias funções usando uma instrução def , uma instrução de retorno informa ao Python qual é o valor de retorno da função. A linha 39 é uma declaração de retorno :

```
38. # Converta a lista de texto cifrado em um único valor de cadeia e retorne-a:  
39. return " ".join(ciphertext)
```

A linha 39 chama join () na string em branco e passa o texto cifrado como o argumento para que as strings na lista de texto cifrado sejam unidas em uma única string.

Um exemplo de instrução de retorno

Uma instrução de retorno é a palavra-chave return seguida do valor a ser retornado. Você pode usar uma expressão em vez de um valor, como na linha 39. Quando você faz isso, o valor de retorno é o que a expressão avaliar. Abra uma nova janela do editor de arquivos, insira o programa a seguir, salve-o como addNumbers.py e pressione F5 para executá-lo:

addNumbers.py

```
1. def addNumbers (a, b):  
2.     return a + b  
3  
4. imprimir (addNumbers (2, 40))
```

Quando você executa o programa *addNumbers.py* , esta é a saída:

42

Isso porque a chamada de função *addNumbers (2, 40)* na linha 4 é avaliada como 42 . A instrução de retorno em *addNumbers ()* na linha 2 avalia a expressão *a + b* retorna o valor avaliado.

Retornando o texto cifrado encriptado

No programa *transpositionEncrypt.py* , a instrução de retorno da função *encryptMessage ()* retorna um valor de string que é criado unindo todas as strings na lista de texto cifrado . Se a lista no texto cifrado for ['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio', 's', 's c'] , a chamada do método *join ()* retornará 'Cenoonommstmme oo snnio. ss c ' . Essa string final, o resultado do código de criptografia, é retornada pela nossa função *encryptMessage ()* .

A grande vantagem de usar funções é que um programador precisa saber o que a função faz, mas não precisa saber como o código da função funciona. Um programador pode entender que, quando chamam a função *encryptMessage ()* e a transmitem um inteiro, bem como uma string para os parâmetros *key* e *message* , a chamada de função é avaliada como uma string criptografada. Eles não precisam saber nada sobre como o código em *encryptMessage ()* realmente faz isso, que é semelhante a como você sabe que quando você passa uma string para *print ()* , ela imprime a string mesmo que você nunca tenha visto o código da função *print ()* .

A variável __name__

Você pode transformar o programa de criptografia de transposição em um módulo usando um truque especial envolvendo a função *main ()* e uma variável chamada __name__ .

Quando você executa um programa em Python, __name__ (dois sublinhados antes do nome e dois sublinhados após) recebe o valor da string '__main__' (novamente, dois sublinhados antes e depois do *main*) mesmo antes da primeira linha do seu programa ser executada. O sublinhado duplo é geralmente chamado de *dunder* em Python e __main__ é chamado de *dunder principal dunder* .

No final do arquivo de script (e, mais importante, depois de todas as instruções def), você deseja ter algum código que verifique se a variável `__name__` tem a string '`__main__`' atribuída a ele. Nesse caso, você quer chamar a função `main ()`.

A instrução if na linha 44 é, na verdade, uma das primeiras linhas de código executadas quando você executa o programa (após a declaração de importação na linha 4 e as instruções def nas linhas 6 e 21).

```
42. # Se transpositionEncrypt.py for executado (ao invés de importado como um módulo) chame  
43. # a função main ():  
44. if __name__ == '__main__':  
45.     main ()
```

A razão pela qual o código é configurado desta forma é que, embora o Python configure `__name__` para '`__main__`' quando o programa é executado, ele o configura para a string '`transpositionEncrypt`' se o programa for importado por outro programa Python. Semelhante a como o programa importa o módulo `pyperclip` para chamar as funções nele, outros programas podem querer importar `transpositionEncrypt.py` para chamar sua função `encryptMessage ()` sem a função `main ()` em execução. Quando uma declaração de importação é executada, o Python procura o arquivo do módulo adicionando. `.py` para o final do nome do arquivo (é por isso que o import `pyperclip` importa o arquivo `pyperclip.py`). É assim que nosso programa sabe se está sendo executado como o programa principal ou importado por um programa diferente como um módulo. (Você importará `transpositionEncrypt.py` como um módulo no [Capítulo 9](#)).

Quando você importa um programa em Python e antes que o programa seja executado, a variável `__name__` é definida para a parte do nome do arquivo anterior. `.py`. Quando o programa `transpositionEncrypt.py` é importado, todas as instruções def são executadas (para definir a função `encryptMessage ()` que o programa importador quer usar), mas a função `main ()` não é chamada, portanto o código de criptografia para 'Common o sentido não é tão comum. com a chave 8 não é executado.

É por isso que o código que criptografa a string `myMessage` com a chave `myKey` está dentro de uma função (que por convenção é chamada `main ()`). Este código dentro de `main ()` não será executado quando `transpositionEncrypt.py` for importado por outros programas, mas esses outros programas ainda podem

chamar sua função `encryptMessage()`. É assim que o código da função pode ser reutilizado por outros programas.

NOTA

Uma forma útil de aprender como um programa funciona é seguindo sua execução passo a passo à medida que ele é executado. Você pode usar uma ferramenta de rastreamento de programas on-line para visualizar os rastreamentos dos programas de criptografia Hello Function e Transposition Cipher em <https://www.nostarch.com/crackingcodes/>. A ferramenta de rastreamento fornecerá uma representação visual do que os programas estão fazendo à medida que cada linha de código é executada.

Resumo

Ufa! Você aprendeu vários novos conceitos de programação neste capítulo. O programa de cifra de transposição é mais complicado (mas muito mais seguro) do que o programa de cifra de César no [Capítulo 6](#). Os novos conceitos, funções, tipos de dados e operadores que você aprendeu neste capítulo permitem manipular dados de maneiras mais sofisticadas. Apenas lembre-se de que muito do que se passa em compreender uma linha de código é avaliar passo a passo da mesma maneira que o Python.

Você pode organizar o código em grupos chamados funções, que você cria com instruções `def`. Valores de argumentos podem ser passados para funções como parâmetros da função. Parâmetros são variáveis locais. Variáveis fora de todas as funções são variáveis globais. As variáveis locais são diferentes das variáveis globais, mesmo se tiverem o mesmo nome que a variável global. Variáveis locais em uma função também são separadas das variáveis locais em outra função, mesmo que tenham o mesmo nome.

Valores de lista podem armazenar vários outros valores, incluindo outros valores de lista. Muitas das operações que você pode usar em seqüências de caracteres (como indexação, fatiamento e usando a função `len()`) podem ser usadas em listas. E operadores de atribuição aumentados fornecem um bom atalho para operadores de atribuição regulares. O método `join()` pode unir-se a uma lista que contém várias cadeias para retornar uma única cadeia de caracteres.

Talvez seja melhor revisar este capítulo se você ainda não estiver familiarizado com esses conceitos de programação. No [Capítulo 8](#), você aprenderá a descriptografar usando a cifra de transposição.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Com papel e lápis, criptografe as seguintes mensagens com a chave 9 usando a cifra de transposição. O número de caracteres foi fornecido para sua conveniência.
 - Debaixo de um enorme carvalho havia de suíno uma enorme companhia, (61 caracteres)
 - Isso grunhiu quando eles mastigaram o mastro: pois estava maduro e caiu rapidamente. (77 caracteres)
 - Então eles correram para longe, pois o vento aumentou: uma bolota foi embora e você não mais poderia espiar. (94 caracteres)
2. No programa a seguir, cada spam é uma variável global ou local?

```
spam = 42
def foo():
    spam global de
    spam = 99
    imprimir (spam)
```

3. Para qual valor cada uma das seguintes expressões é avaliada?

```
[0, 1, 2, 3, 4] [2]
[[1, 2], [3, 4]] [0]
[[1, 2], [3, 4]] [0] [1]
[ 'olá'] [0] [1]
[2, 4, 6, 8, 10] [1: 3]
lista ('Olá mundo!')
lista (intervalo (10)) [2]
```

4. Para qual valor cada uma das seguintes expressões é avaliada?

```
len ([2, 4])
len ([])

len ([" ", " "])
[4, 5, 6] + [1, 2, 3]
3 * [1, 2, 3] + [9]
```

42 em [41, 42, 42, 42]

5. Quais são os quatro operadores de atribuição aumentada?

8

DESCRIPTANDO COM A CIPHER DE TRANSPOSIÇÃO

“Enfraquecer a criptografia ou criar backdoors para dispositivos e dados criptografados para uso dos mocinhos criaria vulnerabilidades para serem exploradas pelos bandidos.”

—Tim Cook, CEO da Apple, 2015



Ao contrário da cifra de César, o processo de descriptografia da cifra de transposição é diferente do processo de criptografia. Neste capítulo, você criará um programa separado chamado *transpositionDecrypt.py* para manipular a descriptografia.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Decifrando com a cifra de transposição
- As funções `round()`, `math.ceil()` e `math.floor()`
- Os operadores `e /` ou `booleanos`
- Tabelas verdade

Como descriptografar com a cifra de transposição em papel

Finja que você enviou o texto cifrado “Cenoconomstmme oo snnio. ssc ”para um amigo (e eles já sabem que a chave secreta é 8). O primeiro passo para descriptografar o texto cifrado é calcular o número de caixas que precisam desenhar. Para determinar esse número, eles devem dividir o comprimento da mensagem de texto cifrado pela chave e arredondar para o número inteiro mais próximo se o resultado já não for um número inteiro. O comprimento do texto cifrado é de 30 caracteres (o mesmo que o texto original) e a chave é 8, então 30

dividido por 8 é 3.75.

C	e	n	o
o	n	o	m
m	s	t	m
m	e	■	o
o	■	s	n
n	i	o	.
■	s	■	
s	■	c	

Figura 8-1: Descriptografando a mensagem invertendo a grade

Arredondando-se de 3,75 a 4, seu amigo desenhará uma grade de caixas com quatro colunas (o número que acabaram de calcular) e oito linhas (a chave).

Seu amigo também precisa calcular o número de caixas para sombrear. Usando o número total de caixas (32), elas subtraem o comprimento do texto cifrado (que é 30): $32 - 30 = 2$. Elas sombreiam nas duas caixas *inferiores*, na coluna *mais à direita*.

Então eles começam a preencher as caixas, colocando um caractere do texto cifrado em cada caixa. Começando no canto superior esquerdo, eles são preenchidos para a direita, como você fez quando estava criptografando. O texto cifrado é “Cenoonommstmme oo snnio. ssc”, então “Ceno” vai na primeira linha, “onom” vai na segunda linha e assim por diante. Quando terminarem, as caixas se parecerão com a [Figura 8-1](#) (a ■ representa um espaço).

Seu amigo que recebeu o texto cifrado percebe que quando lêem o texto descendo as colunas, o texto original é restaurado: “O senso comum não é tão comum”.

Para recapitular, as etapas para descriptografar são as seguintes:

1. Calcule o número de colunas necessárias dividindo o comprimento da mensagem pela chave e arredondando para cima.
2. Desenhe caixas em colunas e linhas. Use o número de colunas que

você calculou na etapa 1. O número de linhas é o mesmo que a chave.

3. Calcule o número de caixas para sombrear, tomando o número total de caixas (o número de linhas multiplicado pelo número de colunas) e subtraindo o comprimento da mensagem de texto cifrado.
4. Sobre no número de caixas que você calculou na etapa 3 na parte inferior da coluna da extrema direita.
5. Preencha os caracteres do texto cifrado, começando na linha superior e indo da esquerda para a direita. Ignore qualquer uma das caixas sombreadas.
6. Obtenha o texto sem formatação lendo a coluna mais à esquerda de cima para baixo e continuando a fazer o mesmo em cada coluna.

Observe que, se você usou uma chave diferente, desenharia o número errado de linhas. Mesmo se você seguisse as outras etapas do processo de descriptografia corretamente, o texto simples seria um lixo aleatório (semelhante a se você usou a chave errada com a cifra de César).

Código Fonte para o Programa de Descriptografia de Transposição de Cifras

Abra uma nova janela do editor de **arquivos** clicando em **Arquivo ▶ Novo arquivo**. Digite o seguinte código no editor de arquivos e salve-o como *transpositionDecrypt.py*. Lembre-se de colocar o *pyperclip.py* no mesmo diretório. Pressione F5 para executar o programa.

transposição

Decrypt.py

```
1. # Descriptografia de cifra de transposição
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar matemática, pyperclip
5
6. def main():
7.     myMessage = 'Cenoonommstmme oo snnio. ss c '
8.     myKey = 8
9
10.    plaintext = decryptMessage (myKey, myMessage)
```

```
11
12. # Imprimir com um | (chamado "pipe" personagem) depois no caso
13. # há espaços no final da mensagem descriptografada:
14. print (texto sem formatação + '|')
15
16. pyperclip.copy (texto sem formatação)
17
18
19. def decryptMessage (chave, mensagem):
20. # A função de decodificação de transposição simulará as "colunas" e
21. # "linhas" da grade em que o texto original é escrito usando uma lista
22. # de cordas. Primeiro, precisamos calcular alguns valores.
23
24. # O número de "colunas" na nossa grelha de transposição:
25. numOfColumns = int (math.ceil (len (mensagem) / float (chave)))
26. # O número de "linhas" em nossa grade:
27. numRows = chave
28. # O número de "caixas sombreadas" na última "coluna" da grade:
29. numShadedBoxes = (numOfColumns * numRows) - len (mensagem)
30
31. # Cada string em texto simples representa uma coluna na grade:
32. plaintext = [""] * numOfColumns
33
34. # As variáveis coluna e linha apontam para onde na grade a próxima
35. # personagem na mensagem criptografada irá:
36. coluna = 0
37. linha = 0
38
39. para o símbolo na mensagem:
40. texto simples [coluna] += símbolo
41. column += 1 # Aponte para a próxima coluna.
42.
43. # Se não houver mais colunas OU estivermos em uma caixa sombreada,
volte
44. # para a primeira coluna e a seguinte linha:
45. if (coluna == numOfColumns) ou (coluna == numOfColumns - 1 e
linha >= numRows - numShadedBoxes):
```

```
46. coluna = 0
47. linha += 1
48
49. return " ".join(texto sem formatação)
50
51
52. # Se transpositionDecrypt.py for executado (em vez de importado como um
módulo),
53. # chama a função main():
54. if __name__ == '__main__':
55.     main()
```

Execução de Amostra do Programa de Descriptografia de Transposição de Cifras

Quando você executa o programa *transpositionDecrypt.py* , ele produz esta saída:

O senso comum não é tão comum.

Se você quiser descriptografar uma mensagem diferente ou usar uma chave diferente, altere o valor atribuído às variáveis myMessage e myKey nas linhas 7 e 8.

Importando Módulos e Configurando a Função main ()

A primeira parte do programa *transpositionDecrypt.py* é semelhante à primeira parte do *transpositionEncrypt.py* :

```
1. # Descriptografia de cifra de transposição
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar matemática, pyperclip
5
6. def main():
7.     myMessage = 'Cenoonommstmme oo snnio. ss c '
8.     myKey = 8
9
10.    plaintext = decryptMessage (myKey, myMessage)
11
12.    # Imprimir com um | (chamado "pipe" personagem) depois no caso
```

13. # há espaços no final da mensagem descriptografada:

14. print (texto sem formatação + '|')

15

16. pyperclip.copy (texto sem formatação)

O módulo pyperclip é importado junto com outro módulo chamado math on line 4. Se você separar os nomes dos módulos com vírgulas, poderá importar vários módulos com uma declaração de importação .

A função main () , que começamos a definir na linha 6, cria variáveis chamadas myMessage e myKey e, em seguida, chama a função decryptMessage () da descriptografia. O valor de retorno de decryptMessage () é o texto plano descriptografado do texto cifrado e da chave. Isso é armazenado em uma variável chamada plaintext , que é impressa na tela (com um caractere pipe no final, caso haja espaços no final da mensagem) e depois copiada para a área de transferência.

Descriptografar a mensagem com a chave

A função decryptMessage () segue as seis etapas de descriptografia descritas na [página 100](#) e retorna os resultados da descriptografia como uma string. Para facilitar a decodificação, usaremos funções do módulo de matemática , que importamos anteriormente no programa.

As funções round (), math.ceil () e math.floor ()

A função round () do Python irá arredondar um número de ponto flutuante (um número com um ponto decimal) para o inteiro mais próximo. As funções math.ceil () e math.floor () (no módulo de matemática do Python) irão arredondar um número para cima e para baixo, respectivamente.

Quando você divide números usando o operador / , a expressão retorna um número de ponto flutuante (um número com um ponto decimal). Isso acontece mesmo se o número se dividir uniformemente. Por exemplo, insira o seguinte no shell interativo:

```
>>> 21/7  
3,0  
>>> 22/5  
4,4
```

Se você quiser arredondar um número para o inteiro mais próximo, você pode usar a função round () . Para ver como a função funciona, digite o seguinte:

```
>>> volta (4.2)
4
>>> volta (4.9)
5
>>> rodada (5,0)
5
>>> redondo (22/5)
4
```

Se você quer apenas arredondar para cima, você precisa usar a função `math.ceil()`, que representa “teto”. Se você quer apenas arredondar para baixo, use `math.floor()`. Essas funções existem no módulo de matemática , que você precisa importar antes de chamá-las. Digite o seguinte no shell interativo:

```
>>> importar matemática
>>> math.floor (4.0)
4
>>> math.floor (4.2)
4
>>> math.floor (4.9)
4
>>> math.ceil (4.0)
4
>>> math.ceil (4.2)
5
>>> math.ceil (4.9)
5
```

A função `math.floor()` irá sempre remover o ponto decimal do float e convertê-lo em um inteiro para arredondar para baixo, e `math.ceil()` irá incrementar o lugar do float e convertê-lo em um inteiro para arredondar para cima .

A função `decryptMessage()`

A função `decryptMessage()` implementa cada uma das etapas de descriptografia como código Python. É preciso uma chave inteira e uma string de mensagem como argumentos. A função `math.ceil()` é usada para a descriptografia de transposição em `decryptMessage()` quando as colunas são calculadas para determinar o número de caixas que precisam ser feitas:

19. def `decryptMessage(chave, mensagem):`

```
20. # A função de decodificação de transposição simulará as "colunas" e
21. # "linhas" da grade em que o texto original é escrito usando uma lista
22. # de cordas. Primeiro, precisamos calcular alguns valores.
23
24. # O número de "colunas" na nossa grelha de transposição:
25. numOfColumns = int (math.ceil (len (mensagem) / float (chave)))
26. # O número de "linhas" em nossa grade:
27. numRows = chave
28. # O número de "caixas sombreadas" na última "coluna" da grade:
29. numShadedBoxes = (numOfColumns * numRows) - len (mensagem)
```

A linha 25 calcula o número de colunas dividindo `len (mensagem)` pelo inteiro na chave . Esse valor é passado para a função `math.ceil ()` e esse valor de retorno é armazenado em `numOfColumns` . Para tornar este programa compatível com o Python 2, chamamos `float ()` para que a chave se torne um valor de ponto flutuante. No Python 2, o resultado da divisão de dois inteiros é automaticamente arredondado para baixo. Chamar `float ()` evita esse comportamento sem afetar o comportamento do Python 3.

A linha 27 calcula o número de linhas, que é o número inteiro armazenado na chave . Este valor é armazenado na variável `numRows` .

A linha 29 calcula o número de caixas sombreadas na grade, que é o número de colunas vezes linhas, menos o comprimento da mensagem.

Se você está decifrando “Cenoonommstmme oo snnio. ssc ”com uma chave de 8, `numOfColumns` é definido como 4 , `numRows` é definido como 8 e `numShadedBoxes` é definido como 2 .

Assim como o programa de criptografia tinha uma variável chamada `ciphertext` que era uma lista de strings para representar a grade do texto cifrado, `decryptMessage ()` também tem uma variável list-of-strings chamada `plaintext` :

```
31. # Cada string em texto simples representa uma coluna na grade:
32. plaintext = [""] * numOfColumns
```

Essas seqüências de caracteres estão em branco no início, com uma seqüência de caracteres para cada coluna da grade. Usando a replicação de lista, você pode multiplicar uma lista de uma string em branco por `numOfColumns` para fazer uma lista de várias strings em branco igual ao número de colunas necessárias.

Tenha em mente que este texto simples é diferente do texto simples na função

main () . Como a função decryptMessage () e a função main () possuem seu próprio escopo local, as variáveis de texto simples das funções são diferentes e apenas têm o mesmo nome.

Lembre-se que a grade para o 'Cenoonommstmme oo snnio. ss c ' exemplo é semelhante à [Figura 8-1 na página 100](#) .

A variável de texto simples terá uma lista de strings e cada string na lista será uma única coluna dessa grade. Para essa descriptografia, você deseja que o texto simples fique com o seguinte valor:

['Senso comum não é tão comum.']}

Dessa forma, você pode unir todas as strings da lista para retornar o "O senso comum não é tão comum". valor da string.

Para fazer a lista, primeiro precisamos colocar cada símbolo na mensagem na string correta dentro da lista de texto simples, um de cada vez. Vamos criar duas variáveis chamadas coluna e linha para rastrear a coluna e linha onde o próximo caractere na mensagem deve ir; essas variáveis devem começar em 0 para começar na primeira coluna e na primeira linha. As linhas 36 e 37 fazem isso:

34. # As variáveis coluna e linha apontam para onde na grade a próxima

35. # personagem na mensagem criptografada irá:

36. coluna = 0

37. linha = 0

A linha 39 inicia um loop for que itera sobre os caracteres na cadeia de mensagens . Dentro deste loop, o código ajustará as variáveis de coluna e linha para concatenar o símbolo à string correta na lista de texto simples :

39. para o símbolo na mensagem:

40. texto simples [coluna] + = símbolo

41. column + = 1 # Aponte para a próxima coluna.

A linha 40 concatena o símbolo para a cadeia na coluna de índice na lista de texto sem formatação , porque cada seqüência de caracteres em texto sem formatação representa uma coluna. Em seguida, a linha 41 adiciona 1 à coluna (isto é, *incrementa a coluna*) para que, na próxima iteração do loop, o símbolo seja concatenado à próxima cadeia na lista de texto sem formatação .

Nós manipulamos a coluna e a linha de incremento, mas também precisaremos redefinir as variáveis para 0 em alguns casos. Para entender o código que faz

isso, você precisará entender os operadores booleanos.

Operadores booleanos

Operadores booleanos comparam valores booleanos (ou expressões que avaliam um valor booleano) e avaliam um valor booleano. Os operadores booleanos e / ou podem ajudá-lo a criar condições mais complicadas para declarações if e while . O operador e conecta duas expressões e avalia como True se ambas as expressões forem avaliadas como True . O operador ou conecta duas expressões e avalia como True se uma ou ambas as expressões forem avaliadas como True ; caso contrário, essas expressões serão avaliadas como Falso . Digite o seguinte no shell interativo para ver como o operador e funciona:

```
>>> 10> 5 e 2 <4
```

Verdade

```
>>> 10> 5 e 4! = 4
```

Falso

A primeira expressão é avaliada como True, pois as expressões de cada lado do operador e são avaliadas como True . Em outras palavras, a expressão $10 > 5$ e $2 < 4$ é avaliada como Verdadeiro e Verdadeiro , o que, por sua vez, é avaliado como Verdadeiro .

No entanto, na segunda expressão, embora $10 > 5$ seja avaliado como Verdadeiro , a expressão $4! = 4$ é avaliada como Falso . Isso significa que a expressão é avaliada como True e False . Como as duas expressões devem ser True para o operador e para avaliar como True , a expressão inteira é avaliada como False .

Se você alguma vez esquecer como funciona um operador booleano, poderá ver sua *tabela de verdade* , que mostra quais diferentes combinações de valores booleanos avaliam com base no operador usado. [A Tabela 8-1](#) é uma tabela de verdade para o operador e .

Tabela 8-1: A tabela da verdade e do operador

A e B	Avalia para
Verdadeiro e Verdadeiro	Verdade
Verdadeiro e falso	Falso

Falsa e Verdadeira Falso

Falso e falso Falso

Para ver como o operador or funciona, digite o seguinte no shell interativo:

```
>>> 10> 5 ou 4! = 4
```

Verdade

```
>>> 10 <5 ou 4! = 4
```

Falso

Quando você está usando o operador or , somente um lado da expressão deve ser True para que o operador or avalie a expressão inteira como True , e é por isso que $10 > 5$ ou $4! = 4$ são avaliados como True . No entanto, como a expressão $10 < 5$ e a expressão $4! = 4$ são False , a segunda expressão é avaliada como False ou False , que, por sua vez, é avaliada como False .

A tabela de verdade do operador ou é mostrada na [Tabela 8-2](#) .

Tabela 8-2: A Tabela da Verdade do Operador

A ou B	Avalia para
--------	-------------

Verdadeiro ou Verdadeiro	Verdade
-----------------------------	---------

Verdadeiro ou falso	Verdade
---------------------	---------

Falso ou Verdadeiro	Verdade
---------------------	---------

Falsa ou falsa	Falso
----------------	-------

O terceiro operador booleano não é. O operador não avalia o valor Booleano oposto do valor em que opera. Então não é verdade é falso e não falso é verdadeiro . Digite o seguinte no shell interativo:

```
>>> não 10> 5
```

Falso

```
>>> não 10 <5
```

Verdade

```
>>> não falso
```

Verdade

```
>>> não não é falso
```

Falso

```
>>> não não não não é falso
```

Verdade

Como você pode ver nas duas últimas expressões, você pode até usar vários operadores não . A tabela de verdade do não operador é mostrada na [Tabela 8-3](#) .

Tabela 8-3: A tabela da verdade não do operador

não A	Avalia para
--------------	--------------------

Não é verdade	Falso
---------------	-------

não é falso	Verdade
-------------	---------

O and e ou operadores são atalhos

Semelhante a como os loops permitem fazer a mesma tarefa que os loops while , mas com menos código, os operadores e / ou também permitem que você encurte seu código. Digite as duas partes de código a seguir, que têm o mesmo resultado, no shell interativo:

```
>>> se 10> 5:
```

```
... se 2 <4:
```

```
... print ('Olá!')
```

```
...
```

Olá!

```
>>> se 10> 5 e 2 <4:
```

```
... print ('Olá!')
```

```
...
```

Olá!

O operador e pode tomar o lugar de duas instruções if que verificam cada parte da expressão separadamente (onde a segunda instrução if está dentro do bloco da primeira instrução if).

Você também pode substituir uma instrução if e elif pelo operador ou . Para tentar, insira o seguinte no shell interativo:

```
>>> se 4! = 4:  
... print ('Olá!')  
... elif 10> 5:  
... print ('Olá!')  
...  
Olá!  
>>> se 4! = 4 ou 10> 5:  
... print ('Olá!')  
...  
Olá!
```

As instruções if e elif verificarão cada uma parte diferente da expressão, enquanto o operador or pode verificar ambas as instruções em uma linha.

Ordem de Operações para Operadores Booleanos

Você sabe que os operadores matemáticos têm uma ordem de operações, assim como os operadores e , ou , e não operadores. Primeiro, não é avaliado, depois e , e depois ou . Digite o seguinte no shell interativo:

```
>>> não falso e falso # não falso avalia primeiro  
Falso  
>>> não (falso e falso) # (falso e falso) avalia primeiro  
Verdade
```

Na primeira linha de código, não é avaliado Falso primeiro, portanto, a expressão se torna True e False , que é avaliada como False . Na segunda linha, os parênteses são avaliados primeiro, mesmo antes do operador not , então False e False são avaliados como False , e a expressão não é (False) , que é True .

Ajustando as variáveis de coluna e linha

Agora que você sabe como os operadores booleanos funcionam, é possível aprender como as variáveis de coluna e linha são reconfiguradas em *transpositionDecrypt.py* .

Existem dois casos nos quais você desejará redefinir a coluna como 0 para que, na próxima iteração do loop, o símbolo seja adicionado à primeira string da lista em texto sem formatação . No primeiro caso, você deseja fazer isso se a coluna for incrementada após o último índice em texto simples . Nessa situação, o valor na coluna será igual a numOfColumns . (Lembre-se de que o último índice em texto simples será numOfColumns menos um. Então, quando column é igual a

`numOfColumns` , ele já passou do último índice.)

O segundo caso é se `column` estiver no último índice e a variável de linha estiver apontando para uma linha que tenha uma caixa sombreada na última coluna. Como um exemplo visual disso, a grade de decriptografia com os índices de coluna ao longo do topo e os índices de linha ao lado é mostrada na [Figura 8-2](#) .

	0	1	2	3
0	C 0	e 1	n 2	o 3
1	o 4	n 5	o 6	m 7
2	m 8	s 9	t 10	m 11
3	m 12	e 13	■ 14	o 15
4	o 16	■ 17	s 18	n 19
5	n 20	i 21	o 22	.
6	■ 24	s 25	■ 26	
7	s 27	■ 28	c 29	

Figura 8-2: Grade de descriptografia com índices de colunas e linhas

Você pode ver que as caixas sombreadas estão na última coluna (cujo índice será `numOfColumns - 1`) nas linhas 6 e 7. Para calcular quais índices de linha potencialmente têm caixas sombreadas, use a linha de expressão `>= numOfRows - numOfShadedBoxes` . Em nosso exemplo com oito linhas (com índices de 0 a 7), as linhas 6 e 7 são sombreadas. O número de caixas não sombreadas é o número total de linhas (no nosso exemplo, 8) menos o número de caixas sombreadas (no nosso exemplo, 2). Se a linha atual for igual ou maior que esse número ($8 - 2 = 6$), podemos saber que temos uma caixa sombreada. Se essa expressão for True e `column` também for igual a `numOfColumns - 1` , o Python encontrou uma caixa sombreada; Neste ponto, você deseja redefinir a coluna como 0 para a próxima iteração:

43. # Se não houver mais colunas OU estivermos em uma caixa sombreada, volte

44. # para a primeira coluna e a seguinte linha:

45. if (`coluna == numOfColumns`) ou (`coluna == numOfColumns - 1` e `linha >= numOfRows - numOfShadedBoxes`):

```
46. coluna = 0  
47. linha += 1
```

Esses dois casos são por que a condição na linha 45 é (coluna == numOfColumns) ou (coluna == numOfColumns - 1 e linha >= numOfRows - numOfShadedBoxes) . Embora pareça uma expressão grande e complicada, lembre-se de que você pode dividi-la em partes menores. A expressão (column == numOfColumns) verifica se a variável da coluna está além do intervalo do índice e a segunda parte da expressão verifica se estamos em um índice de coluna e linha que é uma caixa sombreada. Se uma dessas duas expressões for verdadeira, o bloco de código que é executado redefinirá a coluna para a primeira coluna definindo-a como 0 . Você também incrementará a variável de linha .

No momento em que o loop for na linha 39 terminou o loop sobre cada caractere na mensagem , as strings da lista de texto simples foram modificadas, de modo que agora estão na ordem decriptografada (se a chave correta tiver sido usada). As strings na lista de texto simples são unidas (com uma string em branco entre cada string) pelo método de string join () na linha 49:

```
49. return " ".join(texto sem formatação)
```

A linha 49 também retorna a string que a função decryptMessage () retorna.

Para descriptografia, o texto simples será ['Comum s', 'ense is', 'não é c', 'ommon.'], Portanto " ".join (texto simples) será avaliado como 'Senso comum não é tão comum'.

Chamando a função main ()

A primeira linha que nosso programa executa depois de importar os módulos e executar as instruções def é a declaração if na linha 54.

```
52. # Se transpositionDecrypt.py for executado (em vez de importado como um módulo),  
53. # chama a função main ():  
54. if __name__ == '__main__':  
55.     main()
```

Assim como no programa de criptografia de transposição, o Python verifica se esse programa foi executado (em vez de importado por um programa diferente) verificando se a variável __name__ está configurada com o valor da string

'__main__'. Nesse caso, o código executa a função main () .

Resumo

É isso para o programa de descriptografia. A maior parte do programa está na função decryptMessage () . Os programas que escrevemos podem criptografar e descriptografar a mensagem “O bom senso não é tão comum” com a chave 8; no entanto, você deve tentar várias outras mensagens e chaves para verificar se uma mensagem criptografada e descriptografada resulta na mesma mensagem original. Se você não obtiver os resultados esperados, saberá que o código de criptografia ou o código de descriptografia não funciona. No [Capítulo 9](#) , vamos automatizar esse processo escrevendo um programa para testar nossos programas.

Se você quiser ver um passo a passo da execução do programa de decodificação de códigos de transposição, visite <https://www.nostarch.com/crackingcodes/> .

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. Usando papel e lápis, decriptografe as seguintes mensagens com a tecla 9. ■ marca um único espaço. O número total de caracteres foi contado para você.
 - H ■ cb ■ ■ irhdeuousBdi ■ ■ ■ prrtyevdgp ■ nir ■ ■ eerit ■ eatoreechadihf ■ pak pt ■ ge ■ b ■ te ■ dih ■ aoa.da ■ tts ■ tn (89 caracteres)
 - A ■ b ■ ■ drottthawa ■ nwar ■ eci ■ t ■ nlel ■ ktShw ■ leec, hheat ■ .na ■ ■ e ■ soog mah ■ a ■ ■ ateniAcgakh ■ dmnor ■ ■ (86 caracteres)
 - Bmmsrl ■ dpnaua! Toeboo'kttn ■ uknrwos. ■ yaregonr ■ w ■ nd, tu ■ ■ oiady ■ h gtRwt ■ ■ ■ A ■ hhanhasthtev ■ ■ e ■ t ■ e ■ ■ e (93 caracteres)
2. Quando você insere o código a seguir no shell interativo, o que cada linha imprime?

```
>>> math.ceil (3.0)
>>> math.floor (3.1)
>>> rodada (3.1)
```

```
>>> rodada (3.5)
>>> Falso e Falso
>>> Falso ou Falso
>>> não não é verdade
```

3. Desenhe as tabelas de verdade completas para os operadores e, ou, e não operadores.
4. Qual das seguintes opções está correta?

```
se __name__ == '__main__':
if __main__ == '__name__':
if __name__ == '__main__':
if __main__ == '__name__':
```

9

PROGRAMANDO UM PROGRAMA PARA TESTAR SEU PROGRAMA

"É uma falta de higiene cívica para instalar tecnologias que possam algum dia facilitar um estado policial".

Bruce Schneier, Segredos e Mentiras



Os programas de transposição parecem funcionar muito bem em criptografar e descriptografar diferentes mensagens com várias chaves, mas como você sabe que elas *sempre* funcionam? Você não pode ter certeza absoluta de que os programas sempre funcionem, a menos que você teste as funções encryptMessage () e decryptMessage () com todos os tipos de valores de parâmetro de mensagem e chave . Mas isso levaria muito tempo porque você teria que digitar uma mensagem no programa de criptografia, definir a chave, executar o programa de criptografia, colar o texto cifrado no programa de descriptografia, definir a chave e depois executar o programa de descriptografia. Você também precisará repetir esse processo com várias chaves e mensagens diferentes, resultando em muito trabalho chato!

Em vez disso, vamos escrever outro programa que gere uma mensagem aleatória e uma chave aleatória para testar os programas de criptografia. Este novo programa irá criptografar a mensagem com `encryptMessage()` de `transpositionEncrypt.py` e, em seguida, passar o texto cifrado para `decryptMessage()` de `transpositionDecrypt.py`. Se o texto original retornado por `decryptMessage()` for o mesmo que a mensagem original, o programa saberá que os programas de criptografia e descriptografia funcionam. O processo de testar um programa automaticamente usando outro programa é chamado de *teste automatizado*.

Várias combinações diferentes de mensagens e chaves precisam ser tentadas, mas o computador demora apenas um minuto para testar milhares de combinações. Se todos esses testes forem aprovados, você pode ter mais certeza de que seu código funciona.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A função `random.randint()`
- A função `random.seed()`
- Listar referências
- As funções `copy.deepcopy()`
- A função `random.shuffle()`
- Aleatoriamente misturando uma corda
- A função `sys.exit()`

Código-fonte para o programa de teste de cifra de transposição

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como `transpositionTest.py`. Em seguida, pressione F5 para executar o programa.

transposição

Test.py

1. Teste de cifra de transposição
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importação aleatória, sys, transpositionEncrypt, transpositionDecrypt
- 5

```
6. def main ():  
7.     random.seed (42) # Defina a "semente" aleatória para um valor estático.  
8  
9.     para i na faixa (20): # Execute 20 testes.  
10.    # Gere mensagens aleatórias para testar.  
11  
12.    # A mensagem terá um tamanho aleatório:  
13.    message = 'ABCDEFGHIJKLMNPQRSTUVWXYZ' * random.randint (4,  
14        40)  
14  
15.    # Converta a string de mensagem em uma lista para embaralhá-la:  
16.    message = list (message)  
17.    random.shuffle (mensagem)  
18.    message = " .join (message) # Converte a lista de volta para uma string.  
19  
20.    print ("Teste #% s: "% s ... "% (i + 1, mensagem [: 50]))  
21  
22.    # Verifique todas as chaves possíveis para cada mensagem:  
23.    para chave no intervalo (1, int (len (mensagem) / 2)):  
24.        encriptado = transpositionEncrypt.encryptMessage (chave, mensagem)  
25.        descriptografado = transpositionDecrypt.decryptMessage (chave,  
26            criptografada)  
26  
27.    # Se a decodificação não corresponder à mensagem original, exiba  
28.    # uma mensagem de erro e saia:  
29.    se mensagem! = Descriptografada:  
30.        print ('Incompatibilidade com a chave% s e mensagem% s.'% (Chave,  
31            mensagem))  
31.        print ('descriptografado como:' + descriptografado)  
32.        sys.exit ()  
33  
34.    print ('Teste de cifra de transposição passado')  
35  
36  
37.    # Se transpositionTest.py for executado (em vez de importado como um  
módulo), ligue  
38.    # a função main ():
```

```
39. if __name__ == '__main__':
40. main()
```

Execução de Amostra do Programa do Transponder Cipher Tester

Quando você executa o programa *transpositionTest.py*, a saída deve ficar assim:

Teste nº 1:

```
"JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIU
..."
```

Teste # 2:

```
"SWRCLUCRDOMLWZKOMAGVOTXUVVEPIOJMSBEQRQOFRGCCKEN
..."
```

Teste # 3:

```
"BIZBPZUIWDUFXAPJTHCMDWEGHYOWKWWWSJYKDQVSFWCJNCO
..."
```

Teste # 4:

```
"JEWBCEXVZAILLCHDZJCUTXASSZRKRPMYGTGHXPQPBEBCOD
..."
```

--recorte--

Teste # 17:

```
"KPKHHLPUWPSSIOULGKVEFHZOKBFHXUKVSEOWOENOZSNIDELAV
..."
```

Teste # 18:

```
"OYLFXZXENDFGSXTEAHGHPBNORCFEPBMITILSSJRGDVMNSOMUR
..."
```

Teste 19:

```
"SOCLYBRVDPLNVJKAFDGHCQMXIOPEJSXEAAXNWCCYAGZGLZGF
..."
```

Teste # 20:

```
"JXJGRBCKZXPUIEXOJUNZEYYSEAEGVOJWIRTSSGPUWPNZUBQNDA
..."
```

Teste de cifra de transposição passado.

O programa testador funciona importando os programas *transpositionEncrypt.py* e *transpositionDecrypt.py* como módulos. Em seguida, o programa testador chama *encryptMessage ()* e *decryptMessage ()* dos programas de criptografia e descriptografia. O programa testador cria uma mensagem aleatória e escolhe uma chave aleatória. Não importa que a mensagem seja apenas letras aleatórias,

porque o programa só precisa verificar essa criptografia e depois descriptografar os resultados da mensagem na mensagem original.

Usando um loop, o programa repete este teste 20 vezes. Se em algum momento a string retornada de `transpositionDecrypt()` não for igual à mensagem original, o programa imprime um erro e sai.

Vamos explorar o código fonte com mais detalhes.

Importando os Módulos

O programa começa importando módulos, incluindo dois que você já viu que vêm com o Python, `random` e `sys` :

1. Teste de cifra de transposição
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importação aleatória, `sys`, `transpositionEncrypt`, `transpositionDecrypt`

Também precisamos importar os programas de codificação de transposição (isto é, `transpositionEncrypt.py` e `transpositionDecrypt.py`) apenas digitando seus nomes sem a extensão `.py` .

Criando Números Pseudo-Aleatórios

Para criar números aleatórios para gerar as mensagens e chaves, usaremos a função `seed()` do módulo aleatório . Antes de nos aprofundarmos no que a semente faz, vamos ver como os números aleatórios funcionam no Python testando a função `random.randint()` . A função `random.randint()` que usaremos posteriormente no programa usa dois argumentos inteiros e retorna um inteiro aleatório entre esses dois inteiros (incluindo os inteiros). Digite o seguinte no shell interativo:

```
>>> importar aleatoriamente
>>> random.randint(1, 20)
20
>>> random.randint(1, 20)
18
>>> random.randint(100, 200)
107
```

Claro, os números que você recebe provavelmente serão diferentes daqueles mostrados aqui porque são números aleatórios.

Mas os números gerados pela função random.randint () do Python não são verdadeiramente aleatórios. Eles são produzidos a partir de um algoritmo gerador de números pseudo-aleatórios, que recebe um número inicial e produz outros números baseados em uma fórmula.

O número inicial com o qual o gerador de números pseudo-aleatórios inicia é chamado de *semente*. Se você conhece a semente, o resto dos números produzidos pelo gerador são previsíveis, porque quando você define a semente para um número específico, os mesmos números serão gerados na mesma ordem. Esses números aleatórios, mas previsíveis, são chamados de *números pseudo-aleatórios*. Os programas em Python para os quais você não define uma semente usam a hora atual do computador para definir uma semente. Você pode redefinir a semente aleatória do Python chamando a função random.seed () .

Para ver a prova de que os números pseudo-aleatórios não são completamente aleatórios, digite o seguinte no shell interativo:

```
>>> importar aleatoriamente
❶ >>> random.seed (42)
❷ >>> números = []
>>> para i na faixa (20):
... numbers.append (random.randint (1, 10))
...
❸ [2, 1, 5, 4, 4, 3, 2, 9, 2, 10, 7, 1, 1, 2, 4, 4, 9, 10, 1, 9]
>>> random.seed (42)
>>> números = []
>>> para i na faixa (20):
... numbers.append (random.randint (1, 10))
...
❹ [2, 1, 5, 4, 4, 3, 2, 9, 2, 10, 7, 1, 1, 2, 4, 4, 9, 10, 1, 9]
```

Neste código, geramos 20 números duas vezes usando a mesma semente. Primeiro, importamos aleatoriamente e definimos a semente para 42 ❶. Em seguida, configuramos uma lista chamada números ❷, onde armazenaremos nossos números gerados. Usamos um loop para gerar 20 números e acrescentar cada um à lista de números , que nós imprimimos para que possamos ver cada número gerado ❸ .

Quando a semente do gerador de números pseudo-aleatórios do Python é configurada para 42 , o primeiro número "aleatório" entre 1 e 10 sempre será 2 .

O segundo número será sempre 1 , o terceiro número será sempre 5 e assim por diante. Quando você redefine a semente para 42 e gera números com a semente novamente, o mesmo conjunto de números pseudo-aleatórios é retornado de random.randint () , como você pode ver comparando a lista de números em ❸ e ❹ .

Números aleatórios se tornarão importantes para cifras em capítulos posteriores, porque são usados não apenas para testar cifras, mas também para criptografar e descriptografar em cifras mais complexas. Os números aleatórios são tão importantes que uma falha de segurança comum no software de criptografia é usar números aleatórios previsíveis. Se os números aleatórios em seus programas podem ser previstos, um criptoanalista pode usar essa informação para quebrar sua cifra.

A seleção de chaves de criptografia de uma maneira verdadeiramente aleatória é necessária para a segurança de uma cifra, mas para outros usos, como este teste de código, os números pseudo-aleatórios são bons. Usaremos números pseudo-aleatórios para gerar strings de teste para nosso programa testador. Você pode gerar números realmente aleatórios com o Python usando a função random.SystemRandom () .randint () , sobre a qual você pode aprender mais em <https://www.nostarch.com/crackingcodes/> .

Criando uma seqüência aleatória

Agora que você aprendeu como usar random.randint () e random.seed () para criar números aleatórios, vamos retornar ao código-fonte. Para automatizar completamente nossos programas de criptografia e descriptografia, precisaremos gerar automaticamente mensagens aleatórias de strings.

Para fazer isso, usaremos uma sequência de caracteres para usar nas mensagens, duplicaremos um número aleatório de vezes e armazenaremos isso como uma string. Então, pegaremos a string dos caracteres duplicados e embaralhá-los para torná-los mais aleatórios. Nós vamos gerar uma nova string aleatória para cada teste, para que possamos tentar várias combinações de letras diferentes.

Primeiro, vamos configurar a função main () , que contém o código que testa os programas de criptografia. Ele começa definindo uma semente para a string pseudo-aleatória:

```
6. def main():
7.     random.seed(42) # Defina a "semente" aleatória para um valor estático.
```

Definir a semente aleatória chamando random.seed () é útil para o programa testador porque você quer números previsíveis para que as mesmas mensagens e chaves pseudo-aleatórias sejam escolhidas cada vez que o programa é executado. Como resultado, se você notar que uma mensagem não foi criptografada e descriptografada corretamente, você poderá reproduzir esse caso de teste com falha.

Em seguida, vamos duplicar uma string usando um loop for .

Duplicando uma String um Número Aleatório de Vezes

Usaremos um loop for para executar 20 testes e gerar nossa mensagem aleatória:

9. para i na faixa (20): # Execute 20 testes.
10. # Gere mensagens aleatórias para testar.
- 11
12. # A mensagem terá um tamanho aleatório:
13. message = 'ABCDEFGHIJKLMNPQRSTUVWXYZ' * random.randint (4, 40)

Cada vez que o loop for itera, o programa criará e testará uma nova mensagem. Queremos que este programa execute vários testes, porque quanto mais testes tentarmos, mais certos estaremos de que os programas funcionam.

A linha 13 é a primeira linha do código de teste e cria uma mensagem de tamanho aleatório. Ele pega uma string de letras maiúsculas e usa randint () e replicação de string para duplicar a string um número aleatório de vezes entre 4 e 40 . Em seguida, ele armazena a nova string na variável de mensagem .

Se deixarmos a sequência de mensagens como está agora, será sempre apenas a sequência do alfabeto repetida um número aleatório de vezes. Como queremos testar diferentes combinações de caracteres, precisamos levar as coisas um passo adiante e misturar os caracteres na mensagem . Para fazer isso, vamos primeiro aprender um pouco mais sobre listas.

Listar Variáveis Usar Referências

Variáveis armazenam listas de maneira diferente do que armazenam outros valores. Uma variável conterá uma referência à lista, em vez da própria lista. Uma *referência* é um valor que aponta para algum bit de dados e uma *referência de lista* é um valor que aponta para uma lista. Isso resulta em um comportamento ligeiramente diferente para o seu código.

Você já sabe que as variáveis armazenam strings e valores inteiros. Digite o seguinte no shell interativo:

```
>>> spam = 42
>>> queijo = spam
>>> spam = 100
>>> spam
100
>>> queijo
42
```

Atribuímos 42 à variável de spam e, em seguida, copiamos o valor em spam e atribuímo-lo ao queijo variável. Quando mais tarde alteramos o valor de spam para 100 , o novo número não afeta o valor do queijo, pois o spam e o queijo são variáveis diferentes que armazenam valores diferentes.

Mas as listas não funcionam assim. Quando atribuímos uma lista a uma variável, estamos realmente atribuindo uma referência de lista à variável. O código a seguir faz com que essa distinção seja mais fácil de entender. Digite este código no shell interativo:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> queijo = spam
❸ >>> cheese [1] = 'Olá!'
>>> spam
[0, 'Olá!', 2, 3, 4, 5]
>>> queijo
[0, 'Olá!', 2, 3, 4, 5]
```

Esse código pode parecer estranho para você. O código mudou apenas a lista de queijos , mas as listas de queijo e spam foram alteradas.

Quando criamos a lista we, atribuímos uma referência a ela na variável spam . Mas a próxima linha ❷ copia apenas a referência da lista em spam para o queijo , não o valor da lista. Isso significa que os valores armazenados em spam e queijo agora se referem à mesma lista. Há apenas uma lista subjacente porque a lista real nunca foi realmente copiada. Então, quando modificamos o primeiro elemento do queijo we, estamos modificando a mesma lista à qual o spam se refere.

Lembre-se de que as variáveis são como caixas que contêm valores. Mas as

variáveis de lista não contêm listas - elas contêm referências a listas. (Essas referências terão números de ID que o Python usa internamente, mas você pode ignorá-los.) Usando caixas como uma metáfora para variáveis, a [Figura 9-1](#) mostra o que acontece quando uma lista é atribuída à variável spam .

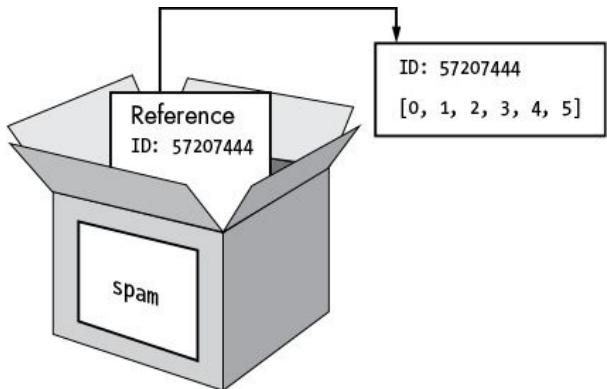


Figura 9-1: spam = [0, 1, 2, 3, 4, 5] armazena uma referência a uma lista, não a lista real.

Então, na [Figura 9-2](#) , a referência em spam é copiada para o queijo . Apenas uma nova referência foi criada e armazenada no queijo , não em uma nova lista. Observe que ambas as referências se referem à mesma lista.

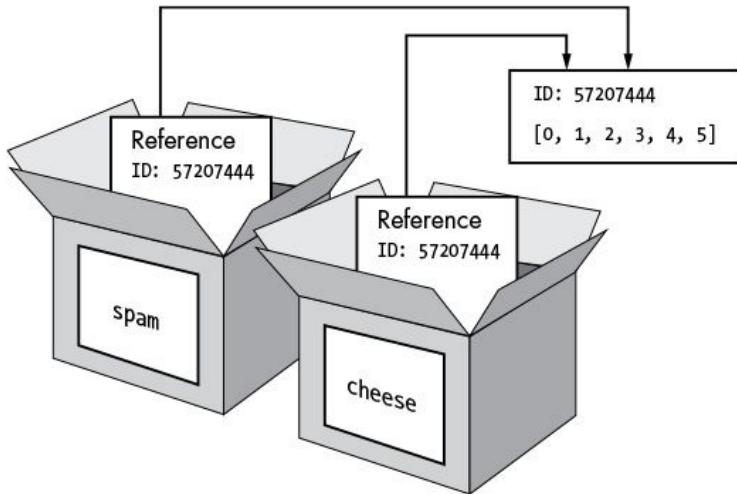


Figura 9-2: spam = cheese copia a referência, não a lista.

Quando alteramos a lista à qual o queijo se refere, a lista à qual o spam se refere também muda, porque tanto o queijo quanto o spam se referem à mesma lista. Você pode ver isso na [Figura 9-3](#) .

Embora as variáveis do Python contenham, tecnicamente, referências a valores de lista, as pessoas geralmente dizem que a variável "contém a lista".

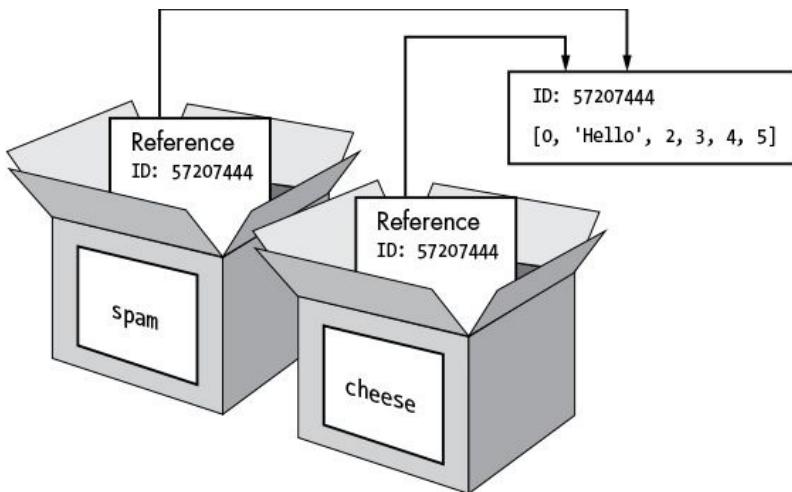


Figura 9-3: queijo [1] = 'Olá!' modifica a lista à qual ambas as variáveis se referem.

Referências Passando

As referências são particularmente importantes para entender como os argumentos são passados para as funções. Quando uma função é chamada, os valores dos argumentos são copiados para as variáveis de parâmetro. Para listas, isso significa que uma cópia da referência é usada para o parâmetro. Para ver as consequências dessa ação, abra uma nova janela do editor de arquivos, insira o código a seguir e salve-a como *passingReference.py*. Pressione F5 para executar o código.

passagem
Reference.py

```
def ovos (someParameter):
    someParameter.append ('Hello')
spam = [1, 2, 3]
ovos (spam)
imprimir (spam)
```

Quando você executa o código, observe que quando eggs () é chamado, um valor de retorno não é usado para atribuir um novo valor ao spam . Em vez disso, a lista é modificada diretamente. Quando executado, este programa produz a seguinte saída:

[1, 2, 3, 'Olá']

Mesmo que spam e someParameter contenham referências separadas, ambos se

referem à mesma lista. É por isso que a chamada do método append ('Hello') dentro da função afeta a lista, mesmo após a chamada da função ter retornado.

Tenha esse comportamento em mente: esquecer que o Python manipula variáveis de lista dessa maneira pode levar a erros confusos.

Usando copy.deepcopy () para duplicar uma lista

Se você deseja copiar um valor de lista, pode importar o módulo de cópia para chamar a função `copy.deepcopy ()`, que retorna uma cópia separada da lista pela qual foi passada:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cópia de importação
>>> cheese = copy.deepcopy (spam)
>>> cheese [1] = 'Olá!'
>>> spam
[0, 1, 2, 3, 4, 5]
>>> queijo
[0, 'Olá!', 2, 3, 4, 5]
```

Como a função `copy.deepcopy ()` foi usada para copiar a lista de `spam` para `queijo`, quando um item no `queijo` é alterado, o `spam` não é afetado.

Usaremos essa função no [Capítulo 17](#) quando hackarmos a simples cifra de substituição.

A função random.shuffle ()

Com uma base em como as referências funcionam, agora você pode entender como funciona a função `random.shuffle ()` que usaremos a seguir. A função `random.shuffle ()` faz parte do módulo aleatório e aceita um argumento de lista cujos itens ele reorganiza aleatoriamente. Digite o seguinte no shell interativo para ver como o `random.shuffle ()` funciona:

```
>>> importar aleatoriamente
>>> spam = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> spam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle (spam)
>>> spam
[3, 0, 5, 9, 6, 8, 2, 4, 1, 7]
>>> random.shuffle (spam)
```

```
>>> spam  
[1, 2, 5, 9, 4, 7, 0, 3, 6, 8]
```

Um detalhe importante a ser observado é que `shuffle()` *não retorna um valor de lista*. Em vez disso, ele altera o valor da lista que é passado para ele (porque `shuffle()` modifica a lista diretamente do valor de referência da lista que é passado). A função `shuffle()` modifica a lista *no lugar*, e é por isso que nós executamos `random.shuffle(spam)` ao invés de `spam = random.shuffle(spam)`.

Aleatoriamente Scrambling uma String

Vamos voltar para `transpositionTest.py`. Para embaralhar os caracteres em um valor de string, primeiro precisamos converter a string em uma lista usando `list()`:

```
15. # Converta a string de mensagem em uma lista para embaralhá-la:  
16. message = list(message)  
17. random.shuffle(mensagem)  
18. message = " ".join(message) # Converte a lista de volta para uma string.
```

O valor de retorno de `list()` é um valor de lista com strings de um caractere de cada caractere na string passada para ele; Então, na linha 16, estamos reatribuindo `mensagem` para ser uma lista de seus personagens. Em seguida, `shuffle()` randomiza a ordem dos itens na `mensagem`. Em seguida, o programa converte a lista de strings de volta para um valor de string usando o método de string `join()`. Esse embaralhamento da sequência de mensagens nos permite testar muitas mensagens diferentes.

Testando cada mensagem

Agora que a mensagem aleatória foi feita, o programa testa as funções de criptografia e descriptografia com ela. O programa imprimirá alguns comentários para que possamos ver o que está fazendo enquanto está testando:

```
20. print("Teste #%s: \"%s ...\"%s (i + 1, mensagem [: 50]))
```

A linha 20 tem uma chamada `print()` que exibe em qual número de teste o programa está (precisamos adicionar 1 a `i` porque eu começo em 0 e os números de teste devem começar em 1). Como a string na `mensagem` pode ser longa, usamos o fatiamento de string para mostrar apenas os primeiros 50 caracteres da `mensagem`.

A linha 20 também usa interpolação de strings. O valor que `i + 1` avalia para

substituir o primeiro % s na string e o valor que a mensagem [: 50] avalia para substituir o segundo % s . Ao usar a interpolação de string, certifique-se de que o número de % s na string corresponda ao número de valores que estão entre os parênteses depois dela.

Em seguida, vamos testar todas as chaves possíveis. Embora a chave para a cifra de César possa ser um número inteiro de 0 a 65 (o comprimento do conjunto de símbolos), a chave para a cifra de transposição pode ser entre 1 e metade do comprimento da mensagem. O loop for na linha 23 executa o código de teste com as chaves 1 até (mas não incluindo) o tamanho da mensagem.

22. # Verifique todas as chaves possíveis para cada mensagem:
23. para chave no intervalo (1, int (len (mensagem) / 2)):
24. encriptado = transpositionEncrypt.encryptMessage (chave, mensagem)
25. descriptografado = transpositionDecrypt.decryptMessage (chave, criptografada)

A linha 24 criptografa a string na mensagem usando a função encryptMessage () . Porque esta função está dentro do arquivo *transpositionEncrypt.py* , precisamos para adicionar transpositionEncrypt. (com o período no final) para a frente do nome da função.

A string criptografada que é retornada de encryptMessage () é então passada para decryptMessage () . Precisamos usar a mesma chave para as duas chamadas de função. O valor de retorno de decryptMessage () é armazenado em uma variável chamada decifrada . Se as funções funcionassem, a string na mensagem deveria ser a mesma que a string descriptografada . Vamos ver como o programa verifica isso a seguir.

Verificando se a cifra funcionou e finalizando o programa

Depois de criptografar e descriptografar a mensagem, precisamos verificar se ambos os processos funcionaram corretamente. Para fazer isso, basta verificar se a mensagem original é a mesma que a mensagem descriptografada.

27. # Se a decodificação não corresponder à mensagem original, exiba
28. # uma mensagem de erro e saia:
29. se mensagem! = Descriptografada:
30. print ('Incompatibilidade com a chave% s e mensagem% s.'% (Chave, mensagem))
31. print ('descriptografado como:' + descriptografado)

```
32. sys.exit ()  
33  
34. print ('Teste de cifra de transposição passado')
```

A linha 29 testa se a mensagem e a descriptografia são iguais. Se não estiverem, o Python exibirá uma mensagem de erro na tela. As linhas 30 e 31 imprimem a chave , a mensagem e os valores descriptografados como comentários para nos ajudar a descobrir o que deu errado. Então o programa sai.

Normalmente, os programas saem quando a execução atinge o final do código e não há mais linhas para executar. No entanto, quando `sys.exit ()` é chamado, o programa termina imediatamente e pára de testar novas mensagens (porque você vai querer consertar seus programas de codificação se um teste falhar!).

Mas se os valores na mensagem e descriptografados forem iguais, a execução do programa ignora o bloco da instrução `if` e a chamada para `sys.exit ()`. O programa continua em loop até terminar a execução de todos os seus testes. Depois que o loop termina, o programa executa a linha 34, que você sabe que está fora do loop da linha 9 porque tem um recuo diferente. A linha 34 imprime 'Teste de cifra de transposição passado'.

Chamando a função main ()

Como com nossos outros programas, queremos verificar se o programa está sendo importado como um módulo ou sendo executado como o programa principal.

```
37. # Se transpositionTest.py for executado (em vez de importado como um  
módulo), ligue  
38. # a função main ():  
39. if __name__ == '__main__':  
40.     main ()
```

As linhas 39 e 40 fazem o truque, verificando se a variável especial `__name__` está definida como '`__main__`' e, em caso afirmativo, chamando a função `main ()`.

Testando o programa de teste

Nós escrevemos um programa que testa os programas de codificação de transposição, mas como sabemos que o programa de teste funciona? E se o programa de teste tiver um bug e indicar que os programas de codificação de

transposição funcionam quando eles realmente não funcionam?

Podemos testar o programa de teste adicionando propositadamente bugs às funções de criptografia ou descriptografia. Então, se o programa de teste não detectar um problema, sabemos que ele não está sendo executado como esperado.

Para adicionar um bug ao programa, abrimos *transpositionEncrypt.py* e adicionamos + 1 à linha 36:

*transposição
Encrypt.py*

35. # Mover currentIndex sobre:

36. currentIndex += chave + 1

Agora que o código de criptografia está quebrado, quando executamos o programa de teste, ele deve imprimir um erro, como este:

Teste nº 1:

```
"JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIU  
..."
```

Incompatibilidade com a chave 1 e a mensagem

```
JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIC  
XXTBFOFHVSIGBWIBBHKGKUWHEUDYONYTZVKNVVTYZPDDMIDK  
WAWXZSFTMJNLJOKKIJXLWAPCQNYCIQOFTEAUHRJODKLGRIZSJB  
OMSCEEEXLUSCFHNELYPYKCNYTOUQGBFSRDDMVIGXNYPHQPIST/  
Descriptografado como:
```

```
JQDKZACYCPTRLHBQEWLWGQRIITGHVZCEZAAIFBZXBOHSGWBHKC  
WWXSTJLOKJLACNCQFEUROKGISBQBQPGVZKWGMYRMCELSFNLPF
```

O programa de teste falhou na primeira mensagem depois que inserimos propositadamente um bug, por isso sabemos que ele está funcionando exatamente como planejamos!

Resumo

Você pode usar suas novas habilidades de programação para mais do que apenas escrever programas. Você também pode programar o computador para testar os programas que você escreve para garantir que eles funcionem para diferentes entradas. Escrever código para testar o código é uma prática comum.

Neste capítulo, você aprendeu como usar a função `random.randint()` para

produzir números pseudo-aleatórios e como usar `random.seed()` para redefinir a semente para criar mais números pseudo-aleatórios. Embora os números pseudo-aleatórios não sejam aleatórios o suficiente para serem usados em programas de criptografia, eles são bons o suficiente para serem usados no programa de testes deste capítulo.

Você também aprendeu a diferença entre uma lista e referência de lista e que a função `copy.deepcopy()` criará cópias de valores de lista em vez de valores de referência. Além disso, você aprendeu como a função `random.shuffle()` pode embaralhar a ordem dos itens em um valor de lista ao embaralhar os itens da lista no local usando referências.

Todos os programas que criamos até agora criptografam apenas mensagens curtas. No [Capítulo 10](#), você aprenderá a criptografar e descriptografar arquivos inteiros.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Se você executou o programa a seguir e imprimiu o número 8, o que imprimiria da próxima vez que você o executasse?

```
importação aleatória  
random.seed(9)  
print(random.randint(1, 10))
```

2. O que o programa a seguir imprime?

```
spam = [1, 2, 3]  
ovos = spam  
ham = ovos  
presunto [0] = 99  
imprimir (ham == spam)
```

3. Qual módulo contém a função `deepcopy()`?

4. O que o programa a seguir imprime?

```
cópia de importação  
spam = [1, 2, 3]  
eggs = copy.deepcopy (spam)  
ham = copy.deepcopy (ovos)
```

presunto [0] = 99
imprimir (ham == spam)

10

ARQUIVOS DE CRIPTOGRAFIA E DECRYPTING

“Por que a polícia de segurança pega pessoas e as tortura? Para obter suas informações. E os discos rígidos não resistem à tortura. Você precisa dar ao disco rígido uma maneira de resistir. Isso é criptografia.

—Patrick Ball, Grupo de Análise de Dados de Direitos Humanos



Nos capítulos anteriores, nossos programas trabalharam apenas em pequenas mensagens que digitamos diretamente no código-fonte como valores de string. O programa de codificação que faremos neste capítulo permitirá que você criptografe e decodifique arquivos inteiros, que podem ter milhões de caracteres em tamanho.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A função open ()
- Lendo e escrevendo arquivos
- Os métodos de objeto de arquivo write () , close () e read ()
- A função os.path.exists ()
- Os métodos de string upper () , lower () e title ()
- Os métodos startswith () e endswith () string
- O módulo de tempo e a função time.time ()

Arquivos de texto simples

O programa de codificação do arquivo de transposição criptografa e descriptografa arquivos de texto simples (não formatados). Esses são os tipos de arquivos que só possuem dados de texto e geralmente têm a extensão de arquivo

.txt . Você pode escrever seus próprios arquivos de texto com programas como o Notepad no Windows, oTextEdit no macOS e o gedit no Linux. (Programas de processamento de texto também podem produzir arquivos de texto simples, mas lembre-se de que eles não salvam nenhuma fonte, tamanho, cor ou outra formatação.) Você pode até mesmo usar o editor de arquivos do IDLE salvando os arquivos com uma extensão .txt em vez da habitual extensão .py .

Para alguns exemplos, você pode baixar arquivos de texto em <https://www.nostarch.com/crackingcodes/> . Eses arquivos de texto de amostra são de livros que estão agora em domínio público e são legais para download e uso. Por exemplo, o romance clássico de Mary Shelley, *Frankenstein*, tem mais de 78.000 palavras em seu arquivo de texto! Digitar este livro em um programa de criptografia levaria muito tempo, mas usando o arquivo baixado, o programa pode fazer a criptografia em alguns segundos.

Código-fonte para o programa de codificação do arquivo de transposição

Assim como no programa de teste de transposição de códigos , o programa de codificação de arquivos de transposição importa os arquivos *transpositionEncrypt.py* e *transpositionDecrypt.py* para que ele possa chamar as funções *encryptMessage ()* e *decryptMessage ()* . Como resultado, você não precisa digitar novamente o código para essas funções no novo programa.

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File** . Digite o seguinte código no editor de arquivos e salve-o como *transpositionFileCipher.py* . Então faça o download do *frankenstein.txt* em <https://www.nostarch.com/crackingcodes/> e coloque este arquivo na mesma pasta que o arquivo *transpositionFileCipher.py* . Pressione F5 para executar o programa.

transposição
FileCipher.py

1. # Cifra de Transposição Criptografar / Descriptografar Arquivo
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. tempo de importação, os, sys, transpositionEncrypt, transpositionDecrypt
- 5
6. def main ():

```
7. inputFilename = 'frankenstein.txt'
8. # TENHA CUIDADO! Se um arquivo com o nome outputFilename já existir,
9. # este programa irá sobrescrever esse arquivo:
10. outputFilename = 'frankenstein.encrypted.txt'
11. myKey = 10
12. myMode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.
13
14. # Se o arquivo de entrada não existir, o programa termina cedo:
15. se não os.path.exists (inputFilename):
16.     print ('O arquivo% s não existe. Saindo ...'% (inputFilename))
17.     sys.exit ()
18
19. # Se o arquivo de saída já existir, dê ao usuário uma chance de sair:
20. if os.path.exists (outputFilename):
21.     print ('Isto sobrescreve o arquivo% s. (C) ontinue ou (Q) uit?'%
22.           (outputFilename))
22.     resposta = entrada ('>')
23.     se não response.lower (). Startswith ('c'):
24.         sys.exit ()
25
26. # Leia na mensagem do arquivo de entrada:
27. fileObj = open (inputFilename)
28. content = fileObj.read ()
29. fileObj.close ()
30
31. print ('% sing ...'% (myMode.title ()))
32
33. # Mede quanto tempo a criptografia / descriptografia leva:
34. startTime = time.time ()
35. if myMode == 'encriptar':
36.     translated = transpositionEncrypt.encryptMessage (myKey, content)
37. elif myMode == 'decifrar':
38.     translated = transpositionDecrypt.decryptMessage (myKey, conteúdo)
39. totalTime = round (time.time () - startTime, 2)
40. print ('% de tempo:% s segundos'% (myMode.title (), totalTime))
41.
42. # Escreva a mensagem traduzida para o arquivo de saída:
```

```
43. outputFileObj = open (outputFilename, 'w')
44. outputFileObj.write (tradução)
45. outputFileObj.close ()
46
47. print ('Feito% sing% s (% s caracteres).' % (MyMode, inputFilename,
len (conteúdo)))
48. print ('o arquivo% sed é% s.' % (MyMode.title (), outputFilename))
49.
50
51. # Se transpositionCipherFile.py for executado (em vez de importado como
um módulo),
52. # chama a função main ():
53. if __name__ == '__main__':
54. main ()
```

Execução de exemplo do programa de codificação de arquivo de transposição

Quando você executa o programa *transpositionFileCipher.py* , ele deve produzir esta saída:

Criptografando ...

Tempo de criptografia: 1,21 segundos

Feito criptografando frankenstein.txt (441034 caracteres).

O arquivo criptografado é frankenstein.encrypted.txt.

Um novo arquivo *frankenstein.encrypted.txt* é criado na mesma pasta que *transpositionFileCipher.py* . Quando você abrir este arquivo com o editor de arquivos do IDLE, você verá o conteúdo criptografado do *frankenstein.py* . Deve ser algo como isto:

PtFiyedleo a arnvmt eneeGLchongnes Mmuyedlsu0 # uiSHTGA r sy, nt ys
s nuaoGeL

sc7s

- recorte -

Depois de ter um texto criptografado, você pode enviá-lo para outra pessoa para descriptografá-lo. O destinatário também precisará ter o programa de codificação do arquivo de transposição.

Para descriptografar o texto, faça as seguintes alterações no código-fonte (em

negrito) e execute o programa de codificação do arquivo de transposição novamente:

```
7. inputFilename = 'frankenstein.encrypted.txt'  
8. # TENHA CUIDADO! Se um arquivo com o nome outputFilename já existir,  
9. # este programa irá sobrescrever esse arquivo:  
10. outputFilename = 'frankenstein.decrypted.txt'  
11. myKey = 10  
12. myMode = 'descriptografar' # Configure para 'criptografar' ou  
'descriptografar'.
```

Desta vez, quando você executar o programa, um novo arquivo chamado *frankenstein.decrypted.txt* que é idêntico ao arquivo *frankenstein.txt* original aparecerá na pasta.

Trabalhando com arquivos

Antes de nos aprofundarmos no código para *transpositionFileCipher.py* , vamos examinar como o Python trabalha com arquivos. As três etapas para ler o conteúdo de um arquivo são abrir o arquivo, ler o conteúdo do arquivo em uma variável e fechar o arquivo. Da mesma forma, para gravar um novo conteúdo em um arquivo, você deve abrir (ou criar) o arquivo, gravar o novo conteúdo e fechar o arquivo.

Abrindo Arquivos

O Python pode abrir um arquivo para ler ou gravar usando a função `open()` . O primeiro parâmetro da função `open()` é o nome do arquivo a ser aberto. Se o arquivo estiver na mesma pasta que o programa Python, você pode usar apenas o nome do arquivo, como '*thetimemachine.txt*' . O comando para abrir o *arquivo thetimemachine.txt*, se ele existisse na mesma pasta do seu programa Python, ficaria assim:

```
fileObj = open ('thetimemachine.txt')
```

Um objeto de arquivo é armazenado na variável `fileObj` , que será usada para ler ou gravar no arquivo.

Você também pode especificar o *caminho absoluto* do arquivo, que inclui as pastas e pastas pai em que o arquivo se encontra. Por exemplo, 'C: \\ Usuários \\ Al \\ frankenstein.txt' (no Windows) e '/ Users /Al/frankenstein.txt ' (no macOS e no Linux) são caminhos absolutos. Lembre-se de que, no Windows, a barra invertida (\) deve ser ignorada, digitando outra barra invertida antes dela.

Por exemplo, se você quiser abrir o arquivo *frankenstein.txt* , passe o caminho do arquivo como uma string para o primeiro parâmetro da função open () (e formate o caminho absoluto de acordo com o sistema operacional):

```
fileObj = open ('C: \\ Usuários \\ Al \\ frankenstein.txt')
```

O objeto de arquivo tem vários métodos para gravar, ler e fechar o arquivo.

Escrevendo e Fechando Arquivos

Para o programa de criptografia, depois de ler o conteúdo do arquivo de texto, você precisará gravar o conteúdo criptografado (ou descriptografado) em um novo arquivo, o que fará usando o método write () .

Para usar write () em um objeto de arquivo, você precisa abrir o objeto de arquivo no modo de gravação, o que você faz passando open () a string 'w' como um segundo argumento. (Esse segundo argumento é um *parâmetro opcional* porque a função open () ainda pode ser usada sem passar dois argumentos.) Por exemplo, insira a seguinte linha de código no shell interativo:

```
>>> fileObj = open ('spam.txt', 'w')
```

Esta linha cria um arquivo chamado *spam.txt* no modo de gravação para que você possa editá-lo. Se um arquivo com o mesmo nome existir onde a função open () cria o novo arquivo, o arquivo antigo é sobreescrito, portanto, tenha cuidado ao usar o método open () no modo de gravação.

Com o *spam.txt* agora aberto no modo de gravação, você pode escrever no arquivo chamando o método write () nele. O método write () recebe um argumento: uma string de texto para gravar no arquivo. Digite o seguinte no shell interativo para escrever "Hello, world!" para *spam.txt* :

```
>>> fileObj.write ('Olá, mundo!')
```

13

Passando a string 'Olá, mundo!' para o método write () grava essa string no arquivo *spam.txt* e, em seguida, imprime 13 , o número de caracteres na string gravados no arquivo.

Quando você terminar de trabalhar com um arquivo, precisará informar ao Python que acabou de fazer o arquivo chamando o método close () no objeto file:

```
>>> fileObj.close ()
```

Há também um modo de acréscimo, que é como o modo de gravação, exceto que o modo de acréscimo não sobrescreve o arquivo. Em vez disso, as strings são gravadas no final do conteúdo que já está no arquivo. Embora não o usemos neste programa, você pode abrir um arquivo no modo append passando a string 'a' como segundo argumento para open () .

Se você receber uma mensagem de erro io.UnsupportedOperation: not readable ao tentar chamar write () em um objeto de arquivo, talvez não tenha aberto o arquivo no modo de gravação. Quando você não inclui o parâmetro opcional da função open () , ele abre automaticamente o objeto de arquivo no modo de leitura ('r'), que permite usar somente o método read () no objeto de arquivo.

Lendo de um arquivo

O método read () retorna uma string contendo todo o texto no arquivo. Para experimentar, vamos ler o arquivo *spam.txt* que criamos anteriormente com o método write () . Execute o seguinte código do shell interativo:

```
>>> fileObj = open ('spam.txt', 'r')
>>> content = fileObj.read ()
>>> imprimir (conteúdo)
Olá Mundo!
>>> fileObj.close ()
```

O arquivo é aberto e o objeto de arquivo criado é armazenado na variável fileObj . Depois de ter o objeto de arquivo, você pode ler o arquivo usando o método read () e armazená-lo na variável de conteúdo , que será impressa. Quando você terminar o objeto file, feche-o com close () .

Se você receber a mensagem de erro IOError: [Errno 2] Nenhum arquivo ou diretório , verifique se o arquivo realmente está onde você pensa que está e verifique se digitou o nome do arquivo e da pasta corretamente. (*Diretório* é outra palavra para *pasta* .)

Usaremos open () , read () , write () e close () nos arquivos que abriremos para criptografar ou descriptografar em *transpositionFileCipher.py* .

Configurando a função main ()

A primeira parte do programa *transpositionFileCipher.py* deve parecer familiar. A linha 4 é uma declaração de importação para os programas *transpositionEncrypt.py* e *transpositionDecrypt.py* , bem como para os módulos time , os e sys do Python. Então nós começamos main () configurando algumas

variáveis para usar no programa.

```
1. # Cifra de Transposição Criptografar / Descriptografar Arquivo
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. tempo de importação, os, sys, transpositionEncrypt, transpositionDecrypt
5
6. def main():
7.     inputFilename = 'frankenstein.txt'
8.     # TENHA CUIDADO! Se um arquivo com o nome outputFilename já existir,
9.     # este programa irá sobrescrever esse arquivo:
10.    outputFilename = 'frankenstein.encrypted.txt'
11.    myKey = 10
12.    myMode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.
```

A variável `inputFilename` contém uma string do arquivo para ler e o texto criptografado (ou descriptografado) é gravado no arquivo denominado em `outputFilename`. A cifra de transposição usa um inteiro para uma chave, que é armazenada em `myKey`. O programa espera que o `myMode` armazene 'encrypt' ou 'decrypt' para informar que ele criptografa ou descriptografa o arquivo `inputFilename`. Mas antes que possamos ler o arquivo `inputFilename`, precisamos verificar se ele existe usando `os.path.exists()`.

Verificando se existe um arquivo

Ler arquivos é sempre inofensivo, mas você precisa ter cuidado ao gravar arquivos. Chamar a função `open()` no modo de gravação em um nome de arquivo que já existe substitui o conteúdo original. Usando a função `os.path.exists()`, seus programas podem verificar se o arquivo já existe ou não.

A função `os.path.exists()`

A função `os.path.exists()` usa um único argumento de string para um nome de arquivo ou um caminho para um arquivo e retorna `True` se o arquivo já existir e `False` se não existir. A função `os.path.exists()` existe dentro do módulo `path`, que existe dentro do módulo `os`, então quando importamos o módulo `os`, o módulo `path` também é importado.

Digite o seguinte no shell interativo:

```
>>> import os
❶ >>> os.path.exists ('spam.txt')
```

Falso

```
>>> os.path.exists ('C:\\ Windows \\ System32 \\ calc.exe') # Windows  
Verdade
```

```
>>> os.path.exists ('/ usr / local / bin / idle3') # macOS
```

Falso

```
>>> os.path.exists ('/ usr / bin / idle3') # Linux
```

Falso

Neste exemplo, a função `os.path.exists()` confirma que o arquivo `calc.exe` existe no Windows. Obviamente, você só obterá esses resultados se estiver executando o Python no Windows. Lembre-se de escapar da barra invertida em um caminho de arquivo do Windows digitando outra barra invertida antes dela. Se você estiver usando o macOS, somente o exemplo do macOS retornará True e somente o último exemplo retornará True para Linux. Se o caminho completo do arquivo não for fornecido, o Python verificará o diretório de trabalho atual. Para o shell interativo do IDLE, esta é a pasta em que o Python está instalado.

Verificando se o arquivo de entrada existe com a função `os.path.exists()`

Usamos a função `os.path.exists()` para verificar se o nome do arquivo em `inputFilename` existe. Caso contrário, não temos nenhum arquivo para criptografar ou descriptografar. Fazemos isso nas linhas 14 a 17:

```
14. # Se o arquivo de entrada não existir, o programa terminará cedo:  
15. se não os.path.exists (inputFilename):  
16. print ('O arquivo% s não existe. Saindo ...'% (inputFilename))  
17. sys.exit ()
```

Se o arquivo não existir, exibiremos uma mensagem para o usuário e depois sairemos do programa.

Usando métodos de string para tornar a entrada do usuário mais flexível

Em seguida, o programa verifica se existe um arquivo com o mesmo nome de `outputFilename` e, em caso afirmativo, solicita ao usuário que digite C, caso queira continuar executando o programa, ou Q para encerrar o programa. Como um usuário pode digitar várias respostas, como 'c' , 'C' ou até mesmo a palavra 'Continuar' , queremos ter certeza de que o programa aceitará todas essas versões. Para fazer isso, usaremos mais métodos de string.

Os métodos de string `upper()`, `lower()` e `title()`

Os métodos de string upper () e lower () retornarão a string na qual são chamados em letras maiúsculas ou minúsculas, respectivamente. Digite o seguinte no shell interativo para ver como os métodos funcionam na mesma string:

```
>>> 'Hello'.upper()  
'OLÁ'  
>>> 'Hello'.lower()  
'Olá'
```

Assim como os métodos lower () e upper () retornam uma string em letras minúsculas ou maiúsculas, o método title () retorna uma string no caso do título. *O caso do título* é onde o primeiro caractere de cada palavra é maiúsculo e o restante dos caracteres são minúsculos. Digite o seguinte no shell interativo:

```
>>> 'hello'.title()  
'Olá'  
>>> 'HELLO'.title()  
'Olá'  
>>> 'extra! extra! o homem morde o tubarão! '.title()  
'Extra! Extra! O homem morde o tubarão!'
```

Usaremos title () um pouco mais tarde no programa para formatar as mensagens que emitimos para o usuário.

Os métodos de sequência startswith () e endswith ()

O método startswith () retorna True se seu argumento de string for encontrado no começo da string. Digite o seguinte no shell interativo:

```
>>> 'hello'.startswith('h')  
Verdade  
>>> 'hello'.startswith('H')  
Falso  
>>> spam = 'albert'  
❶ >>> spam.startswith('Al')  
Verdade
```

O método startswith () faz distinção entre maiúsculas e minúsculas e também pode ser usado em strings com vários caracteres **❶**.

O método string endswith () é usado para verificar se um valor de string termina com outro valor de string especificado. Digite o seguinte no shell interativo:

```
>>> 'Olá mundo!'.endswith('mundo!')  
Verdade  
❷ >>> 'Olá mundo!'.endswith('mundo')  
Falso
```

Os valores da string devem corresponder perfeitamente. Observe que a falta do ponto de exclamação em 'mundo' faz com que endswith () retorne False .

Usando esses métodos de seqüência de caracteres no programa

Conforme observado, queremos que o programa aceite qualquer resposta que comece com um C, independentemente da capitalização. Isso significa que queremos que o arquivo seja sobreescrito se o usuário digitar c , continue , C ou outra cadeia que comece com C. Usaremos os métodos de string lower () e startswith () para tornar o programa mais flexível ao tomar a entrada do usuário:

```
19. # Se o arquivo de saída já existir, dê ao usuário uma chance de sair:  
20. if os.path.exists (outputFilename):  
21.     print ('Isto sobrescreve o arquivo% s. (C) ontinue ou (Q) uit?'%  
(outputFilename))  
22.     resposta = entrada ('>')  
23.     se não response.lower ().Startswith ('c'):  
24.         sys.exit ()
```

Na linha 23, pegamos a primeira letra da string e verificamos se é um C usando o método startswith () . O método startswith () que usamos é sensível a maiúsculas e minúsculas e verifica a minúscula 'c' , então usamos o método lower () para modificar a capitalização da string de resposta para sempre ser minúscula. Se o usuário não inseriu uma resposta começando com um C , então startswith () retorna False , o que faz com que a instrução if seja avaliada como True (por causa da não na instrução if), e sys.exit () é chamado para finalizar o programa. Tecnicamente, o usuário não precisa digitar Q para sair; qualquer string que não comece com C faz com que a função sys.exit () seja chamada para sair do programa.

Lendo o arquivo de entrada

Na linha 27, começamos a usar os métodos de objeto de arquivo discutidos no início deste capítulo.

```
26. # Leia na mensagem do arquivo de entrada:  
27. fileObj = open (inputFilename)
```

```
28. content = fileObj.read ()  
29. fileObj.close ()  
30  
31. print ('% sing ...%' (myMode.title ()))
```

As linhas 27 a 29 abrem o arquivo armazenado em inputFilename , lêem seu conteúdo na variável de conteúdo e fecham o arquivo. Depois de ler no arquivo, a linha 31 exibe uma mensagem para o usuário informando que a criptografia ou descriptografia foi iniciada. Como myMode deve conter a string 'encrypt' ou 'decrypt' , chamar o método de string title () capitaliza a primeira letra da string em myMode e une a string na string '% sing' , portanto, exibe 'Encrypting'. . . ' or ' Decrypting ... '.

Medindo o tempo que levou para criptografar ou descriptografar
Criptografar ou descriptografar um arquivo inteiro pode levar muito mais tempo do que uma string curta. Um usuário pode querer saber quanto tempo leva o processo para um arquivo. Podemos medir o comprimento do processo de criptografia ou descriptografia usando o módulo de tempo .

O módulo de tempo e a função time.time ()

A função time.time () retorna a hora atual como um valor flutuante do número de segundos desde 1º de janeiro de 1970. Esse momento é chamado de *Unix Epoch* . Digite o seguinte no shell interativo para ver como esta função funciona:

```
>>> tempo de importação  
>>> time.time ()  
1540944000.7197928  
>>> time.time ()  
1540944003.4817972
```

Como time.time () retorna um valor flutuante, ele pode ser preciso para um *milissegundo* (isto é, 1/1000 de segundo). Claro, os números que time.time () exibe depende do momento em que você chama essa função e pode ser difícil de interpretar. Pode não estar claro que 1540944000.7197928 é terça-feira, 30 de outubro de 2018, aproximadamente às 5 da tarde . No entanto, a função time.time () é útil para comparar o número de segundos entre as chamadas para time.time () . Podemos usar essa função para determinar por quanto tempo um programa está sendo executado.

Por exemplo, se você subtrair os valores de ponto flutuante retornados quando

eu chamei time.time () anteriormente no shell interativo, você obteria a quantidade de tempo entre essas chamadas enquanto eu estava digitando:

```
>>> 1540944003.4817972 - 1540944000.7197928  
2.7620043754577637
```

Se você precisar escrever um código que manipule datas e horas, consulte <https://www.nostarch.com/crackingcodes/> para obter informações sobre o módulo datetime .

Usando a função time.time () no programa

Na linha 34, time.time () retorna a hora atual para armazenar em uma variável chamada startTime . As linhas 35 a 38 chamam encryptMessage () ou decryptMessage () , dependendo se 'encrypt' ou 'decrypt' está armazenado na variável myMode .

```
33. # Mede quanto tempo a criptografia / descriptografia leva:  
34. startTime = time.time()  
35. if myMode == 'encriptar':  
36.     translated = transpositionEncrypt.encryptMessage (myKey, content)  
37. elif myMode == 'decifrar':  
38.     translated = transpositionDecrypt.decryptMessage (myKey, conteúdo)  
39. totalTime = round (time.time () - startTime, 2)  
40. print ('% de tempo:% s segundos'% (myMode.title (), totalTime))
```

A linha 39 chama time.time () novamente depois que o programa descriptografa ou criptografa e subtrai startTime a partir da hora atual. O resultado é o número de segundos entre as duas chamadas para time.time () . A expressão time.time () - startTime é avaliada como um valor que é passado para a função round () , que é arredondada para os dois pontos decimais mais próximos, porque não precisamos de precisão de milissegundos para o programa. Este valor é armazenado em totalTime . A linha 40 usa emenda de seqüência de caracteres para imprimir o modo de programa e exibe ao usuário a quantidade de tempo que levou para o programa criptografar ou descriptografar.

Escrevendo o arquivo de saída

O conteúdo do arquivo criptografado (ou descriptografado) agora é armazenado na variável traduzida . Mas esta string é esquecida quando o programa termina, então queremos armazenar a string em um arquivo para ter mesmo depois que o programa terminar de rodar. O código nas linhas 43 a 45 faz isso abrindo um

novo arquivo (e passando 'w' para a função open ()) e então chamando o método do objeto write () :

42. # Escreva a mensagem traduzida para o arquivo de saída:

43. outputFileObj = open (outputFilename, 'w')

44. outputFileObj.write (tradução)

45. outputFileObj.close ()

Em seguida, as linhas 47 e 48 imprimem mais mensagens para o usuário, indicando que o processo está concluído e o nome do arquivo gravado:

47. print ('Feito% sing% s (% s caracteres).' % (MyMode, inputFilename, len (conteúdo)))

48. print ('o arquivo% sed é% s.' % (MyMode.title (), outputFilename))

A linha 48 é a última linha da função main () .

Chamando a função main ()

As linhas 53 e 54 (que são executadas depois que a instrução def na linha 6 é executada) chamam a função main () se este programa estiver sendo executado em vez de ser importado:

51. # Se transpositionCipherFile.py for executado (em vez de importado como um módulo),

52. # chama a função main ():

53. if __name__ == '__main__':

54. main ()

Isso é explicado em detalhes em “ [A variável __name__](#) ” na [página 95](#) .

Resumo

Parabéns! Não havia muito para o programa *transpositionFileCipher.py* além das funções open () , read () , write () e close () , que permitiam criptografar grandes arquivos de texto em um disco rígido. Você aprendeu a usar a função os.path.exists () para verificar se um arquivo já existe. Como você viu, podemos ampliar os recursos de nossos programas importando suas funções para uso em novos programas. Isso aumenta muito nossa capacidade de usar computadores para criptografar informações.

Você também aprendeu alguns métodos de string úteis para tornar um programa mais flexível ao aceitar a entrada do usuário e como usar o módulo de tempo para medir a velocidade com que o programa é executado.

Ao contrário do programa de cifra de César, a cifra do arquivo de transposição tem muitas chaves possíveis para atacar simplesmente usando força bruta. Mas se pudermos escrever um programa que reconheça o inglês (em oposição a cadeias de texto sem sentido), o computador poderá examinar a saída de milhares de tentativas de descriptografia e determinar qual chave pode descriptografar uma mensagem com êxito em inglês. Você aprenderá como fazer isso no [Capítulo 11](#).

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Qual é o correto: os.exists () ou os.path.exists () ?
2. Quando é a época do Unix?
3. O que as expressões a seguir avaliam?

'Foobar'.startswith (' Foo ')
'Foo'.startswith (' Foobar ')
'Foobar'.startswith (' foo ')
'bar'.endswith (' Foobar ')
'Foobar'.endswith (' bar ')
"A rápida raposa marrom saltou sobre o cão preguiçoso amarelo. '. Title ()

11

DETECÇÃO DE INGLÊS PROGRAMMATICAMENTE

O funcionário diz algo mais longo e mais complicado.

Depois de um tempo, Waterhouse (agora usando seu chapéu de criptoanalista, procurando por significados em meio a aparente aleatoriedade, seus circuitos neurais explorando as redundâncias no sinal) percebe que o homem está falando inglês com forte sotaque.

-Neal Stephenson, Cryptonomicon



Anteriormente, usamos a codificação do arquivo de transposição para criptografar e descriptografar arquivos inteiros, mas ainda não tentamos escrever um programa de força bruta para hackar a cifra. As mensagens criptografadas com a cifra do arquivo de transposição podem ter milhares de chaves possíveis, que seu computador ainda pode facilmente forçar a força bruta, mas você teria que examinar centenas de descriptografia para encontrar o texto simples correto. Como você pode imaginar, isso pode ser um grande problema, mas há uma solução alternativa.

Quando o computador descriptografa uma mensagem usando a chave errada, a seqüência resultante é um texto incorreto em vez de texto em inglês. Podemos programar o computador para reconhecer quando uma mensagem descriptografada é o inglês. Dessa forma, se o computador descriptografar usando a chave errada, ele saberá continuar e tentar a próxima chave possível. Eventualmente, quando o computador tenta uma chave que descriptografa o texto em inglês, ele pode parar e chamar a atenção para essa chave, evitando que você tenha que examinar milhares de descriptografia incorretas.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- O tipo de dados do dicionário
- O método `split()`
- O valor `Nenhum`
- Erros de divisão por zero
- As funções `float()`, `int()` e `str()` e divisão inteira
- O método da lista `append()`
- Argumentos padrão
- Calculando porcentagens

Como um computador pode entender inglês?

Um computador não consegue entender o inglês, pelo menos, não da maneira que os seres humanos entendem o inglês. Computadores não entendem matemática, xadrez ou rebeliões humanas, mais do que um relógio entende a hora do almoço. Computadores apenas executam instruções uma após a outra. Mas essas instruções podem imitar comportamentos complexos para resolver problemas de matemática, vencer no xadrez ou caçar os futuros líderes da resistência humana.

Idealmente, o que precisamos criar é uma função Python (vamos chamá-la de função `isEnglish()`) para a qual podemos passar uma string e obter um valor de retorno `True` se a string for texto em inglês ou `False` se for um texto aleatório. Vamos ver alguns textos em inglês e algum texto lixo para ver quais padrões eles podem ter:

Robôs são seus amigos. Exceto pelo RX-686. Ela vai tentar te comer.
ai-pey e. xrx ne augura iirl6 Rtiyt fhubE6d hrSei t8..ow eo.telyoosEs t

Observe que o texto em inglês é composto de palavras que você encontraria em um dicionário, mas o texto de lixo não é. Como as palavras geralmente são separadas por espaços, uma maneira de verificar se uma string de mensagem é o inglês é dividir a mensagem em strings menores em cada espaço e verificar se cada substring é uma palavra do dicionário. Para dividir as strings de mensagem em substrings, podemos usar o método `string` Python chamado `split()`, que verifica onde cada palavra começa e termina procurando espaços entre os caracteres. (“[O método `split\(\)`](#)” na [página 150](#) aborda isso com mais detalhes.) Podemos comparar cada substring para cada palavra no dicionário usando uma instrução `if`, como no código a seguir:

```
if word == 'aardvark' ou word == 'abacus' ou word == 'abandonar' ou palavra == 'abandonado' ou palavra == 'abreviar' ou palavra == 'abreviatura' ou palavra == 'abdomen' ou ...
```

Poderíamos escrever código assim, mas provavelmente não o faríamos porque seria tedioso digitar tudo. Felizmente, podemos usar *arquivos de dicionário em inglês*, que são arquivos de texto que contêm quase todas as palavras em inglês. Vou lhe fornecer um arquivo de dicionário para usar, então só precisamos escrever a função `isEnglish()` que verifica se as subsequências na mensagem estão no arquivo do dicionário.

Nem toda palavra existe em nosso arquivo de dicionário. O arquivo do dicionário pode estar incompleto; por exemplo, pode não ter a palavra *aardvark*.

Há também decifrações perfeitamente boas que podem ter palavras não inglesas, como o *RX-686* em nosso exemplo de frase em inglês. O texto simples também pode estar em um idioma diferente, mas vamos supor que seja em inglês por enquanto.

Portanto, a função `isEnglish()` não será infalível, mas se a *maioria* das palavras no argumento da string forem palavras em inglês, é uma boa aposta que a string seja texto em inglês. Há uma probabilidade muito baixa de que um texto cifrado descriptografado usando a chave errada seja descriptografado em inglês.

Você pode baixar o arquivo de dicionário que usaremos para este livro (que tem mais de 45.000 palavras) em <https://www.nostarch.com/crackingcodes/>. O arquivo de texto do dicionário lista uma palavra por linha em maiúsculas. Abra e você verá algo assim:

```
AARHUS
AARON
ABAABA
ABACK
ABAFT
ABANDONO
ABANDONADO
ABANDONANDO
ABANDONO
ABANDONOS
- recorte -
```

Nossa função `isEnglish()` dividirá uma string descriptografada em substrings individuais e verificará se cada substring existe como uma palavra no arquivo do dicionário. Se um determinado número das substrings forem palavras em inglês, identificaremos esse texto como inglês. E se o texto estiver em inglês, há uma boa chance de que tenhamos descriptografado o texto cifrado com a chave correta.

Código Fonte para o Módulo Detectar Inglês

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como `detectEnglish.py`. Certifique-se de que `dictionary.txt` esteja no mesmo diretório que `detectEnglish.py` ou este código não funcionará. Pressione F5 para executar o programa.

detectEnglish.py

```
1. # Detectar módulo em inglês
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. # Para usar, digite este código:
5. importação # importEnglish
6. # detectEnglish.isEnglish (someString) # Retorna True ou False
7. # (Deve haver um arquivo "dictionary.txt" neste diretório com todos
8. # palavras em inglês, uma palavra por linha. Você pode baixar isso de
9. # https://www.nostarch.com/crackingcodes/.)
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower ()
+ '\t\n'
12
13. def loadDictionary():
14.     dictionaryFile = open ('dictionary.txt')
15.     englishWords = {}
16.     para palavra em dictionaryFile.read (). Split ('\n'):
17.         englishWords [palavra] = nenhuma
18.     dictionaryFile.close ()
19.     voltar para o inglês
20
21. ENGLISH_WORDS = loadDictionary ()
22
23
24. def getEnglishCount (mensagem):
25.     message = message.upper ()
26.     message = removeNonLetters (mensagem)
27.     possíveisWords = message.split ()
28.
29. se possívelWords == []:
30.     return 0.0 # Sem palavras, então retorne 0.0
31
32. correspondências = 0
33. por palavra em possíveisPalavras:
34. se palavra em ENGLISH_WORDS:
35.     correspondências + = 1
```

```
36. return float (correspondências) / len (possibleWords)
37
38
39. def removeNonLetters (mensagem):
40.     lettersOnly = []
41.     para símbolo na mensagem:
42.         se símbolo em LETTERS_AND_SPACE:
43.             lettersOnly.append (símbolo)
44.     return " .join (letrasApenas)
45
46
47. Def isEnglish (message, wordPercentage = 20, letterPercentage = 85):
48. # Por padrão, 20% das palavras devem existir no arquivo do dicionário, e
49. # 85% de todos os caracteres da mensagem devem ser letras ou espaços
50. # (não pontuação ou números).
51. wordsMatch = getEnglishCount (mensagem) * 100> = wordPercentage
52. numLetters = len (removeNonLetters (mensagem))
53. messageLettersPercentage = float (numLetters) / len (mensagem) * 100
54. lettersMatch = messageLettersPercentage> = letterPercentage
55. return wordsMatch e lettersMatch
```

Exemplo de Execução do Módulo Detectar Inglês

O programa *detectEnglish.py* que escreveremos neste capítulo não será executado sozinho. Em vez disso, outros programas de criptografia importarão o *detectorEnglish.py* para que possam chamar a função *detectEnglish.isEnglish ()*, que retorna True quando a string é determinada como inglês. É por isso que não damos a função *detectEnglish.py* a *main ()*. As outras funções em *detectEnglish.py* são funções auxiliares que a função *isEnglish ()* chamará. Todo o trabalho que faremos neste capítulo permitirá que qualquer programa importe o módulo *detectEnglish* com uma declaração de importação e use as funções nele.

Você também poderá usar este módulo no shell interativo para verificar se uma string individual está em inglês, como mostrado aqui:

```
>>> import detectEnglish
>>> detectEnglish.isEnglish ('Is this sentence English text?')
Verdade
```

Neste exemplo, a função determinou que a string 'Is this sentence English text?' é de fato em inglês, então retorna True .

Instruções e Configuração de Constantes

Vamos ver a primeira parte do programa *detectEnglish.py* . As primeiras nove linhas de código são comentários que fornecem instruções sobre como usar este módulo.

```
1. # Detectar módulo em inglês
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. # Para usar, digite este código:
5. importação # importEnglish
6. # detectEnglish.isEnglish (someString) # Retorna True ou False
7. # (Deve haver um arquivo "dictionary.txt" neste diretório com todos
8. # palavras em inglês, uma palavra por linha. Você pode baixar isso de
9. # https://www.nostarch.com/crackingcodes/.)
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower ()
+ '\t \n'
```

As primeiras nove linhas de código são comentários que fornecem instruções sobre como usar este módulo. Eles lembram os usuários que este módulo não funcionará a menos que um arquivo chamado *dictionary.txt* esteja no mesmo diretório que *detectEnglish.py* .

As linhas 10 e 11 configuraram algumas variáveis como constantes, que estão em maiúsculas. Como você aprendeu no [Capítulo 5](#) , as constantes são variáveis cujos valores nunca devem ser alterados depois de definidos. UPPERLETTERS é uma constante contendo as 26 letras maiúsculas que configuramos por conveniência e para economizar tempo de digitação. Usamos a constante UPPERLETTERS para configurar LETTERS_AND_SPACE , que contém todas as letras maiúsculas e minúsculas do alfabeto, bem como o caractere de espaço, o caractere de tabulação e o caractere de nova linha. Em vez de digitar todas as letras maiúsculas e minúsculas, apenas concatenamos UPPERLETTERS com as letras minúsculas retornadas por UPPERLETTERS.lower () e os caracteres adicionais que não são letras. Os caracteres tab e newline são representados com os caracteres de escape \ t e \ n .

O tipo de dados do dicionário

Antes de continuarmos com o resto do código *detectEnglish.py* , você precisa aprender mais sobre o tipo de dados do dicionário para entender como converter o texto no arquivo em um valor de string. O tipo de dados do *dicionário* (não confundir com o arquivo do dicionário) armazena valores, que podem conter vários outros valores, assim como as listas. Nas listas, usamos um índice inteiro para recuperar itens na lista, como *spam* [42] . Mas para cada item no valor do dicionário, usamos uma chave para recuperar um valor. Embora possamos usar apenas inteiros para recuperar itens de uma lista, a chave em um valor de dicionário pode ser um número inteiro ou uma string, como *spam* ['hello'] ou *spam* [42] . Os dicionários nos permitem organizar os dados de nosso programa com mais flexibilidade do que as listas e não armazenam itens em nenhuma ordem específica. Em vez de usar colchetes como as listas, os dicionários usam chaves. Por exemplo, um dicionário vazio se parece com isso {} .

NOTA

Tenha em mente que os arquivos de dicionário e os valores de dicionário são conceitos completamente diferentes que, por acaso, possuem nomes semelhantes. Um valor do dicionário Python pode conter vários outros valores. Um arquivo de dicionário é um arquivo de texto contendo palavras em inglês.

Os itens de um dicionário são digitados como *pares de valor-chave* , nos quais as chaves e os valores são separados por dois-pontos. Vários pares de valores-chave são separados por vírgulas. Para recuperar valores de um dicionário, use colchetes com a chave entre eles, semelhante à indexação com listas. Para tentar recuperar valores de um dicionário usando chaves, insira o seguinte no shell interativo:

```
>>> spam = {'key1': 'Isto é um valor', 'key2': 42}  
>>> spam ['key1']  
'Isso é um valor'
```

Primeiro, configuramos um dicionário chamado *spam* com dois pares de valores-chave. Em seguida, acessamos o valor associado à chave string 'key1' , que é outra string. Como nas listas, você pode armazenar todos os tipos de dados nos seus dicionários.

Note que, como nas listas, as variáveis não armazenam valores de dicionário; em vez disso, eles armazenam referências para dicionários. O código de exemplo a

seguir mostra duas variáveis com referências ao mesmo dicionário:

```
>>> spam = {'olá': 42}  
>>> ovos = spam  
>>> ovos ['olá'] = 99  
>>> ovos  
{'olá': 99}  
>>> spam  
{'olá': 99}
```

A primeira linha de código configura outro dicionário chamado spam , dessa vez com apenas um par de valores-chave. Você pode ver que ele armazena um valor inteiro 42 associado à chave 'hello' string. A segunda linha atribui o par de valores-chave do dicionário a outra variável chamada ovos . Você pode então usar ovos para alterar o valor do dicionário original associado à chave 'hello' para 99 . Agora, as duas variáveis, ovos e spam , devem retornar o mesmo par de valores-chave do dicionário com o valor atualizado.

A diferença entre dicionários e listas

Os dicionários são como listas de várias maneiras, mas existem algumas diferenças importantes:

- Os itens do dicionário não estão em qualquer ordem. Não há primeiro ou último item em um dicionário como há em uma lista.
- Você não pode concatenar dicionários com o operador + . Se você quiser adicionar um novo item, use a indexação com uma nova chave. Por exemplo, foo ['uma nova chave'] = 'uma string' .
- As listas só têm valores de índices inteiros que variam de 0 ao comprimento da lista menos um, mas os dicionários podem usar qualquer chave. Se você tiver um dicionário armazenado em uma variável spam , poderá armazenar um valor em spam [3] sem precisar de valores para spam [0] , spam [1] ou spam [2] .

Adicionando ou alterando itens em um dicionário

Você também pode adicionar ou alterar valores em um dicionário usando as chaves do dicionário como índices. Digite o seguinte no shell interativo para ver como isso funciona:

```
>>> spam = {42: 'olá'}
```

```
>>> imprimir (spam [42])
```

Olá

```
>>> spam [42] = 'tchau'
```

```
>>> imprimir (spam [42])
```

Tchau

Este dicionário possui um valor de string de dicionário existente 'hello' associado à chave 42 . Podemos atribuir novamente um novo valor de string 'adeus' a essa chave usando spam [42] = 'tchau' . Atribuir um novo valor a uma chave de dicionário existente sobrescreve o valor original associado a essa chave. Por exemplo, quando tentamos acessar o dicionário com a chave 42 , obtemos o novo valor associado a ele.

E assim como as listas podem conter outras listas, os dicionários também podem conter outros dicionários (ou listas). Para ver um exemplo, insira o seguinte no shell interativo:

```
>>> foo = {'fizz': {'nome': 'Al', 'idade': 144}, 'moo': ['a', 'brown', 'cow']}
```

```
>>> foo ['fizz']
```

```
{'age': 144, 'name': 'Al'}
```

```
>>> foo ['fizz'] ['nome']
```

```
'Al'
```

```
>>> foo ['moo']
```

```
['a', 'brown', 'cow']
```

```
>>> foo ['moo'] [1]
```

```
'Castanho'
```

Este código de exemplo mostra um dicionário (chamado foo) que contém duas chaves 'fizz' e 'moo' , cada uma correspondendo a um valor e tipo de dados diferentes. A tecla 'fizz' contém outro dicionário, e a tecla 'moo' contém uma lista. (Lembre-se que valores de dicionário não mantêm seus itens em ordem. É por isso que foo ['fizz'] mostra os pares de valores-chave em uma ordem diferente da que você digitou.) Para recuperar um valor de um dicionário aninhado em outro dicionário , você primeiro especifica a chave do maior conjunto de dados que deseja acessar usando colchetes, o que é 'fizz' neste exemplo. Em seguida, use os colchetes novamente e insira a chave 'name' correspondente ao valor da string aninhada 'Al' que você deseja recuperar.

Usando a função len () com dicionários

A função len () mostra o número de itens em uma lista ou o número de caracteres

em uma string. Também pode mostrar o número de itens em um dicionário. Digite o seguinte código no shell interativo para ver como usar a função len () para contar itens em um dicionário:

```
>>> spam = {}  
>>> len (spam)  
0  
>>> spam ['name'] = 'Al'  
>>> spam ['pet'] = 'Zophie o gato'  
>>> spam ['age'] = 89  
>>> len (spam)  
3
```

A primeira linha deste exemplo mostra um dicionário vazio chamado spam . A função len () mostra corretamente que o comprimento deste dicionário vazio é 0 . No entanto, depois de introduzir os três valores a seguir, 'Al' , 'Zophie the cat' e 89 , no dicionário, a função len () agora retorna 3 para os três pares de valores-chave que você acabou de atribuir à variável .

Usando o em Operador com Dicionários

Você pode usar o operador in para ver se existe uma determinada chave em um dicionário. É importante lembrar que o operador in verifica as chaves, não os valores. Para ver esse operador em ação, insira o seguinte no shell interativo:

```
>>> ovos = {'foo': 'leite', 'barra': 'pão'}  
>>> 'foo' em ovos  
Verdade  
>>> 'leite' em ovos  
Falso  
>>> 'blá blá blá' em ovos  
Falso  
>>> 'blá blá blá' não em ovos  
Verdade
```

Configuramos um dicionário chamado eggs com alguns pares de valores-chave e, em seguida, verificamos quais chaves existem no dicionário usando o operador in . A chave 'foo' é uma chave nos ovos , então True é retornado. Considerando que 'milk' retorna False porque é um valor, não uma chave, 'blah blah blah' é avaliada como False porque nenhum item desse tipo existe neste dicionário. O operador não em funciona também com os valores do dicionário,

que você pode ver no último comando.

Encontrar itens é mais rápido com dicionários do que com listas

Imagine os seguintes valores de lista e dicionário no shell interativo:

```
>>> listVal = ['spam', 'eggs', 'bacon']
>>> dictionaryVal = {'spam': 0, 'eggs': 0, 'bacon': 0}
```

O Python pode avaliar a expressão 'bacon' no dictionaryVal um pouco mais rápido que 'bacon' em listVal . Isso ocorre porque, para uma lista, o Python deve começar no início da lista e, em seguida, percorrer cada item em ordem até encontrar o item de pesquisa. Se a lista é muito grande, o Python deve pesquisar por vários itens, um processo que pode levar muito tempo.

Mas um dicionário, também chamado de *tabela de hash* , traduz diretamente onde, na memória do computador, o valor do par de valores-chave é armazenado, motivo pelo qual os itens de um dicionário não têm um pedido. Não importa o tamanho do dicionário, encontrar qualquer item sempre leva o mesmo tempo.

Essa diferença de velocidade é dificilmente perceptível ao procurar listas curtas e dicionários. Mas nosso módulo detectEnglish terá dezenas de milhares de itens, e a palavra de expressão em ENGLISH_WORDS , que usaremos em nosso código, será avaliada várias vezes quando a função isEnglish () for chamada. A utilização de valores de dicionário acelera esse processo ao manipular um grande número de itens.

Usando Loops com Dicionários

Você também pode iterar sobre as teclas em um dicionário usando loops for , assim como pode percorrer os itens de uma lista. Digite o seguinte no shell interativo:

```
>>> spam = {'name': 'Al', 'age': 99}
>>> for k in spam:
...     print(k, spam[k])
...
Idade 99
nome Al
```

Para usar uma instrução for para iterar as chaves em um dicionário, comece com a palavra-chave for . Defina a variável k , use a palavra-chave in para especificar

que você deseja repetir o spam e terminar a instrução com dois pontos. Como você pode ver, inserir print (k, spam [k]) retorna cada chave no dicionário junto com seu valor correspondente.

Implementando o arquivo do dicionário

Agora vamos retornar ao *detectorEnglish.py* e configurar o arquivo do dicionário. O arquivo de dicionário fica no disco rígido do usuário, mas, a menos que carreguemos o texto nesse arquivo como um valor de string, nosso código Python não poderá usá-lo. Vamos criar uma função auxiliar `loadDictionary ()` para fazer isso:

```
13. def loadDictionary ():  
14.     dictionaryFile = open ('dictionary.txt')  
15.     englishWords = {}
```

Primeiro, obtemos o objeto de arquivo do dicionário chamando `open ()` e passando a string do nome de arquivo 'dictionary.txt' . Em seguida, nomeamos a variável do dicionário `englishWords` e definimos como um dicionário vazio.

Armazenaremos todas as palavras no arquivo de dicionário (o arquivo que armazena as palavras em inglês) em um valor de dicionário (o tipo de dados Python). Os nomes semelhantes são lamentáveis, mas os dois são completamente diferentes. Mesmo que pudéssemos ter usado uma lista para armazenar os valores de string de cada palavra no arquivo do dicionário, estamos usando um dicionário porque o operador `in` trabalha mais rápido nos dicionários do que nas listas.

Em seguida, você aprenderá sobre o método de string `split ()` , que usaremos para dividir nosso arquivo de dicionário em substrings.

O método `split ()`

O método de string `split ()` usa uma string e retorna uma lista de várias strings, dividindo a string passada em cada espaço. Para ver um exemplo de como isso funciona, insira o seguinte no shell interativo:

```
>>> 'Minha mãe muito energética acabou de nos servir Nutella.'. Split ()  
['Meu', 'muito', 'energético', 'mãe', 'apenas', 'servido', 'nós', 'Nutella'].]
```

O resultado é uma lista de oito strings, uma string para cada uma das palavras na string original. Espaços são descartados dos itens na lista, mesmo se houver mais de um espaço. Você pode passar um argumento opcional para o método `split ()`

para ordenar que ele seja dividido em uma string diferente de um espaço. Digite o seguinte no shell interativo:

```
>>> 'helloXXXworldXXXhowXXXareXXyou?'. split ('XXX')
['hello', 'world', 'how', 'areXXyou?']
```

Observe que a string não possui espaços. Usar split ('XXX') divide a string original onde quer que 'XXX' ocorra, resultando em uma lista de quatro strings. A última parte da string, 'areXXyou?' não é dividido porque "XX" não é o mesmo que "XXX".

Dividindo o arquivo do dicionário em palavras individuais

Vamos voltar ao nosso código-fonte em *detectEnglish.py* para ver como dividimos a string no arquivo do dicionário e armazenamos cada palavra em uma chave.

16. para palavra em dictionaryFile.read (). Split ('\ n'):
17. englishWords [palavra] = nenhuma

Vamos dividir a linha 16. A variável dictionaryFile armazena o objeto de arquivo do arquivo aberto. A chamada do método dictionaryFile.read () lê o arquivo inteiro e o retorna como um valor de cadeia grande. Em seguida, chamamos o método split () nessa longa cadeia e dividimos em caracteres de nova linha. Como o arquivo de dicionário tem uma palavra por linha, a divisão em caracteres de nova linha retorna um valor de lista composto de cada palavra no arquivo de dicionário.

O loop for no início da linha itera sobre cada palavra para armazenar cada uma em uma chave. Mas não precisamos de valores associados às chaves, pois estamos usando o tipo de dados do dicionário, portanto, apenas armazenaremos o valor Nenhum para cada chave.

Nenhum é um tipo de valor que você pode atribuir a uma variável para representar a falta de um valor. Enquanto o tipo de dados booleano tem apenas dois valores, o NoneType tem apenas um valor, Nenhum . É sempre escrito sem aspas e com um *N* maiúsculo.

Por exemplo, digamos que você tivesse uma variável chamada quizAnswer , que detém a resposta de um usuário a uma pergunta de questionário pop true-false. Se o usuário pular uma pergunta e não respondê-la, faz mais sentido atribuir quizAnswer a None como um valor padrão, em vez de True ou False . Caso contrário, pode parecer que o usuário respondeu à pergunta quando não o fez. Da

mesma forma, as chamadas de função que saem, atingindo o final da função e não de uma instrução de retorno , são avaliadas como Nenhuma, porque não retornam nada.

A linha 17 usa a palavra que está sendo iterada como uma chave em englishWords e armazena None como um valor para essa chave.

Retornando os dados do dicionário

Depois que o loop for terminar, o dicionário englishWords deve ter dezenas de milhares de chaves nele. Neste ponto, fechamos o objeto de arquivo porque terminamos de ler e retornamos em inglésWords :

18. dictionaryFile.close ()
19. voltar para o inglês

Em seguida, chamamos loadDictionary () e armazenamos o valor do dicionário que retorna em uma variável denominada ENGLISH_WORDS :

21. ENGLISH_WORDS = loadDictionary ()

Nós queremos chamar loadDictionary () antes do resto do código no módulo detectEnglish , mas o Python deve executar a instrução def para loadDictionary () antes que possamos chamar a função. Esta é a razão pela qual a atribuição para o ENGLISH_WORDS vem depois do código da função loadDictionary () .

Contando o número de palavras inglesas na mensagem

As linhas 24 a 27 do código do programa definem a função getEnglishCount () , que recebe um argumento de string e retorna um valor flutuante indicando a proporção de palavras inglesas reconhecidas para o total de palavras. Nós vamos representar a razão como um valor entre 0,0 e 1,0 . Um valor de 0,0 significa que nenhuma das palavras na mensagem são palavras em inglês e 1.0 significa que todas as palavras na mensagem são palavras em inglês. Muito provavelmente, getEnglishCount () retornará um valor flutuante entre 0.0 e 1.0 . A função isEnglish () usa esse valor de retorno para determinar se deve ser avaliado como Verdadeiro ou Falso .

24. def getEnglishCount (mensagem):
25. message = message.upper ()
26. message = removeNonLetters (mensagem)
27. possíveisWords = message.split ()

Para codificar esta função, primeiro criamos uma lista de strings de palavras

individuais a partir da string na mensagem . A linha 25 converte a string em letras maiúsculas. Em seguida, a linha 26 remove os caracteres que não são letras da sequência, como números e pontuação, chamando removeNonLetters () . (Você verá como essa função funciona mais tarde.) Finalmente, o método split () na linha 27 divide a string em palavras individuais e armazena-as em uma variável denominada possibleWords .

Por exemplo, se a string 'Hello there. Como você está?' é passado após chamar getEnglishCount () , o valor armazenado em possíveis palavras - chave após as linhas 25 a 27 executar seria ['HELLO', 'THERE', 'HOW', 'ARE', 'YOU'] .

Se a string na mensagem é composta de inteiros, como '12345' , a chamada para removeNonLetters () retornaria uma string em branco, a qual split () seria chamada para retornar uma lista vazia. No programa, uma lista vazia é o equivalente a zero palavras sendo inglês, o que poderia causar um erro de divisão por zero.

Erros de divisão por zero

Para retornar um valor de float entre 0,0 e 1,0 , dividimos o número de palavras em possíveisWords reconhecidas como inglês pelo número total de palavras em possíveisWords . Embora isso seja mais direto, precisamos garantir que o PossibleWords não seja uma lista vazia. Se possibleWords estiver vazio, significa que o número total de palavras em possibleWords é 0 .

Como na matemática, dividir por zero não tem significado, dividir por zero no Python resulta em um erro de divisão por zero. Para ver um exemplo desse erro, digite o seguinte no shell interativo:

```
>>> 42/0
Traceback (última chamada mais recente):
Arquivo "<pyshell # 0>", linha 1, em <module>
42/0
ZeroDivisionError: divisão por zero
```

Você pode ver que 42 dividido por 0 resulta em um ZeroDivisionError e uma mensagem explicando o que deu errado. Para evitar um erro de divisão por zero, precisamos garantir que a lista de possíveis palavras-chave nunca fique vazia.

A linha 29 verifica se o possívelWords é uma lista vazia e a linha 30 retorna 0,0, se não houver palavras na lista.

29. se possívelWords == []:

```
30. return 0.0 # Sem palavras, então retorne 0.0
```

Essa verificação é necessária para evitar um erro de divisão por zero.

Contando as correspondências da palavra em inglês

Para produzir a proporção de palavras em inglês para o total de palavras, dividiremos o número de palavras em possíveis palavras que são reconhecidas como inglês pelo número total de palavras em possíveis palavras. Para fazer isso, precisamos contar o número de palavras inglesas reconhecidas em possíveis palavras-chave. A linha 32 define a variável `correspondências` a 0 . A linha 33 usa o loop `for` para iterar sobre cada palavra em possíveisPalavras e verificar se a palavra existe no dicionário ENGLISH_WORDS . Se a palavra existir no dicionário, o valor em `correspondências` é incrementado na linha 35.

```
32. correspondências = 0
```

```
33. por palavra em possíveisPalavras:
```

```
34. se palavra em ENGLISH_WORDS:
```

```
35. correspondências += 1
```

Depois que o loop `for` for concluído, o número de palavras inglesas na string é armazenado na variável `matches` . Lembre-se de que estamos confiando que o arquivo do dicionário seja preciso e completo para que o módulo `detectEnglish` funcione corretamente. Se uma palavra não estiver no arquivo de texto do dicionário, ela não será contada como inglês, mesmo que seja uma palavra real. Por outro lado, se uma palavra for digitada incorretamente no dicionário, palavras que não são inglesas podem ser contadas accidentalmente como palavras reais.

No momento, o número de palavras em possíveis palavras que são reconhecidas como inglês e o número total de palavras em possíveisPalavras é representado por números inteiros. Para retornar um valor flutuante entre 0.0 e 1.0 , dividindo esses dois inteiros, precisaremos alterar um ou outro para um float.

As funções float(), int() e str() Functions e Integer Division

Vamos ver como mudar um inteiro para um float porque os dois valores que precisamos dividir para encontrar a proporção são ambos inteiros. O Python 3 sempre faz uma divisão regular, independentemente do tipo de valor, enquanto o Python 2 executa a divisão inteira quando ambos os valores na operação de divisão são inteiros. Como os usuários podem usar o Python 2 para importar `detectEnglish.py` , precisaremos passar pelo menos uma variável inteira para

`float()` para garantir que um `float` seja retornado ao fazer a divisão. Isso garante que a divisão regular seja executada, independentemente da versão do Python usada. Este é um exemplo de tornar o código *compatível* com as versões anteriores.

Embora não os usemos neste programa, vamos revisar algumas outras funções que convertem valores em outros tipos de dados. A função `int()` retorna uma versão inteira de seu argumento e a função `str()` retorna uma string. Para ver como essas funções funcionam, insira o seguinte no shell interativo:

```
>>> flutuador(42)
42,0
>>> int(42,0)
42
>>> int(42,7)
42
>>> int('42 ')
42
>>> str(42)
'42'
>>> str(42,7)
'42 .7 '
```

Você pode ver que a função `float()` altera o inteiro 42 em um valor flutuante. A função `int()` pode transformar os floats 42.0 e 42.7 em inteiros, truncando seus valores decimais, ou pode transformar um valor de string '42' em um inteiro. A função `str()` altera valores numéricos em valores de string. Essas funções são úteis se você precisar de um valor equivalente a um tipo de dados diferente.

Encontrando a Razão das Palavras Inglesas na Mensagem

Para encontrar a proporção de palavras em inglês para o total de palavras, dividimos o número de correspondências que encontramos pelo número total de possíveisWords . A linha 36 usa o operador / para dividir esses dois números:

36. return float (correspondências) / len (possibleWords)

Depois que passamos as correspondências inteiras para a função `float()`, ela retorna uma versão flutuante desse número, que dividimos pelo tamanho da lista de possíveis palavras-chave.

A única maneira de retornar o `float (matches) / len (possíveisWords)` levaria a

um erro de divisão por zero se len (possibleWords) fosse avaliado como 0 . A única maneira que seria possível, se possível, as palavras do AdWords eram uma lista vazia. No entanto, as linhas 29 e 30 especificamente verificam este caso e retornam 0.0 se a lista estiver vazia. Se possível, as palavras foram definidas para a lista vazia, a execução do programa nunca passaria da linha 30, por isso podemos ter certeza de que a linha 36 não causará um ZeroDivisionError .

Removendo caracteres não-letra

Certos caracteres, como números ou sinais de pontuação, farão com que a detecção de palavras falhe, porque as palavras não ficarão exatamente como estão escritas em nosso arquivo de dicionário. Por exemplo, se a última palavra na mensagem for "você". e não removemos o período no final da string, não seria contado como uma palavra em inglês porque "você" não seria escrito com um ponto no arquivo do dicionário. Para evitar essa má interpretação, números e sinais de pontuação precisam ser removidos.

A função getEnglishCount () explicada anteriormente chama a função removeNonLetters () em uma string para remover quaisquer números e caracteres de pontuação dela.

```
39. def removeNonLetters (mensagem):  
40.     lettersOnly = []  
41.     para símbolo na mensagem:  
42.         se símbolo em LETTERS_AND_SPACE:  
43.             lettersOnly.append (símbolo)
```

A linha 40 cria uma lista em branco chamada lettersOnly e a linha 41 usa um loop for para fazer um loop sobre cada caractere no argumento da mensagem . Em seguida, o loop for verifica se o caractere existe na cadeia LETTERS_AND_SPACE . Se o caractere for um número ou um sinal de pontuação, ele não existirá na string LETTERS_AND_SPACE e não será adicionado à lista.Se o caractere existir na string, ele será adicionado ao final da lista usando o método append () , que veremos a seguir.

O método de lista append ()

Quando adicionamos um valor ao final de uma lista, dizemos que estamos *anexando* o valor à lista. Isso é feito com listas tão frequentemente em Python que há um método de lista append () que usa um único argumento para anexar ao final da lista. Digite o seguinte no shell interativo:

```
>>> ovos = []
>>> eggs.append ('hovercraft')
>>> ovos
['hovercraft']
>>> eggs.append ('enguias')
>>> ovos
['hovercraft', 'enguias']
```

Depois de criar uma lista vazia chamada eggs , podemos inserir eggs.append ('hovercraft') para adicionar o valor da string 'hovercraft' a essa lista. Então, quando entramos em ovos , ele retorna o único valor armazenado nessa lista, que é ' hovercraft ' . Se você usar append () novamente para adicionar 'eels' ao final da lista, os ovos agora retornam 'hovercraft' seguido por 'eels' . Da mesma forma, podemos usar o método de lista append () para adicionar itens à lista lettersOnly que criamos em nosso código anteriormente. Isso é o quelettersOnly.append (symbol) na linha 43 faz no loop for .

Criando uma sequência de letras

Depois de terminar o loop for , lettersOnly deve ser uma lista de cada letra e caractere de espaço da cadeia de mensagens original . Como uma lista de strings de um caractere não é útil para encontrar palavras em inglês, a linha 44 une as cadeias de caracteres da lista lettersOnly em uma sequência e a retorna:

44. return " .join (letrasApenas)

Para concatenar os elementos da lista em lettersOnly em uma string grande, chamamos o método de string join () em uma string em branco " . Isso une as strings em lettersOnly com uma string em branco entre elas. Este valor de string é então retornado como o valor de retorno da função removeNonLetters () .

Detectando Palavras Inglesas

Quando uma mensagem é descriptografada com a chave errada, ela geralmente produz muito mais caracteres não-letra e não-espaço do que os encontrados em uma mensagem típica em inglês. Além disso, as palavras que produz frequentemente serão aleatórias e não encontradas em um dicionário de palavras inglesas. A função isEnglish () pode verificar esses dois problemas em uma determinada string.

47. def isEnglish (message, wordPercentage = 20, letterPercentage = 85):
48. # Por padrão, 20% das palavras devem existir no arquivo do dicionário e

49. # 85% de todos os caracteres na mensagem devem ser letras ou espaços
50. # (não pontuação ou números).

A linha 47 configura a função `isEnglish()` para aceitar um argumento de `string` e retornar um valor booleano de `True` quando a `string` é texto em inglês e `False` quando não é. Esta função possui três parâmetros: `message`, `wordPercentage = 20` e `letterPercentage = 85`. O primeiro parâmetro contém a `string` a ser verificada, e os segundo e terceiro parâmetros definem porcentagens padrão para palavras e letras, que a `string` deve conter para ser confirmada como inglês. (Uma *porcentagem* é um número entre 0 e 100 que mostra quanto de algo é proporcional ao número total dessas coisas.) Exploraremos como usar argumentos padrão e calcular porcentagens nas seções a seguir.

Usando argumentos padrão

Às vezes, uma função quase sempre passa os mesmos valores quando chamada. Em vez de incluí-los para cada chamada de função, você pode especificar um argumento padrão na instrução `def` da função .

A instrução `def` da linha 47 tem três parâmetros, com argumentos padrão de 20 e 85 fornecidos para `wordPercentage` e `letterPercentage` , respectivamente. A função `isEnglish()` pode ser chamada com um a três argumentos. Se nenhum argumento for passado para `wordPercentage` ou `letterPercentage` , os valores atribuídos a esses parâmetros serão seus argumentos padrão.

Os argumentos padrão definem que porcentagem da `string` de mensagem precisa ser composta de palavras inglesas reais para `isEnglish()` para determinar que a mensagem é uma `string` em inglês e qual porcentagem da mensagem precisa ser composta de letras ou espaços em vez de números ou sinais de pontuação. Por exemplo, se `isEnglish()` for chamado com apenas um argumento, os argumentos padrão serão usados para `wordPercentage` (o inteiro 20) e `letterPercentage` (o inteiro 85).) parâmetros, o que significa que 20% da `string` precisa ser composta de palavras inglesas e 85% da `string` precisa ser composta de letras. Essas porcentagens funcionam para a detecção de inglês na maioria dos casos, mas você pode querer tentar outras combinações de argumentos em casos específicos em que `isEnglish()` precisa de limites mais fracos ou mais restritivos. Nessas situações, um programa pode apenas passar argumentos para `wordPercentage` e `letterPercentage` em vez de usar os argumentos padrão. [A Tabela 11-1](#) mostra as chamadas de função para `isEnglish()` e a que elas são equivalentes.

Tabela 11-1: Chamadas de função com e sem argumentos padrão

Chamada de função	Equivalente a
isEnglish ('Hello')	isEnglish ('Olá', 20, 85)
isEnglish ('Olá', 50)	isEnglish ('Olá', 50, 85)
isEnglish ('Hello', 50, 60)	isEnglish ('Hello', 50, 60)
isEnglish ('Hello', letterPercentage = 60)	isEnglish ('Olá', 20, 60)

Por exemplo, o terceiro exemplo na [Tabela 11-1](#) mostra que quando a função é chamada com o segundo e terceiro parâmetros especificados, o programa usará esses argumentos, não os argumentos padrão.

Calculando Porcentagens

Quando soubermos as porcentagens que nosso programa usará, precisaremos calcular as porcentagens da string de mensagem . Por exemplo, o valor da string 'Hello cat MOOSE fsdkl ewpin' tem cinco “palavras”, mas apenas três são em inglês. Para calcular a porcentagem de palavras inglesas nessa string, você divide o número de palavras em inglês pelo número total de palavras e multiplica o resultado por 100. A porcentagem de palavras inglesas em 'Hello cat MOOSE fsdkl ewpin' é $3/5 * 100$, que é de 60 por cento. [A Tabela 11-2](#) mostra alguns exemplos de porcentagens calculadas.

Tabela 11-2: Cálculo de porcentagens de palavras em inglês

Número de palavras inglesas	Número total de palavras	Razão de palavras inglesas	$\times \frac{100}{100} =$	Percentagem
3	5	0,6	$\times \frac{100}{100} =$	60
6	10	0,6	$\times \frac{100}{100} =$	60

300	500	0,6	$\times \frac{100}{100} = 60$
32	87	0,3678	$\times \frac{100}{100} = 36,78$
87	87	1,0	$\times \frac{100}{100} = 100$
0	10	0	$\times \frac{100}{100} = 0$

A porcentagem sempre estará entre 0% (ou seja, nenhuma palavra é o inglês) e 100% (ou seja, todas as palavras são em inglês). Nossa função isEnglish () considerará uma string inglesa se pelo menos 20% das palavras existirem no arquivo do dicionário e 85% dos caracteres na string forem letras ou espaços. Isso significa que a mensagem ainda será detectada como inglês, mesmo se o arquivo de dicionário não for perfeito ou se algumas palavras na mensagem forem algo diferente do que definimos como palavras em inglês.

A linha 51 calcula a porcentagem de palavras inglesas reconhecidas na mensagem passando a mensagem para getEnglishCount () , que faz a divisão e retorna um float entre 0.0 e 1.0 :

51. wordsMatch = getEnglishCount (mensagem) * 100 > = wordPercentage

Para obter uma porcentagem desse float, multiplique por 100 . Se o número resultante for maior ou igual ao parâmetro wordPercentage , True será armazenado em wordsMatch . (Lembre-se de que o operador de comparação > = avalia expressões para um valor booleano.) Caso contrário, False é armazenado em wordsMatch .

As linhas 52 a 54 calculam a porcentagem de caracteres alfabéticos na sequência de mensagens dividindo o número de caracteres alfabéticos pelo número total de caracteres na mensagem .

52. numLetters = len (removeNonLetters (mensagem))

53. messageLettersPercentage = float (numLetters) / len (mensagem) * 100

54. lettersMatch = messageLettersPercentage > = letterPercentage

Anteriormente no código, escrevemos a função removeNonLetters () para

encontrar todos os caracteres de letra e espaço em uma string, para que pudéssemos reutilizá-lo. A linha 52 chama removeNonLetters (message) para obter uma string apenas com os caracteres letter e space na mensagem . Passar esta string para len () deve retornar o número total de caracteres de letra e espaço na mensagem , que armazenamos como um inteiro na variável numLetters .

A linha 53 determina a porcentagem de letras obtendo uma versão flutuante do inteiro em numLetters e dividindo-a por len (mensagem) . O valor de retorno de len (mensagem) será o número total de caracteres na mensagem . Como discutido anteriormente, a chamada para float () é feita para garantir que a linha 53 execute uma divisão regular em vez de uma divisão inteira, caso o programador que importa o módulo detectEnglish esteja executando o Python 2.

A linha 54 verifica se a porcentagem em messageLettersPercentage é maior ou igual ao parâmetro letterPercentage . Essa expressão é avaliada como um valor booleano armazenado em lettersMatch .

Queremos isEnglish () para retornar verdadeira apenas se o wordsMatch e lettersMatch variáveis contêm Verdadeiro . A linha 55 combina esses valores em uma expressão usando o operador e :

55. return wordsMatch e lettersMatch

Se tanto o wordsMatch e lettersMatch variáveis são verdadeira , isEnglish () irá declarar a mensagem é o Inglês e retornar verdadeiro . Caso contrário, isEnglish () retornará False .

Resumo

A cifra do arquivo de transposição é uma melhoria em relação à cifra de César porque pode ter centenas ou milhares de chaves possíveis para mensagens, em vez de apenas 26 chaves diferentes. Mesmo que um computador não tenha problemas para descriptografar uma mensagem com milhares de chaves em potencial, precisamos escrever um código que possa determinar se uma string decriptografada é um inglês válido e, portanto, a mensagem original.

Neste capítulo, criamos um programa de detecção de inglês usando um arquivo de texto de dicionário para criar um tipo de dados de dicionário. O tipo de dados do dicionário é útil porque pode conter vários valores como uma lista. No entanto, ao contrário de uma lista, você pode indexar valores em um dicionário usando valores de string como chaves, em vez de apenas inteiros. A maioria das tarefas que você pode fazer com uma lista também pode ser feita com um

dicionário, como passar para len () ou usar os operadores in e not in nele. No entanto, o operador in é executado em um valor de dicionário muito grande, muito mais rápido do que em uma lista muito grande. Isso se mostrou particularmente útil para nós porque nossos dados de dicionário continham milhares de valores que precisávamos examinar rapidamente.

Este capítulo também introduziu o método split () , que pode dividir strings em uma lista de strings, e o tipo de dados NoneType, que possui apenas um valor: None . Este valor é útil para representar uma falta de valor.

Você aprendeu como evitar erros de divisão por zero ao usar o operador / ; converta valores em outros tipos de dados usando as funções int () , float () e str () ; e use o método de lista append () para adicionar um valor ao final de uma lista.

Quando você define funções, você pode fornecer alguns dos argumentos padrão dos parâmetros. Se nenhum argumento for passado para esses parâmetros quando a função for chamada, o programa usará o valor do argumento padrão, que pode ser um atalho útil em seus programas. No [Capítulo 12](#) , você aprenderá a hackear a cifra de transposição usando o código de detecção em inglês!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. O que o código a seguir imprime?

```
spam = {'name': 'Al'}  
print (spam ['nome'])
```

2. O que esse código imprime?

```
spam = {'eggs': 'bacon'}  
print ('bacon' em spam)
```

3. O que para código de loop iria imprimir os valores no dicionário de spam a seguir ?

```
spam = {'name': 'Zophie', 'species': 'gato', 'idade': 8}
```

4. O que a seguinte linha imprime?

```
print ('Olá, mundo!'. split ())
```

5. Qual será o código a seguir?

```
def spam (ovos = 42):  
    print (ovos)  
    spam ()  
    spam ('Hello')
```

6. Qual porcentagem de palavras nesta frase são palavras inglesas válidas?

"Se é flobulllar na mente para quarfalog os slings e as setas do guuuuuuuuur ultrajante."

12

ATACANDO A CIPHER DE TRANSPOSIÇÃO

"Ron Rivest, um dos inventores da RSA, acha que restringir a criptografia seria imprudente: 'É uma má política reprimir indiscriminadamente uma tecnologia só porque alguns criminosos podem usá-la em seu benefício'".

- Simon Singh, o livro de códigos



Neste capítulo, usaremos uma abordagem de força bruta para hackear a cifra de transposição. Das milhares de chaves que podem estar associadas à cifra de transposição, a chave correta deve ser a única que resulta em inglês legível. Usando o módulo *detectEnglish.py* que escrevemos no [Capítulo 11](#), nosso programa de criptografia de transposição nos ajudará a encontrar a chave correta.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Strings Multiline com citações triplas
- O método da string strip ()

Código-fonte do programa Hacker de criptografia de transposição

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como *transpositionHacker.py*. Como nos programas anteriores, verifique se o módulo

pyperclip.py , o módulo *transpositionDecrypt.py* ([Capítulo 8](#)) e o módulo *detectEnglish.py* e o arquivo *dictionary.txt* ([Capítulo 11](#)) estão no mesmo diretório que o arquivo *transpositionHacker.py* . Em seguida, pressione F5 para executar o programa.

transposição

Hacker.py

```
1. # Transpose Cipher Hacker
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip, detectEnglish, transpositionDecrypt
5
6. def main():
7. # Você pode querer copiar e colar este texto a partir do código-fonte em
8. # https://www.nostarch.com/crackingcodes/:
9. myMessage = """AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e enlh
na inde n n uhoretrm au ieu v er Ne2 gmanw, forwnlbsya apor tE.no
euarisfatt e mealefedhsppmgAnlnoe (c -ou) alat r lw oeb nglom, Ain
um dtes ilhetcdba. t tg eturmudg, tfl1e1 v nitiaicynhrCsaemie-sp
ncgHt nie cetrgmnoa yc r, ieaa toesa- e a0m82e1w shcnth ekh
gaecnpeutaiaeetgn iodhsd ro hAe snrsfccegrt NCsLc b17m8aEheideikfr
aBercaeu thllnrshicwsg etriebruaiss d iorr. """
10
11. hackedMessage = hackTransposition (myMessage)
12
13. se hackedMessage == None:
14. print ('Falha ao hackar criptografia')
15. mais:
16. print ('Copiando a mensagem hackeada para a área de transferência:')
17. print (hackedMessage)
18. pyperclip.copy (hackedMessage)
19
20
21. def hackTransposition (mensagem):
22. print ('Hacking ...')
23
24. Os programas # Python podem ser interrompidos a qualquer momento
```

pressionando

25. # Ctrl-C (no Windows) ou Ctrl-D (no macOS e no Linux):

26. print ('(Pressione Ctrl-C (no Windows) ou Ctrl-D (no macOS e Linux) para desistir a qualquer momento.) ')

27

28. Força bruta percorrendo todas as chaves possíveis:

29. para chave no intervalo (1, len (mensagem)):

30. print ("Tecla Tentativa #% s ...'% (chave))

31

32. decryptedText = transpositionDecrypt.decryptMessage (chave, mensagem)

33

34. se detectEnglish.isEnglish (decryptedText):

35. # Pergunte ao usuário se esta é a descriptografia correta:

36. print ()

37. print ('Possível violação de criptografia:')

38. print ('Tecla% s:% s'% (chave, decryptedText [: 100]))

39. print ()

40. print ('Digite D se feito, qualquer outra coisa para continuar hackeando:')

41. response = input ('>')

42.

43. if response.strip (). Upper (). Startswith ('D'):

44. return decryptedText

45

46. retorno Nenhum

47

48. if __name__ == '__main__':

49. main ()

Execução de Amostra do Programa Hacker de Cifra de Transposição

Quando você executa o programa *transpositionHacker.py* , a saída deve ficar assim:

Hacking ...

(Pressione Ctrl-C (no Windows) ou Ctrl-D (no macOS e no Linux) para sair a qualquer momento.)

Tentando a chave # 1 ...

Tentando a chave # 2 ...

Tentando a chave # 3 ...

Tentando a chave # 4 ...

Tentando a chave # 5 ...

Tentando chave # 6 ...

Hack de criptografia possível:

Chave 6: Augusta Ada King-Noel, Condessa de Lovelace (10 de dezembro de 1815 - 27

Novembro de 1852) foi um tapete Inglês

Digite D, se estiver pronto, mais alguma coisa para continuar hacking:

> D

Copiando mensagem hackeada para a área de transferência:

Augusta Ada King-Noel, Condessa de Lovelace (10 de dezembro de 1815 - 27 de novembro

1852) foi um matemático e escritor inglês, conhecido principalmente por seu trabalho em

O primeiro computador mecânico de uso geral de Charles Babbage, o Analytical Motor. Suas notas no motor incluem o que é reconhecido como o primeiro algoritmo destinado a ser realizado por uma máquina. Como resultado, ela é frequentemente

considerado como o primeiro programador de computador.

Depois de tentar a chave # 6, o programa retorna um trecho da mensagem descriptografada para o usuário confirmar que encontrou a chave certa. Neste exemplo, a mensagem parece promissora. Quando o usuário confirma que a descriptografia está correta, digitando D , o programa retorna toda a mensagem hackeada. Você pode ver que é uma nota biográfica sobre Ada Lovelace. (Seu algoritmo para calcular os números de Bernoulli, criado em 1842 e 1843, fez dela a primeira programadora de computador.) Se a decodificação for um falso positivo, o usuário pode pressionar qualquer outra coisa e o programa continuará tentando outras teclas.

Execute o programa novamente e pule a descriptografia correta pressionando qualquer coisa diferente de D. O programa assume que não encontrou a descriptografia correta e continua sua abordagem de força bruta através de outras chaves possíveis.

- recorte -

Tentando a chave # 417 ...

Tentando a chave # 418 ...

Tentando a chave # 419 ...

Falha ao hackar criptografia.

Eventualmente, o programa percorre todas as chaves possíveis e, em seguida, desiste, informando ao usuário que não foi possível hackear o texto cifrado.

Vamos dar uma olhada no código-fonte para ver como o programa funciona.

Importando os Módulos

As primeiras linhas do código informam ao usuário o que esse programa fará. A linha 4 importa vários módulos que escrevemos ou vimos nos capítulos anteriores: *pyperclip.py* , *detectEnglish.py* e *transpositionDecrypt.py* .

1. # Transpose Cipher Hacker
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar pyperclip, detectEnglish, transpositionDecrypt

O programa de transposição de hackers cifrados, contendo aproximadamente 50 linhas de código, é razoavelmente curto, porque grande parte dele existe em outros programas que estamos usando como módulos.

Cordas Multiline com Citações Triplas

A variável *myMessage* armazena o texto cifrado que estamos tentando invadir. A linha 9 armazena um valor de string que começa e termina com aspas triplas. Observe que é uma string muito longa.

6. def main ():
7. # Você pode querer copiar e colar este texto a partir do código-fonte em
8. # <https://www.nostarch.com/crackingcodes/>:
9. myMessage = """ "AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e enlh
na inde n n uhoretrm au ieu v er Ne2 gmanw, forwnlbsya apor tE.no
euarisfatt e mealefedhsppmgAnlnoe (c -ou) alat r lw oeb nglom, Ain
um dtes ilhetcdba. t tg eturmudg, tfl1e1 v nitiaicynhrCsaemie-sp
ncgHt nie cetrgmnoa yc r, ieaa toesa- e a0m82e1w shcnth ekh
gaecnpeutaiaeetgn iodhsd ro hAe snrsfcgegrt NCsLc b17m8aEheideikfr
aBercaeu thllnrshicwsg etriebruaiss d iorr. """ "

As cadeias de caracteres de aspas triplas também são chamadas de *cadeias de caracteres de múltiplas linhas* porque elas abrangem várias linhas e podem conter quebras de linha dentro delas. Cordas Multiline são útil para colocar strings grandes no código-fonte de um programa e porque aspas simples e duplas

não precisam ser escapadas dentro delas. Para ver um exemplo de uma cadeia multilinha, insira o seguinte no shell interativo:

```
>>> spam = """ Querida Alice,  
Por que você vestiu meu hamster com roupas de boneca?  
Eu olho para o Sr. Fuzz e penso: " Eu sei que isso foi feito por Alice " .  
Atenciosamente,  
Brienne """  
>>> imprimir (spam)  
Querida Alice,  
Por que você vestiu meu hamster com roupas de boneca?  
Eu olho para o Sr. Fuzz e penso: "Eu sei que isso foi feito por Alice".  
Atenciosamente,  
Brienne
```

Observe que esse valor de string, como nossa string de texto cifrado, abrange várias linhas. Tudo após as aspas triplas de abertura será interpretado como parte da string até que o programa atinja as aspas triplas que terminam. Você pode criar strings de múltiplas linhas usando três caracteres de aspas duplas ou três caracteres de aspas simples.

Exibindo os resultados de hackear a mensagem

O código ciphertext-hacking existe dentro da função `hackTransposition()` , que é chamada na linha 11 e que definiremos na linha 21. Essa função recebe um argumento de string: a mensagem criptografada em texto cifrado que estamos tentando hackear. Se a função puder cortar o texto cifrado, ele retornará uma string do texto descriptografado. Caso contrário, retorna o valor `Nenhum` .

```
11. hackedMessage = hackTransposition (myMessage)  
12  
13. se hackedMessage == None:  
14. print ('Falha ao hackar criptografia')  
15. mais:  
16. print ('Copiando a mensagem hackeada para a área de transferência:')  
17. print (hackedMessage)  
18. pyperclip.copy (hackedMessage)
```

A linha 11 chama a função `hackTransposition()` para retornar a mensagem hackeada se a tentativa for bem-sucedida ou o valor `None` se a tentativa não for bem-sucedida e armazenar o valor retornado em `hackedMessage` .

As linhas 13 e 14 informam ao programa o que fazer se a função não for capaz de hackear o texto cifrado. Se Nenhum foi armazenado em hackedMessage , o programa informa ao usuário que não conseguiu quebrar a criptografia na mensagem.

As próximas quatro linhas mostram o que o programa faz se a função for capaz de hackear o texto cifrado. A linha 17 imprime a mensagem descriptografada e a linha 18 a copia para a área de transferência. No entanto, para esse código funcionar, também precisamos definir a função hackTransposition () , que faremos a seguir.

Obtendo a mensagem hackeada

A função hackTransposition () inicia com algumas instruções print () .

21. def hackTransposition (mensagem):
22. print ('Hacking ...')
- 23
24. Os programas # Python podem ser interrompidos a qualquer momento pressionando
25. # Ctrl-C (no Windows) ou Ctrl-D (no macOS e no Linux):
26. print ('(Pressione Ctrl-C (no Windows) ou Ctrl-D (no macOS e Linux) para desistir a qualquer momento.) ')

Como o programa pode tentar muitas chaves, o programa exibe uma mensagem informando ao usuário que o hacking foi iniciado e que pode levar um momento para concluir o processo. A chamada print () na linha 26 diz ao usuário para pressionar ctrl -C (no Windows) ou ctrl -D (no macOS e no Linux) para sair do programa a qualquer momento. Você pode realmente pressionar essas teclas para sair de qualquer programa Python em execução.

As próximas duas linhas informam ao programa quais teclas devem ser percorridas, especificando o intervalo de chaves possíveis para a cifra de transposição:

28. Força bruta percorrendo todas as chaves possíveis:
29. para chave no intervalo (1, len (mensagem)):
30. print ('Tecla Tentativa #% s ...%' (chave))

As chaves possíveis para a cifra de transposição variam entre 1 e o comprimento da mensagem. O loop for na linha 29 executa a parte hacker da função com cada uma dessas chaves. A linha 30 usa a interpolação de strings para imprimir a

chave atualmente sendo testada, usando a interpolação de strings para fornecer feedback ao usuário.

Usando a função `decryptMessage()` no programa `transpositionDecrypt.py` que já escrevemos, a linha 32 obtém a saída descriptografada da chave atual que está sendo testada e a armazena na variável `decryptedText` :

32. `decryptedText = transpositionDecrypt.decryptMessage(chave, mensagem)`

A saída descriptografada no `decryptedText` será em inglês somente se a chave correta foi usada. Caso contrário, ele aparecerá como texto inválido.

Então o programa passa a string em descriptografado para a função `detectEnglish.isEnglish()` que escrevemos no [Capítulo 11](#) e imprime parte do `decryptedText`, a chave usada e as instruções para o usuário:

34. `se detectEnglish.isEnglish(decryptedText):`

35. # Pergunte ao usuário se esta é a descriptografia correta:

36. `print()`

37. `print('Possível violação de criptografia:')`

38. `print('Tecla% s:% s%'(chave, decryptedText[:100]))`

39. `print()`

40. `print('Digite D se feito, qualquer outra coisa para continuar hackeando:')`

41. `response = input('>')`

Mas só porque `detectEnglish.isEnglish()` retorna True e move a execução para a linha 35 não significa que o programa encontrou a chave correta. Pode ser um falso positivo, o que significa que o programa detectou algum texto como o inglês, que na verdade é lixo de texto. Para ter certeza, a linha 38 dá uma prévia do texto para que o usuário possa confirmar que o texto é realmente inglês. Ele usa a fatia `decryptedText[:100]` para imprimir os primeiros 100 caracteres de `decryptedText`.

O programa pausa quando a linha 41 é executada, aguarda o usuário digitar D ou qualquer outra coisa e, em seguida, armazena essa entrada como uma cadeia de caracteres em resposta .

O método da string strip ()

Quando um programa fornece instruções específicas ao usuário, mas o usuário não as segue exatamente, ocorre um erro. Quando o programa `transpositionHacker.py` solicita ao usuário que digite D para confirmar a mensagem hackeada, isso significa que o programa não aceita nenhuma entrada

além de D. Se um usuário inserir um espaço ou caractere adicional junto com D , o programa não o aceitará. Vamos ver como usar o método de string strip () para fazer o programa aceitar outras entradas, desde que sejam similares o suficiente para D.

O método de string strip () retorna uma versão da string com todos os caracteres de espaço em branco no início e no final da string removidos. Os *caracteres de espaço em branco* são o caractere de espaço, o caractere de tabulação e o caractere de nova linha. Digite o seguinte no shell interativo para ver como isso funciona:

```
>>> 'Hello'.strip()  
'Olá'  
>>> 'Olá' .strip()  
'Olá'  
>>> 'Olá mundo' .strip()  
'Olá Mundo'
```

Neste exemplo, strip () remove os caracteres de espaço no início ou no final das duas primeiras strings. Se uma string como 'Hello World' incluir espaços no início e no final da string, o método os removerá de ambos os lados, mas não removerá espaços entre outros caracteres.

O método strip () também pode ter um argumento de string passado para ele que instrui o método a remover caracteres diferentes do espaço em branco desde o início e o final da string. Para ver um exemplo, insira o seguinte no shell interativo:

```
>>> 'aaaaaHELLOaa'.strip(' a ')  
'OLÁ'  
>>> 'ababaHELLObaba'.strip(' ab ')  
'OLÁ'  
>>> 'abccabcbacbXYZabcXYZacccab'.strip(' abc ')  
'XYZabcXYZ'
```

Observe que passar os argumentos da string 'a' e 'ab' remove esses caracteres quando eles ocorrem no início ou no final da string. No entanto, strip () não remove caracteres incorporados no meio da string. Como você pode ver no terceiro exemplo, a string 'abc' permanece em ' XYZabcXYZ' .

Aplicando o strip () String Method

Vamos retornar ao código-fonte em *transpositionHacker.py* para ver como aplicar `strip()` no programa. A linha 43 define uma condição usando a instrução `if` para dar ao usuário alguma flexibilidade de entrada:

```
43. if response.strip().Upper().Startswith('D'):
44. return decryptedText
```

Se a condição para a declaração fosse simplesmente `response == 'D'`, o usuário teria que digitar D exatamente e nada mais para finalizar o programa. Por exemplo, se o usuário inserir 'd' , 'D' ou 'Done' , a condição será `False` e o programa continuará verificando outras chaves em vez de retornar a mensagem hackeada.

Para evitar esse problema, a string em resposta remove os espaços em branco do início ou fim da string com a chamada para `strip()`. Em seguida, a string que `response.strip()` avalia tem o método `upper()` chamado. Independentemente de o usuário inserir 'd' ou 'D' , a string retornada de `upper()` sempre será capitalizada como 'D' . Adicionar flexibilidade ao tipo de entrada que o programa pode aceitar facilita o uso.

Para fazer o programa aceitar a entrada do usuário que começa com 'D', mas é uma palavra completa, usamos `startswith()` para verificar apenas a primeira letra. Por exemplo, se o usuário inserir 'done' como resposta , o espaço em branco será removido e, em seguida, a string 'done' será passada para `upper()` . Depois que `upper()` capitaliza toda a string para 'DONE' , a string é passada para `startswith()` , que retorna `True` porque a string começa com a subcadeia 'D' .

Se o usuário indicar que a seqüência descriptografada está correta, a função `hackTransposition()` na linha 44 retorna o texto descriptografado.

Não Hackear a Mensagem

A linha 46 é a primeira linha após o loop `for` iniciado na linha 29:

```
46. retorno Nenhum
```

Se a execução do programa atingir este ponto, significa que o programa nunca alcançou a declaração de retorno na linha 44, o que aconteceria se o texto corretamente descriptografado nunca fosse encontrado para nenhuma das chaves que foram tentadas. Nesse caso, a linha 46 retorna o valor `None` para indicar que o hacking falhou.

Chamando a função `main()`

As linhas 48 e 49 chamam a função main () se este programa foi executado por si próprio em vez de ser importado por outro programa usando sua função hackTransposition () :

```
48. if __name__ == '__main__':
49.     main()
```

Lembre-se que a variável `__name__` é definida pelo Python. A função `main ()` não será chamada se `transpositionHacker.py` for importado como um módulo.

Resumo

Como no [Capítulo 6](#), este capítulo foi curto porque a maior parte do código já foi escrita em outros programas. Nossa programa de hackers pode usar funções de outros programas, importando-os como módulos.

Você aprendeu como usar aspas triplas para incluir um valor de string que abrange várias linhas no código-fonte. Você também aprendeu que o método de `string.strip ()` é útil para remover espaços em branco ou outros caracteres do início ou final de uma string.

Usando o programa `detectEnglish.py` nos poupou muito tempo, teríamos que passar manualmente inspecionando cada saída descriptografada para ver se era em inglês. Isso nos permitiu usar a técnica de força bruta para hackear uma cifra que possui milhares de chaves possíveis.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. O que esta expressão avalia para?
`'Hello world'.strip ()`
2. Quais caracteres são caracteres em branco?
3. Por que `'Hello world'.strip (' o ')` avalia uma string que ainda contém `O s`?
4. Por que `'xxxHelloxxx'.strip (' X ')` avalia uma string que ainda possui `X s`?

13

UM MÓDULO ARITMÉTICO MODULAR PARA A

CIPRA DE AFFINE

“As pessoas têm defendido sua própria privacidade há séculos com sussurros, escuridão, envelopes, portas fechadas, apertos de mão secretos e mensageiros. As tecnologias do passado não permitiam uma privacidade forte, mas as tecnologias eletrônicas sim. ”

—Eric Hughes, “Um Manifesto de Cypherpunk” (1993)



Neste capítulo, você aprenderá sobre a cifra multiplicativa e a cifra afim. A cifra multiplicativa é semelhante à cifra de César, mas criptografa usando multiplicação em vez de adição. A cifra afim combina a cifra multiplicativa e a cifra de César, resultando em uma criptografia mais forte e confiável.

Mas primeiro, você aprenderá sobre a aritmética modular e os maiores divisores comuns - dois conceitos matemáticos necessários para entender e implementar a cifra afim. Usando esses conceitos, criaremos um módulo para lidar com o wrapper e encontrar chaves válidas para a cifra afim. Usaremos este módulo quando criarmos um programa para a cifra afim no [Capítulo 14](#).

TÓPICOS ABORDADOS NESTE CAPÍTULO

- aritmética modular
- O operador de módulo (%)
- O maior divisor comum (GCD)
- Múltipla atribuição
- Algoritmo de Euclides para encontrar o GCD
- As cifras multiplicativas e afins
- Algoritmo estendido de Euclides para encontrar inversos modulares

Aritmética Modular

Aritmética modular, ou *aritmética de relógio*, refere-se a matemática em que os números se envolvem quando atingem um valor específico. Usaremos a

aritmética modular para lidar com a envolvente na cifra afim. Vamos ver como isso funciona.

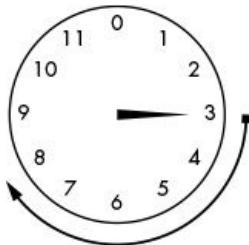


Figura 13-1: 3 horas + 5 horas = 8 horas

Imagine um relógio com apenas um ponteiro de uma hora e o 12 substituído por um 0. (Se os relógios projetados pelos programadores, a primeira hora começaria em 0.) Se a hora atual for 3 horas, que horas serão em 5 horas? Isto é bastante fácil de descobrir: $3 + 5 = 8$. Serão 8 horas em 5 horas. Pense no ponteiro das horas a partir de 3 e depois em 5 horas no sentido horário, como mostra a [Figura 13-1](#).

Se a hora atual for 10 horas, que horas serão em 5 horas? Adicionando $5 + 10 = 15$, mas 15 horas não faz sentido para relógios que mostram apenas 12 horas. Para descobrir que horas serão, você subtrai $15 - 12 = 3$, então serão 3 horas. (Normalmente, você distinguiria entre 3 da manhã e 3 da tarde, mas isso não importa na aritmética modular).

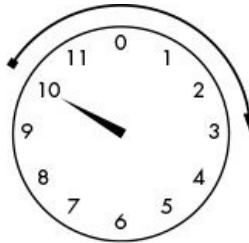


Figura 13-2: 10 horas + 5 horas = 3 horas

Verifique novamente essa matemática movendo o ponteiro das horas no sentido horário por 5 horas, começando em 10. Ele realmente pousa em 3, conforme mostrado na [Figura 13-2](#).

Se a hora atual for 10 horas, que horas serão em 200 horas? Adicionar $200 + 10 = 210$ e 210 é certamente maior que 12. Como uma rotação completa traz o ponteiro das horas de volta à sua posição original, podemos resolver esse problema subtraindo 12 (que é uma rotação completa) até que o resultado seja um número menor que 12. Subtraindo $210 - 12 = 198$. Mas 198 ainda é maior que 12, então continuamos a subtrair 12 até que a diferença seja menor que 12;

neste caso, a resposta final será 6. Se a hora atual for 10 horas, a hora 200 horas depois será 6 horas, conforme mostrado na [Figura 13-3](#).

Se você quiser verificar novamente as 10 horas + 200 horas de matemática, você pode repetidamente mover o ponteiro das horas em torno do mostrador do relógio. Quando você move o ponteiro das horas pela 200^a hora, ele deve pousar em 6.

No entanto, é mais fácil fazer com que o computador faça essa aritmética modular para nós com o operador de módulo.

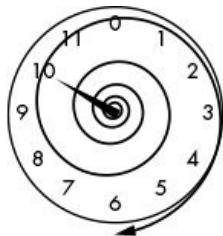


Figura 13-3: 10 horas + 200 horas = 6 horas

O operador Modulo

Você pode usar o *operador de módulo*, abreviado como *mod*, para escrever expressões modulares. No Python, o operador mod é o sinal de porcentagem (%). Você pode pensar no operador mod como um tipo de operador de divisão de divisão; por exemplo, $21 \div 5 = 4$ com um resto de 1 e $21 \% 5 = 1$. Da mesma forma, $15 \% 12$ é igual a 3, assim como 15 horas seriam 3 horas. Digite o seguinte no shell interativo para ver o operador mod em ação:

```
>>> 21% 5  
1  
>>> (10 + 200)% 12  
6  
>>> 10% 10  
0  
>>> 20% 10  
0
```

Assim como 10 horas mais 200 horas irá envolver as 6 horas em um relógio com 12 horas, $(10 + 200)\% 12$ será avaliado como 6. Observe que os números que dividem uniformemente serão modificados para 0, como $10\% 10$ ou $20\% 10$.

Mais tarde, usaremos o operador mod para lidar com a inclusão na cifra afim.

Ele também é usado no algoritmo que usaremos para encontrar o maior divisor comum de dois números, o que nos permitirá encontrar chaves válidas para a cifra afim.

Encontrando Fatores para Calcular o Maior Divisor Comum

Fatores são os números que são multiplicados para produzir um número específico. Considere $4 \times 6 = 24$. Nesta equação, 4 e 6 são fatores de 24. Como os fatores de um número também podem ser usados para dividir esse número sem deixar um resto, os fatores também são chamados de *divisores*.

O número 24 também tem alguns outros fatores:

$$8 \times 3 = 24$$

$$12 \times 2 = 24$$

$$24 \times 1 = 24$$

Portanto, os fatores de 24 são 1, 2, 3, 4, 6, 8, 12 e 24.

Vamos olhar para os fatores de 30:

$$1 \times 30 = 30$$

$$2 \times 15 = 30$$

$$3 \times 10 = 30$$

$$5 \times 6 = 30$$

Os fatores de 30 são 1, 2, 3, 5, 6, 10, 15 e 30. Observe que qualquer número sempre terá 1 e ele próprio como seus fatores, porque 1 vezes um número é igual a esse número. Observe também que a lista de fatores para 24 e 30 tem 1, 2, 3 e 6 em comum. O maior desses fatores comuns é 6, então 6 é o *maior fator comum*, mais comumente conhecido como o *maior divisor comum (GCD)*, de 24 e 30.

É mais fácil encontrar um GCD de dois números visualizando seus fatores.

Vamos visualizar os fatores e o GCD usando *as varas Cuisenaire*. Uma barra Cuisenaire é composta de quadrados iguais ao número que a haste representa, e as hastas nos ajudam a visualizar operações matemáticas. [A Figura 13-4](#) usa varetas Cuisenaire para visualizar $3 + 2 = 5$ e $5 \times 3 = 15$.

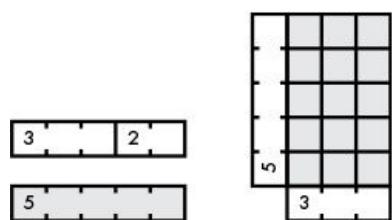


Figura 13-4: Usando varetas Cuisenaire para demonstrar adição e

multiplicação

Uma barra de 3 adicionada a uma barra de 2 tem o mesmo comprimento de uma barra de 5. Você pode até usar barras para encontrar respostas para problemas de multiplicação fazendo um retângulo com lados feitos de varetas dos números que você deseja multiplicar. O número de quadrados no retângulo é a resposta para o problema de multiplicação.

Se uma haste com 20 unidades de comprimento representa o número 20, um número é um fator de 20 se as barras desse número puderem se encaixar dentro da haste de 20 quadrados. [A Figura 13-5](#) mostra que 4 e 10 são fatores de 20 porque se encaixam uniformemente em 20.

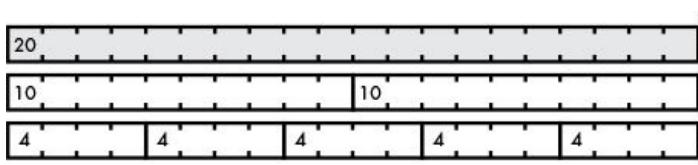


Figura 13-5: As barras de Cuisenaire demonstrando 4 e 10 são fatores de 20

Mas 6 e 7 não são fatores de 20, porque as hastes de 6 e 7 quadrados não caberão uniformemente na haste de 20 quadrados, como mostrado na [Figura 13-6](#).

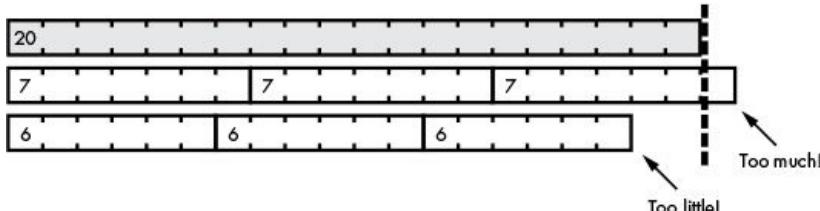


Figura 13-6: Os varões Cuisenaire demonstrando 6 e 7 não são fatores de 20

O GCD de duas hastes, ou dois números representados por essas hastes, é a haste *mais longa* que pode se encaixar de maneira uniforme em *ambas as* hastes, como mostrado na [Figura 13-7](#).

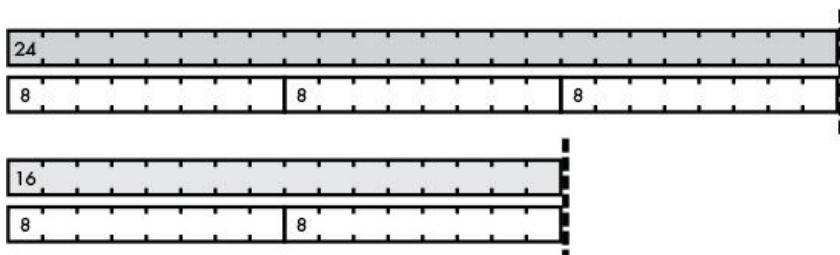


Figura 13-7: Varas de Cuisenaire demonstrando o GCD de 16 e 24

Neste exemplo, a haste de 8-quadrados é a haste mais longa que pode se

encaixar de forma uniforme em 24 e 32. Portanto, 8 é o seu GCD.

Agora que você sabe como os fatores e o GCD funcionam, vamos encontrar o GCD de dois números usando uma função que podemos escrever em Python.

Atribuição Múltipla

A função gcd () que vamos escrever encontra o GCD de dois números. Mas antes de aprender como codificá-lo, vamos ver um truque no Python chamado de *atribuição múltipla*. O truque de atribuição múltipla permite atribuir valores a mais de uma variável de uma vez em uma única instrução de atribuição. Digite o seguinte no shell interativo para ver como isso funciona:

```
>>> spam, ovos = 42, 'Olá'  
>>> spam  
42  
>>> ovos  
'Olá'  
>>> a, b, c, d = ['Alice', 'Brienne', 'Carol', 'Danielle']  
>>> um  
'Alice'  
>>> d  
'Danielle'
```

Você pode separar os nomes das variáveis no lado esquerdo do operador = , bem como os valores no lado direito do operador = usando vírgulas. Você também pode atribuir cada um dos valores em uma lista à sua própria variável, desde que o número de itens na lista seja o mesmo que o número de variáveis no lado esquerdo do operador = . Se você não tiver o mesmo número de variáveis que os valores, o Python exibirá um erro que indica que a chamada precisa de mais ou possui muitos valores.

Um dos principais usos de várias atribuições é trocar os valores em duas variáveis. Digite o seguinte no shell interativo para ver um exemplo:

```
>>> spam = 'olá'  
>>> ovos = 'tchau'  
>>> spam, ovos = ovos, spam  
>>> spam  
'Tchau'  
>>> ovos
```

'Olá'

Depois de atribuir 'olá' ao spam e 'adeus' aos ovos , trocamos esses valores usando várias atribuições. Vamos ver como usar esse truque de troca para implementar o algoritmo de Euclides para encontrar o GCD.

Algoritmo de Euclides para encontrar o GCD

Encontrar o GCD parece bastante simples: identifique todos os fatores dos dois números que você usará e depois encontre o maior fator que eles têm em comum. Mas não é tão fácil encontrar o GCD de números maiores.

Euclides, um matemático que viveu há 2000 anos, surgiu com um algoritmo curto para encontrar o GCD de dois números usando a aritmética modular. Aqui está uma função gcd () que implementa seu algoritmo em código Python, retornando o GCD dos inteiros a e b :

```
def gcd (a, b):  
    enquanto a! = 0:  
        a, b = b% a, a  
    retornar b
```

A função gcd () usa dois números aeb , e usa um loop e uma atribuição múltipla para encontrar o GCD. [A Figura 13-8](#) mostra como a função gcd () encontra o GCD de 24 e 32.

Exatamente como o algoritmo de Euclides funciona está além do escopo deste livro, mas você pode confiar nessa função para retornar o GCD dos dois inteiros que você passa. Se você chamar esta função do shell interativo e passar 24 e 32 para os parâmetros a e b , a função retornará 8 :

```
>>> gcd (24, 32)  
8  
a, b = b % a, a  
  
a, b = 32 % 24, 24 ← Expression calculates b mod a.  
a, b = 8 , 24 ← Loop continues because a != 0.  
a, b = b % a, a ← Multiple assignment statement  
                     swaps the positions of the values.  
a, b = 24 % 8, 8 ← Expression calculates b mod a.  
a, b = 0 , 8 ← Loop ends because a = 0.  
b = 8 ← The final value of b is the GCD.
```

Figura 13-8: Como funciona a função gcd ()

O grande benefício dessa função gcd () , no entanto, é que ela pode lidar facilmente com números grandes:

```
>>> gcd (409119243, 87780243)  
6837
```

Esta função gcd () será útil ao escolher chaves válidas para as cifras multiplicativas e afins, como você aprenderá na próxima seção.

Entendendo como funcionam as cifras multiplicativas e afins

Na cifra de César, criptografar e descriptografar símbolos envolvia convertê-los em números, adicionar ou subtrair a chave e depois converter o novo número de volta em um símbolo.

Ao criptografar com a *cifra multiplicativa* , você *multiplicará* o índice pela chave. Por exemplo, se você criptografou a letra E com a chave 3, você encontraria o índice de E (4) e o multiplicaria pela chave (3) para obter o índice da letra criptografada ($4 \times 3 = 12$), que seria M.

Quando o produto excede o número total de letras, a cifra multiplicativa tem um problema envolvente semelhante à cifra de César, mas agora podemos usar o operador mod para resolver esse problema. Por exemplo, a variável SYMBOLS da cifra de César continha a string

'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456 . A seguir, uma tabela com os primeiros e últimos caracteres de SYMBOLS junto com seus índices:

0	1	2	3	4	5	6	...	59	60	61	62	63	64	65
A	B	C	D	E	F	G	...	8	9	0		!	?	.

Vamos calcular o que esses símbolos criptografam quando a chave é 17. Para criptografar o símbolo F com a chave 17, multiplique seu índice de 5 por 17 e modifique o resultado por 66 para manipular o contorno do conjunto de 66 símbolos. O resultado de $(5 \times 17) \text{ mod } 66$ é 19 e 19 corresponde ao símbolo T. Então, F criptografa para T na cifra multiplicativa com a chave 17. As duas cadeias seguintes mostram todos os caracteres em texto simples e seus símbolos de texto cifrado correspondentes. O símbolo em um determinado índice na primeira string criptografa para o símbolo nesse mesmo índice na segunda string:

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456
'ARizCTk2EVm4GXo6IZq8Kbs0Mdu! Ofw.QhyBSj1DUI3FWn5HYp7Jar9Lct
Nev? Pgx'

Compare essa saída de criptografia com a que você obteria ao criptografar usando a cifra de César, que simplesmente altera os símbolos de texto simples para criar os símbolos de texto cifrado:

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456
'RSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890!?.
ABCDEFGHIJKLMNPQ'

Como você pode ver, a cifra multiplicativa com chave 17 resulta em um texto cifrado que é mais aleatório e mais difícil de decifrar. No entanto, você precisará ser cuidadoso ao escolher as chaves para as cifras multiplicativas. Eu vou discutir o porquê.

Escolhendo Chaves Multiplicativas Válidas

Você não pode simplesmente usar qualquer número para a chave da cifra multiplicativa. Por exemplo, se você escolheu a chave 11, aqui está o mapeamento com o qual você terminaria:

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456
'ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4

Observe que essa chave não funciona porque os símbolos A, G e M criptografam todos na mesma letra, A. Quando você encontra um A no texto cifrado, não sabe qual símbolo ele descriptografa. Usando essa chave, você teria o mesmo problema ao criptografar as letras A, N, F, S e outras.

Na cifra multiplicativa, a chave e o tamanho do conjunto de símbolos devem ser relativamente primos um ao outro. Dois números são *relativamente primos* (ou *coprime*) se o seu GCD for 1. Em outras palavras, eles não têm nenhum fator em comum exceto 1. Por exemplo, os números num1 e num2 são relativamente primos se $\text{mdc}(\text{num1}, \text{num2}) = 1$, onde num1 é a chave e num2 é o tamanho do conjunto de símbolos. No exemplo anterior, porque 11 (a chave) e 66 (o tamanho do conjunto de símbolos) têm um GCD que não é 1, eles não são relativamente primos, o que significa que a chave 11 não pode ser usada para a cifra multiplicativa. Observe que os números não precisam ser números primos para serem relativamente primos um para o outro.

Saber usar a aritmética modular e a função gcd () é importante ao usar a cifra

multiplicativa. Você pode usar a função gcd () para descobrir se um par de números é relativamente primo, o que você precisa saber para escolher chaves válidas para a cifra multiplicativa.

A cifra multiplicativa tem apenas 20 chaves diferentes para um conjunto de 66 símbolos, menos ainda que a cifra de César! No entanto, você pode combinar a cifra multiplicativa e a cifra de César para obter a cifra afim mais poderosa, que eu explico a seguir.

Criptografando com a Cifra Afim

Uma desvantagem de usar a cifra multiplicativa é que a letra A sempre é mapeada para a letra A. A razão é que o número de A é 0 e 0 multiplicado por qualquer coisa sempre será 0. Você pode corrigir esse problema adicionando uma segunda chave para executar uma criptografia de cifra de César após a multiplicação e modulação da cifra multiplicativa. Este passo extra muda a cifra multiplicativa para a *cifra afim*.

A cifra afim tem duas chaves: Chave A e Chave B. A Chave A é o número inteiro que você usa para multiplicar o número da letra. Depois de multiplicar o texto sem formatação pela chave A, você adiciona a chave B ao produto. Então modifique a soma por 66, como fez na cifra original de César. Isso significa que a cifra afim tem 66 vezes mais chaves possíveis que a cifra multiplicativa. Também garante que a letra A nem sempre criptografe para si mesma.

O processo de descriptografia da cifra afim espelha o processo de criptografia; ambos são mostrados na [Figura 13-9](#).

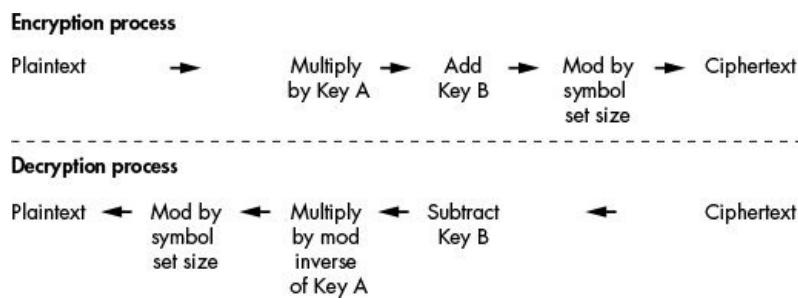


Figura 13-9: Os processos de criptografia e descriptografia da criptografia afim

Descriptografamos a cifra afim usando as operações opostas usadas para criptografia. Vamos ver o processo de descriptografia e como calcular o inverso modular em mais detalhes.

Decifrando com a Cifra Afim

Na cifra de César, você usou além de criptografar e subtrair para descriptografar. Na cifra afim, você usa multiplicação para criptografar. Naturalmente, você pode pensar que pode dividir para descriptografar com a cifra afim. Mas se você tente isso, você verá que não funciona. Para decifrar com a cifra afim, você precisa multiplicar pelo inverso modular da chave. Isso reverte a operação de modificação do processo de criptografia.

Um *inverso modular* de dois números é representado pela expressão $(a * i) \% m == 1$, onde i é o inverso modular e a e m são os dois números. Por exemplo, o inverso modular de $5 \text{ mod } 7$ seria algum número i onde $(5 * i)\% 7$ é igual a 1. Você pode fazer a força bruta deste cálculo da seguinte forma:

1 não é o inverso modular de $5 \text{ mod } 7$, porque $(5 * 1)\% 7 = 5$.

2 não é o inverso modular de $5 \text{ mod } 7$, porque $(5 * 2)\% 7 = 3$.

3 é o inverso modular de $5 \text{ mod } 7$, porque $(5 * 3)\% 7 = 1$.

Embora as chaves de criptografia e descriptografia da parte de cifra de César da cifra afim sejam as mesmas, as chaves de criptografia e descriptografia da cifra multiplicativa são dois números diferentes. A chave de criptografia pode ser qualquer coisa que você escolher, desde que seja relativamente primo para o tamanho do conjunto de símbolos, que neste caso é 66. Se você escolher a chave 53 para criptografar com a cifra afim, a chave de descriptografia é o inverso modular $53 \text{ mod } 66$:

1 não é o inverso modular de $53 \text{ mod } 66$, porque $(53 * 1)\% 66 = 53$.

2 não é o inverso modular de $53 \text{ mod } 66$, porque $(53 * 2)\% 66 = 40$.

3 não é o inverso modular de $53 \text{ mod } 66$, porque $(53 * 3)\% 66 = 27$.

4 não é o inverso modular de $53 \text{ mod } 66$, porque $(53 * 4)\% 66 = 14$.

5 é o inverso modular de $53 \text{ mod } 66$, porque $(53 * 5)\% 66 = 1$.

Como 5 é o inverso modular de 53 e 66, você sabe que a chave de decodificação de cifra afim também é 5. Para descriptografar uma letra de texto cifrado, multiplique o número dessa letra por 5 e, em seguida, mod 66. O resultado é o número da letra original do texto original .

Usando o conjunto de símbolos de 66 caracteres, vamos criptografar a palavra *Cat* usando a tecla 53 . C está no índice 2 e $2 * 53$ é 106 , que é maior que o tamanho do conjunto de símbolos, portanto modamos 106 por 66 e o resultado é

40 . O caractere no índice 40 no conjunto de símbolos é 'o' , então o símbolo C criptografa para o .

Vamos usar os mesmos passos para a próxima carta, a . A string 'a' está no índice 26 no conjunto de símbolos e $26 * 53\% 66$ é 58 , que é o índice de '7' . Então o símbolo criptografa para 7 . A string 't' está no índice 45 e $45 * 53\% 66$ é 9 , que é o índice de 'J' . Portanto, a palavra *Cat* é criptografada para *o7J* .

Para descriptografar, multiplicamos pelo inverso modular de 53 % 66 , que é 5 . O símbolo o está no índice 40 e $40 * 5\% 66$ é 2 , que é o índice de 'C' . O símbolo 7 está no índice 58 e $58 * 5\% 66$ é 26 , que é o índice de 'a' . O símbolo J está no índice 9 e $9 * 5\% 66$ é 45 , que é o índice de 't' . O texto cifrado descriptografa *Cat* , que é o texto original, exatamente como esperado.

Encontrando Inversas Modulares

Para calcular o inverso modular para determinar a chave de decodificação, você poderia adotar uma abordagem de força bruta e começar a testar o inteiro 1 e, em seguida, 2 e, em seguida, 3, e assim por diante. Mas isso é demorado para chaves grandes, como 8.953.851.

Felizmente, você pode usar o algoritmo estendido de Euclides para encontrar o inverso modular de um número, que em Python se parece com isso:

```
def findModInverse (a, m):
se mdc (a, m)! = 1:
return Nenhum # Nenhum mod inverso se a & m não são relativamente primos.
u1, u2, u3 = 1, 0, um
v1, v2, v3 = 0, 1, m
enquanto v3! = 0:
q = u3 // v3 # Observe que // é o operador de divisão inteira.
v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3),
v1, v2, v3
retorno u1% m
```

Você não precisa entender como o algoritmo estendido de Euclid funciona para usar a função `findModInverse ()` . Contanto que os dois argumentos que você passa para a função `findModInverse ()` sejam relativamente primos, `findModInverse ()` retornará o inverso modular do parâmetro a.

Você pode aprender mais sobre como o algoritmo estendido de Euclides funciona em <https://www.nostarch.com/crackingcodes/> .

Operador da Divisão Inteira

Você deve ter notado o // operador usado na função findModInverse () na seção anterior. Este é o *operador de divisão inteira*. Ele divide dois números e arredonda para o inteiro mais próximo. Digite o seguinte no shell interativo para ver como o // operador funciona:

```
>>> 41/7  
5.857142857142857  
>>> 41 // 7  
5  
>>> 10 // 5  
2
```

Considerando que 41/7 avalia a 5.857142857142857 , usando 41 // 7 avalia a 5 . Para expressões de divisão que não são divididas uniformemente, o // operador é útil para obter o número inteiro da resposta (às vezes chamado de *quociente*), enquanto o operador % obtém o restante. Uma expressão que usa o // operador de divisão inteira sempre avalia um int, não um float. Como você pode ver ao avaliar 10 // 5 , o resultado é 2 em vez de 2.0 .

Código-fonte para o módulo de criptografia

Usaremos gcd () e findModInverse () em mais programas de codificação posteriormente neste livro, então vamos colocar ambas as funções em um módulo. Abra uma nova janela do editor de arquivos, insira o código a seguir e salve o arquivo como *cryptomath.py* :

cryptomath.py

1. Módulo Cryptomath
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. def gcd (a, b):
5. Retorne o GCD de aeb usando o algoritmo de Euclides:
6. enquanto a! = 0:
7. a, b = b% a, um
8. retornar b
- 9
- 10
11. def findModInverse (a, m):

```
12. # Retorna o inverso modular de um% m, que é
13. # o número x tal que a * x% m = 1.
14
15. se mdc (a, m)! = 1:
16. return None # Não mod inverso se a & m não for relativamente primo.
17
18. # Calcular usando o algoritmo euclidiano estendido:
19. u1, u2, u3 = 1, 0, um
20. v1, v2, v3 = 0, 1, m
21. enquanto v3! = 0:
22.     q = u3 // v3 # Observe que // é o operador de divisão inteira.
23.     v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2),
(u3 - q * v3), v1, v2, v3
24. retorno u1% m
```

Este programa contém a função gcd () descrita anteriormente neste capítulo e a função findModInverse () que implementa o algoritmo estendido de Euclides.

Depois de importar o módulo *cryptomath.py* , você pode experimentar essas funções a partir do shell interativo. Digite o seguinte no shell interativo:

```
>>> import cryptomath
>>> cryptomath.gcd (24, 32)
8
>>> cryptomath.gcd (37, 41)
1
>>> cryptomath.findModInverse (7, 26)
15
>>> cryptomath.findModInverse (8953851, 26)
17
```

Como você pode ver, você pode chamar a função gcd () e a função findModInverse () para encontrar o GCD ou o inverso modular de dois números.

Resumo

Este capítulo abordou alguns conceitos matemáticos úteis. O operador % localiza o restante depois de dividir um número por outro. A função gcd () retorna o maior número que pode dividir dois números uniformemente. Se o GCD de dois números for 1, você sabe que esses números são relativamente primos um para o outro. O algoritmo mais útil para encontrar o GCD de dois

números é o algoritmo de Euclides.

Ao contrário da cifra de César, a cifra afim usa multiplicação e adição em vez de apenas adição para criptografar letras. No entanto, nem todos os números funcionam como chaves para a cifra afim. O número da chave e o tamanho do conjunto de símbolos devem ser relativamente primos um para o outro.

Para decifrar com a cifra afim, multiplique o índice do texto cifrado pelo inverso modular da chave. O inverso modular de $um \% m$ é um número i tal que $(a * i) \% m == 1$. Você pode usar o algoritmo estendido de Euclides para calcular inversos modulares. A codificação de chave pública do [Capítulo 23](#) também usa inversões modulares.

Usando os conceitos matemáticos que você aprendeu neste capítulo, você irá escrever um programa para a cifra afim no [Capítulo 14](#). Como a cifra multiplicativa é a mesma coisa que a cifra afim usando uma chave B de 0, você não terá um programa de cifra multiplicativo separado. E como a cifra multiplicativa é apenas uma versão menos segura da cifra afim, você não deve usá-la de qualquer maneira.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. O que as expressões a seguir avaliam?

17% 1000

5% 5

2. Qual é o GCD de 10 e 15?
3. O que o spam contém depois de executar spam, ovos = 'olá', 'mundo' ?
4. O GCD de 17 e 31 é 1. São 17 e 31 relativamente primos?
5. Por que não são 6 e 8 relativamente primos?
6. Qual é a fórmula para o inverso modular de $A \bmod C$?

14

PROGRAMANDO A CIPE DE AFFINE

"Eu deveria ser capaz de sussurrar algo em seu ouvido, mesmo que seu ouvido esteja a mil milhas de distância, e o governo discordar disso."

—Philip Zimmermann, criador do Pretty Good Privacy (PGP), o software de criptografia de e-mail mais usado no mundo



No [Capítulo 13](#), você aprendeu que a cifra afim é na verdade a cifra multiplicativa combinada com a cifra de César ([Capítulo 5](#)), e a cifra multiplicativa é semelhante à cifra de César, exceto que usa multiplicação em vez de adição para criptografar mensagens. Neste capítulo, você construirá e executará programas para implementar a cifra afim. Como a cifra afim usa duas cifras diferentes como parte de seu processo de criptografia, ela precisa de duas chaves: uma para a cifra multiplicativa e outra para a cifra de César. Para o programa de criptografia afim, dividiremos um único inteiro em duas chaves.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- O tipo de dados da tupla
- Quantas chaves diferentes podem ter a cifra afim?
- Gerando chaves aleatórias

Código-fonte para o programa de códigos afim

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como *affineCipher.py*. Certifique-se de que o módulo *pyperclip.py* e o módulo *cryptomath.py* que você criou no [Capítulo 13](#) estejam na mesma pasta que o arquivo *affineCipher.py*.

affineCipher.py

```
1. # Affine Cipher
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar sys, pyperclip, cryptomath, random
5. SÍMBOLOS =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
67890!
6
```

7

8. def main ():

9. myMessage = """ """ Um computador mereceria ser chamado de inteligente
se pudesse enganar um humano acreditando que era humano """.

-Alan Turing """ "

10. myKey = 2894

11. myMode = 'encrypt' # Defina como 'encriptar' ou 'decifrar'.

12

13. se myMode == 'encriptar':

14. translated = encryptMessage (myKey, myMessage)

15. elif myMode == 'decifrar':

16. translated = decryptMessage (myKey, myMessage)

17. print ('Chave:% s'% (myKey))

18. print ('% sed text:'% (myMode.title ()))

19. print (tradução)

20. pyperclip.copy (tradução)

21. print ('Full% sed text copiado para a área de transferência.'% (MyMode))

22

23

24. def getKeyParts (chave):

25. keyA = chave // len (SÍMBOLOS)

26. keyB = chave% len (SÍMBOLOS)

27. retorno (keyA, keyB)

28.

29

30. def checkKeys (keyA, keyB, mode):

31. se keyA == 1 e mode == 'encrypt':

32. sys.exit ('A cifra é fraca se a chave A é 1. Escolha uma chave diferente.')

33. se keyB == 0 e mode == 'encrypt':

34. sys.exit ('A cifra é fraca se a chave B é 0. Escolha uma chave diferente.')

35. se keyA <0 ou keyB <0 ou keyB> len (SYMBOLS) - 1:

36. sys.exit ('A chave deve ser maior que 0 e a tecla B deve ser
entre 0 e% s. ' % (len (SÍMBOLOS) - 1))

37. if cryptomath.gcd (keyA, len (SÍMBOLOS))! = 1:

38. sys.exit ('Key A (% s) e o tamanho do conjunto de símbolos (% s) não são
relativamente primo. Escolha uma chave diferente. % (keyA,
len (SÍMBOLOS)))

39
40.
41. def encryptMessage (key, message):
42. keyA, keyB = getKeyParts (chave)
43. checkKeys (keyA, keyB, 'encriptar')
44. texto cifrado = "
45. para o símbolo na mensagem:
46. se símbolo em SÍMBOLOS:
47. # Criptografar o símbolo:
48. symbolIndex = SYMBOLS.find (símbolo)
49. texto cifrado + = SÍMBOLOS [(symbolIndex * keyA + keyB)%
len (SÍMBOLOS)]
50. else:
51. texto cifrado + = símbolo # Anexa o símbolo sem criptografar.
52. retornar texto cifrado
53
54
55. def decryptMessage (chave, mensagem):
56. keyA, keyB = getKeyParts (chave)
57. checkKeys (keyA, keyB, 'decifrar')
58. texto simples = "
59. modInverseOfKeyA = cryptomath.findModInverse (keyA, len
(SÍMBOLOS))
60
61. para símbolo na mensagem:
62. se símbolo em SÍMBOLOS:
63. # Descriptografar o símbolo:
64. symbolIndex = SYMBOLS.find (símbolo)
65. texto sem formatação + = SYMBOLS [(symbolIndex - keyB) *
modInverseOfKeyA%
len (SÍMBOLOS)]
66. else:
67. texto sem formatação + = símbolo # Anexe o símbolo sem descriptografar.
68. retornar texto simples
69
70
71. def getRandomKey ():

```
72. enquanto verdadeiro:  
73.     keyA = random.randint (2, len (SÍMBOLOS))  
74.     keyB = random.randint (2, len (SÍMBOLOS))  
75.     if cryptomath.gcd (keyA, len (SYMBOLS)) == 1:  
76.         retornar keyA * len (SÍMBOLOS) + keyB  
77  
78  
79. # Se affineCipher.py for executado (em vez de importado como um módulo),  
chame  
80. # a função main ():  
81. if __name__ == '__main__':  
82.     main ()
```

Exemplo de Execução do Programa de Cifra Afim

A partir do editor de arquivos, pressione F5 para executar o programa *affineCipher.py* ; a saída deve ficar assim:

Chave: 2894

Texto criptografado:

```
"5QG9oI3La6QI93! XQxaia6faQL9QdaQG1 !! axQARLa !!  
AuaRLQADQALQG93! XQxaGaAfaQ1QX3o1R  
QARL9Qda! AafARuQLX1LQALQI1iQX3o1RN "Q-5! 1RQP36ARuFull texto  
criptografado copiado para  
prancheta.
```

No programa de criptografia afim, a mensagem "Um computador mereceria ser chamado de inteligente, se pudesse enganar um humano, acreditando que ele era humano". -Alan Turing , é criptografado com a chave 2894 em texto cifrado. Para descriptografar esse texto cifrado, você pode copiá-lo e colá-lo como o novo valor a ser armazenado em myMessage na linha 9 e alterar myMode na linha 13 para a string ' decrypt' .

Configurando módulos, constantes e a função main ()

As linhas 1 e 2 do programa são comentários descrevendo o que é o programa. Há também uma declaração de importação para os módulos usados neste programa:

1. # Affine Cipher
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)

3

4. importar sys, pyperclip, cryptomath, random

Os quatro módulos importados neste programa atendem às seguintes funções:

- O módulo sys é importado para a função exit () .
- O módulo pyperclip é importado para a função de área de transferência copy () .
- O módulo cryptomath que você criou no [Capítulo 13](#) é importado para as funções gcd () e findModInverse () .
- O módulo aleatório é importado para a função random.randint () para gerar chaves aleatórias.

A string armazenada na variável SYMBOLS é o conjunto de símbolos, que é a lista de todos os caracteres que podem ser criptografados:

5. SÍMBOLOS =

```
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345  
67890!'
```

Todos os caracteres da mensagem que não aparecem em SYMBOLS permanecem descriptografados no texto cifrado. Por exemplo, na execução de amostra de *affineCipher.py* , as aspas e o hífen (-) não são criptografados no texto cifrado porque não pertencem ao conjunto de símbolos.

A linha 8 chama a função main () , que é quase exatamente a mesma dos programas de criptografia de transposição. As linhas 9, 10 e 11 armazenam a mensagem, chave e modo em variáveis, respectivamente:

8. def main ():

```
9. myMessage = """ Um computador mereceria ser chamado de inteligente  
se pudesse enganar um humano acreditando que era humano """.
```

-Alan Turing """

10. myKey = 2894

```
11. myMode = 'encrypt' # Defina como 'encriptar' ou 'decifrar'.
```

O valor armazenado no myMode determina se o programa criptografa ou descriptografa a mensagem:

13. se myMode == 'encriptar':

```
14. translated = encryptMessage (myKey, myMessage)
```

```
15. elif myMode == 'decifrar':  
16.     translated = decryptMessage (myKey, myMessage)
```

Se myMode for definido como 'encrypt' , a linha 14 será executada e o valor de retorno de encryptMessage () será armazenado na tradução . Mas se myMode é definido como 'decifrar' , decryptMessage () é chamado na linha 16 e o valor de retorno é armazenado na tradução . Eu cobrirei como as funções encryptMessage () e decryptMessage () funcionam quando as definimos mais tarde no capítulo.

Depois que a execução passar a linha 16, a variável traduzida terá a versão criptografada ou descriptografada da mensagem em myMessage .

A linha 17 exibe a chave usada para a cifra usando o marcador de posição % s , e a linha 18 informa ao usuário se a saída é criptografada ou descriptografada:

```
17. print ('Chave:% s'% (myKey))  
18. print ('% sed text:'% (myMode.title ()))  
19. print (tradução)  
20. pyperclip.copy (tradução)  
21. print ('Full% sed text copiado para a área de transferência.'% (MyMode))
```

A linha 19 imprime a string em traduzida , que é a versão criptografada ou descriptografada da string em myMessage , e a linha 20 a copia para a área de transferência. A linha 21 notifica o usuário que está na área de transferência.

Calculando e validando as chaves

Ao contrário da cifra de César, que usa adição com apenas uma chave, a cifra afim usa multiplicação e adição com duas chaves inteiras, que chamaremos de Chave A e Chave B. Como é mais fácil lembrar apenas um número, usaremos uma truque matemático para converter duas chaves em uma chave. Vamos ver como isso funciona em *affineCipher.py* .

A função getKeyParts () na linha 24 divide uma única chave inteira em dois inteiros para a Chave A e a Chave B:

```
24. def getKeyParts (chave):  
25.     keyA = chave // len (SÍMBOLOS)  
26.     keyB = chave% len (SÍMBOLOS)  
27.     retorno (keyA, keyB)
```

A chave para dividir é passada para o parâmetro chave . Na linha 25, a chave A é calculada usando divisão inteira para dividir a chave por len (SYMBOLS) , o

tamanho do conjunto de símbolos. A divisão inteira (//) retorna o quociente sem resto. O operador mod (%) na linha 26 calcula o restante, que usaremos para a chave B.

Por exemplo, com 2894 como o parâmetro- chave e uma sequência SYMBOLS de 66 caracteres, a Chave A seria $2894 \text{ // } 66 = 43$ e a Chave B seria $2894 \% 66 = 56$.

Para combinar a Chave A e a Chave B de volta em uma única tecla, multiplique a Chave A pelo tamanho do conjunto de símbolos e adicione a Chave B ao produto: $(43 * 66) + 56$ é avaliado como 2894 , que é a chave de número inteiro com a qual começamos .

NOTA

Tenha em mente que, de acordo com Maxim de Shannon ("O inimigo conhece o sistema!"), Devemos assumir que os hackers sabem tudo sobre o algoritmo de criptografia, incluindo o conjunto de símbolos e o tamanho do conjunto de símbolos. Assumimos que a única peça que um hacker não conhece é a chave que foi usada. A segurança do nosso programa de criptografia deve depender apenas do sigilo da chave, não do sigilo do conjunto de símbolos ou do código-fonte do programa.

O tipo de dados Tuple

A linha 27 parece retornar um valor de lista, com exceção de parênteses usados em vez de colchetes. Este é um valor de tupla .

27. retorno (keyA, keyB)

Um valor de tupla é semelhante a um valor de lista, pois pode armazenar outros valores, que podem ser acessados com índices ou fatias. No entanto, ao contrário dos valores de lista, os valores da tupla não podem ser modificados. Não há método append () para valores de tupla.

Como *affineCipher.py* não precisa modificar o valor retornado por getKeyParts () , o uso de uma tupla é mais apropriado que uma lista.

Checando Chaves Fracas

Criptografar com a cifra afim envolve o índice de um caractere em SÍMBOLOS sendo multiplicado pela Chave A e adicionado à Chave B. Mas se a chave A for 1 , o texto criptografado é muito fraco porque multiplicar o índice por 1 resulta no mesmo índice. De fato, conforme definido pela propriedade de identidade

multiplicativa, o produto de qualquer número e 1 é esse número. Da mesma forma, se keyB for 0 , o texto criptografado será fraco, pois a adição de 0 ao índice não o altera. Se keyA for 1 e keyB for 0 ao mesmo tempo, a saída "criptografada" será idêntica à mensagem original. Em outras palavras, não seria criptografado!

Verificamos chaves fracas usando a função checkKeys () na linha 30. As instruções if nas linhas 31 e 33 verificam se keyA é 1 ou keyB é 0 .

```
30. def checkKeys (keyA, keyB, mode):  
31.     se keyA == 1 e mode == 'encrypt':  
32.         sys.exit ('A cifra é fraca se a chave A é 1. Escolha uma chave diferente.')  
33.     se keyB == 0 e mode == 'encrypt':  
34.         sys.exit ('A cifra é fraca se a chave B é 0. Escolha uma chave diferente.')
```

Se essas condições forem atendidas, o programa sairá com uma mensagem indicando o que deu errado. Cada uma das linhas 32 e 34 passa uma string para a chamada sys.exit () . A função sys.exit () tem um parâmetro opcional que permite imprimir uma string na tela antes de terminar o programa. Você pode usar essa função para exibir uma mensagem de erro na tela antes que o programa seja encerrado.

Essas verificações impedem que você criptografe com chaves fracas, mas se o seu modo estiver configurado para 'descriptografar' , as verificações nas linhas 31 e 33 não se aplicam.

A condição na linha 35 verifica se keyA é um número negativo (isto é, se é menor que 0) ou se keyB é maior que 0 ou menor que o tamanho do conjunto de símbolos menos um:

```
35.     se keyA <0 ou keyB <0 ou keyB> len (SYMBOLS) - 1:  
36.         sys.exit ('A chave deve ser maior que 0 e a tecla B deve ser  
entre 0 e% s. ' % (len (SÍMBOLOS) - 1))
```

A razão pela qual as chaves estão nesses intervalos é descrita na próxima seção. Se alguma dessas condições for True , as chaves serão inválidas e o programa será encerrado.

Além disso, a chave A deve ser relativamente primo para o tamanho do conjunto de símbolos. Isso significa que o maior divisor comum (GCD) de keyA e len (SYMBOLS) deve ser igual a 1 . A linha 37 verifica isso usando uma instrução if e a linha 38 sai do programa se os dois valores não forem relativamente primos:

37. if cryptomath.gcd (keyA, len (SÍMBOLOS))! = 1:

38. sys.exit ('Key A (% s) e o tamanho do conjunto de símbolos (% s) não são relativamente primo. Escolha uma chave diferente. % (keyA, len (SÍMBOLOS)))

Se todas as condições na função checkKeys () retornarem False , nada está errado com a chave e o programa não sai. Execução do programa retorna para a linha que originalmente chamado checkKeys () .

Quantas chaves a cifra afim pode ter?

Vamos tentar calcular o número de chaves possíveis que a cifra afim tem. A chave B da afinidade é limitada ao tamanho do conjunto de símbolos, onde len (SÍMBOLOS) é 66 . À primeira vista, parece que a Chave A pode ser tão grande quanto você quiser, desde que seja relativamente primo ao tamanho do conjunto de símbolos. Portanto, você pode pensar que a cifra afim tem um número infinito de chaves e não pode ser forçada de forma bruta.

Mas este não é o caso. Lembre-se de como as teclas grandes na cifra de César acabaram sendo as mesmas que as teclas menores devido ao efeito envolvente. Com um tamanho de conjunto de símbolos de 66, a chave 67 na cifra de César produziria o mesmo texto criptografado que a chave 1 . A cifra afim também se envolve desta maneira.

Como a parte da chave B da cifra afim é a mesma da cifra de César, seu alcance é limitado de 1 ao tamanho do conjunto de símbolos. Para determinar se a chave A da cifra afim também é limitada, escreveremos um pequeno programa para criptografar uma mensagem usando vários inteiros diferentes para a Chave A e ver como é o texto cifrado.

Abra uma nova janela do editor de arquivos e insira o código-fonte a seguir. Salve este arquivo como *affineKeyTest.py* na mesma pasta que *affineCipher.py* e *cryptomath.py* . Em seguida, pressione F5 para executá-lo.

affineKeyTest.py

1. # Este programa prova que o espaço de chaves da cifra afim é limitado
2. # para menos que len (SYMBOLS) ^ 2.
- 3
4. import affineCipher, cryptomath
- 5
6. message = 'Torne as coisas o mais simples possível, mas não mais simples.'

```
7. para keyA no intervalo (2, 80):  
8. key = keyA * len (affineCipher.SYMBOLS) + 1  
9  
10. if cryptomath.gcd (keyA, len (affineCipher.SYMBOLS)) == 1:  
11. print (chaveA, affineCipher.encryptMessage (chave, mensagem))
```

Este programa importa o módulo affineCipher para sua função encryptMessage () e o módulo cryptomath para sua função gcd () . Nós sempre criptografamos a string armazenada na variável de mensagem . O loop for permanece em um intervalo entre 2 e 80 , porque 0 e 1 não são permitidos como inteiros válidos da Chave A, conforme explicado anteriormente.

Em cada iteração do loop, a linha 8 calcula a chave do valor da keyA atual e sempre usa 1 para a Key B, e é por isso que 1 é adicionado no final da linha 8. Lembre-se de que a Key A deve ser relativamente primo com a chave. tamanho do conjunto de símbolos para ser válido. A chave A é relativamente primo com o tamanho do conjunto de símbolos se o GCD da chave e o tamanho do conjunto de símbolos for igual a 1 . Portanto, se o GCD da chave e o tamanho do conjunto de símbolos não forem iguais a 1 , a instrução if na linha 10 ignorará a chamada para encryptMessage () na linha 11.

Em suma, este programa imprime a mesma mensagem criptografada com vários inteiros diferentes para a chave A. A saída deste programa é semelhante a:

```
5 0.xTvcnÍdXv.XvXn8I3Tv.XvIDXnE3T, vEhcv? DcvXn8I3TS  
7 Tz4Nn1ipKbtntntpDY NnztnYRttp7 N, n781nKR1ntpDY Nm9  
13 ZJH0P7ivuVtPJtPtvhGU0PJtPG8ttvWU0, PWF7Pu87PtvhGU0g3  
17 HvTx.oizERX.vX.Xz2mkx.vX.mVXXz? Kx,?? 6o.EVo.Xz2mkxGy  
- recorte -  
67 Nblf! Uijoht! Bt! Tjnqmf! Bt! Qpttjcmf,! Cvu! Opu! TjnqmfsA  
71 0.xTvcina? DXv.XvXn8I3Tv.XvIDXnE3T, vEhcv? DcvXn8I3TS  
73 Tz4Nn1ipKbtntntpDY NnztnYRttp7 N, n781nKR1ntpDY Nm9  
79 ZJH0P7ivuVtPJtPtvhGU0PJtPG8ttvWU0, PWF7Pu87PtvhGU0g3
```

Observe atentamente a saída e você notará que o texto cifrado da Chave A de 5 é o mesmo que o texto cifrado da Chave A de 71 ! De fato, o texto cifrado das teclas 7 e 73 é o mesmo, assim como o texto cifrado das teclas 13 e 79 !

Observe também que subtrair 5 de 71 resulta em 66, o tamanho do nosso conjunto de símbolos. É por isso que uma Chave A de 71 faz a mesma coisa que uma Chave A de 5 : a saída criptografada se repete, ou envolve, a cada 66 teclas.

Como você pode ver, a cifra afim tem o mesmo efeito envolvente para a Chave A e para a Chave B. Em suma, a Chave A também é limitada ao tamanho do conjunto de símbolos.

Quando você multiplica 66 chaves possíveis da tecla A por 66 chaves possíveis da tecla B, o resultado é 4356 combinações possíveis. Então, quando você subtrair os inteiros que não podem ser usados para a Chave A, porque eles não são relativamente primos com 66, o número total de combinações de teclas possíveis para a cifra afim cai para 1320.

Escrevendo a função de criptografia

Para criptografar a mensagem em *affineCipher.py*, primeiro precisamos da chave e da mensagem para criptografar, que a função `encryptMessage()` usa como parâmetros:

41. `def encryptMessage(key, message):`
42. `keyA, keyB = getKeyParts(chave)`
43. `checkKeys(keyA, keyB, 'encriptar')`

Então, precisamos obter os valores inteiros para a Chave A e Chave B da função `getKeyParts()` passando a chave para a linha 42. Em seguida, verificamos se esses valores são chaves válidas, passando-os para a função `checkKeys()`. Se a função `checkKeys()` não fizer com que o programa seja encerrado, as chaves serão válidas e o restante do código na função `encryptMessage()` após a linha 43 poderá continuar.

Na linha 44, a variável de texto cifrado começa como uma string em branco, mas eventualmente conterá a string criptografada. O loop `for` que começa na linha 45 percorre cada um dos caracteres da mensagem e, em seguida, adiciona o caractere criptografado ao texto cifrado :

44. `texto_cifrado = "`
45. `para o símbolo na mensagem:`

No momento em que o loop `for` concluído, a variável de texto cifrado conterá a sequência completa da mensagem criptografada.

Em cada iteração do loop, a variável de símbolo é atribuída a um único caractere da mensagem . Se este caracter existe em `SYMBOLS` , que é o nosso conjunto de símbolos, o índice em `SYMBOLS` é encontrado e atribuído a `symbolIndex` na linha 48:

46. se símbolo em SÍMBOLOS:
47. # Criptografar o símbolo:
48. symbolIndex = SYMBOLS.find (símbolo)
49. texto cifrado + = SÍMBOLOS [(symbolIndex * keyA + keyB)%
len (SÍMBOLOS)]
50. else:
51. texto cifrado + = símbolo # Anexa o símbolo sem criptografar.

Para criptografar o texto, precisamos calcular o índice da letra criptografada. A linha 49 multiplica este symbolIndex por keyA e adiciona o keyB ao produto. Então modifica o resultado pelo tamanho do conjunto de símbolos, representado pela expressão len (SYMBOLS) . Modding por len (SYMBOLS) lida com o wraparound garantindo que o índice calculado esteja sempre entre 0 e até, mas não incluindo, len (SYMBOLS) . O número resultante será o índice em SÍMBOLOS do caractere criptografado, que é concatenado ao final da string em texto cifrado .

Tudo no parágrafo anterior é feito na linha 49, usando uma única linha de código!

Se o símbolo não estiver em nosso conjunto de símbolos, o símbolo será concatenado ao final da cadeia de texto cifrado na linha 51. Por exemplo, as aspas e o hífen na mensagem original não estão no conjunto de símbolos e, portanto, são concatenados na sequência.

Depois que o código tiver iterado por meio de cada caractere na sequência de mensagens, a variável de texto cifrado deve conter a cadeia criptografada completa. A linha 52 retorna a string criptografada de encryptMessage () :

52. retornar texto cifrado

Escrevendo a função de descriptografia

A função decryptMessage () que descriptografa o texto é quase a mesma que encryptMessage () . As linhas 56 a 58 são equivalentes às linhas 42 a 44.

55. def decryptMessage (chave, mensagem):
56. keyA, keyB = getKeyParts (chave)
57. checkKeys (keyA, keyB, 'decifrar')
58. texto simples = "
59. modInverseOfKeyA = cryptomath.findModInverse (keyA, len
(SÍMBOLOS))

No entanto, em vez de multiplicar pela chave A, o processo de descriptografia multiplica pelo inverso modular da chave A. O mod inverso é calculado chamando cryptomath.findModInverse () , conforme explicado no [capítulo 13](#) .

As linhas 61 a 68 são quase idênticas às linhas 45 a 52 da função encryptMessage () . A única diferença está na linha 65.

61. para símbolo na mensagem:
62. se símbolo em SÍMBOLOS:
63. # Descriptografar o símbolo:
64. symbolIndex = SYMBOLS.find (símbolo)
65. texto sem formatação += SYMBOLS [(symbolIndex - keyB) *
modInverseOfKeyA%
len (SÍMBOLOS)]
66. else:
67. texto sem formatação += símbolo # Anexe o símbolo sem descriptografar.
68. retornar texto simples

Na função encryptMessage () , o índice de símbolo foi multiplicado pela Chave A e, em seguida, a Chave B foi adicionada a ela. Na função decryptMessage () linha 65, o índice de símbolo primeiro subtrai a chave B do índice de símbolo e multiplica-a pelo inverso modular. Então modifica este número pelo tamanho do conjunto de símbolos, len (SYMBOLS) .

É assim que o processo de descriptografia em *affineCipher.py* desfaz a criptografia. Agora vamos ver como podemos alterar *affineCipher.py* para que ele selecione aleatoriamente chaves válidas para a cifra afim.

Gerando Chaves Aleatórias

Pode ser difícil criar uma chave válida para a cifra afim, portanto, você pode usar a função getRandomKey () para gerar uma chave aleatória, mas válida. Para fazer isso, basta alterar a linha 10 para armazenar o valor de retorno de getRandomKey () na variável myKey :

10. myKey = getRandomKey ()
- recorte -
17. print ('Chave:% s' % (myKey))

Agora, o programa seleciona aleatoriamente a chave e a imprime na tela quando a linha 17 é executada. Vamos ver como funciona a função getRandomKey () .

O código na linha 72 entra em um loop while, onde a condição é True . Esse *loop infinito* será loop para sempre até que seja dito para retornar ou o usuário finaliza o programa. Se o seu programa ficar preso em um loop infinito, você pode finalizar o programa pressionando ctrl -C (ctrl -D no Linux ou macOS). A função getRandomKey () acabará por sair do loop infinito com uma instrução de retorno .

```
71. def getRandomKey ():  
72.     enquanto verdadeiro:  
73.         keyA = random.randint (2, len (SÍMBOLOS))  
74.         keyB = random.randint (2, len (SÍMBOLOS))
```

As linhas 73 e 74 determinam números aleatórios entre 2 e o tamanho do conjunto de símbolos para keyA e keyB . Este código garante que não há chance de que a Chave A ou Chave B seja igual aos valores inválidos 0 ou 1 .

A instrução if na linha 75 verifica se a keyA é relativamente primo com o tamanho do conjunto de símbolos chamando a função gcd () no módulo cryptomath .

```
75. if cryptomath.gcd (keyA, len (SYMBOLS)) == 1:  
76.     retornar keyA * len (SÍMBOLOS) + keyB
```

Se keyA for relativamente primo com o tamanho do conjunto de símbolos, essas duas chaves selecionadas aleatoriamente serão combinadas em uma única chave, multiplicando-se keyA pelo tamanho do conjunto de símbolos e adicionando-se keyB ao produto. (Observe que isso é o oposto da função getKeyParts () , que divide uma única chave inteira em dois inteiros.) A linha 76 retorna esse valor da função getRandomKey () .

Se a condição na linha 75 retornar False , o código retornará ao início do loop while na linha 73 e selecionará números aleatórios para keyA e keyB novamente. O loop infinito garante que o programa continue em loop até encontrar números aleatórios que sejam chaves válidas.

Chamando a função main ()

As linhas 81 e 82 chamam a função main () se este programa foi executado por si próprio em vez de ser importado por outro programa:

```
79. # Se affineCipher.py for executado (em vez de importado como um módulo),  
    chame
```

```
80. # a função main ():  
81. if __name__ == '__main__':  
82.     main ()
```

Isso garante que a função main () seja executada quando o programa for executado, mas não quando o programa for importado como um módulo.

Resumo

Assim como fizemos no [Capítulo 9](#), neste capítulo, escrevemos um programa (*affineKeyTest.py*) que pode testar nosso programa de criptografia. Usando este programa de teste, você aprendeu que a cifra afim tem aproximadamente 1320 chaves possíveis, que é um número que você pode facilmente hackear usando força bruta. Isso significa que teremos que jogar a cifra afim na pilha de cifras fracas facilmente hackáveis.

Portanto, a cifra afim não é muito mais segura do que as cifras anteriores que examinamos. A cifra de transposição pode ter mais chaves possíveis, mas o número de chaves possíveis é limitado ao tamanho da mensagem. Para uma mensagem com apenas 20 caracteres, a cifra de transposição pode ter no máximo 18 chaves, com chaves variando de 2 a 19. Você pode usar a cifra afim para criptografar mensagens curtas com mais segurança do que a cifra de César fornece, porque seu número de possíveis chaves é baseado no conjunto de símbolos.

No [Capítulo 15](#), vamos escrever um programa de força bruta que pode quebrar mensagens criptografadas por códigos!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. A cifra afim é a combinação de quais outras duas cifras?
2. O que é uma tupla? Como uma tupla é diferente de uma lista?
3. Se a chave A é 1, por que ela torna a cifra afim fraca?
4. Se a chave B é 0, por que ela torna a cifra afim fraca?

15

ATACANDO A CIPE DE AFFINE

“A criptoanálise não poderia ser inventada até que uma civilização tivesse atingido um nível de escolaridade suficientemente sofisticado em várias disciplinas, incluindo matemática, estatística e lingüística”.

- Simon Singh, The Code Book



No [Capítulo 14](#), você aprendeu que a cifra afim está limitada a apenas alguns milhares de chaves, o que significa que podemos facilmente executar um ataque de força bruta contra ela. Neste capítulo, você aprenderá a escrever um programa que pode quebrar mensagens criptografadas por códigos afim.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- O operador expoente (`**`)
- A instrução continue

Código Fonte do Programa Hacker de Cifra Afim

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como `affineHacker.py`. Digitar a string da variável `myMessage` manualmente pode ser complicado, então você pode copiá-la e colá-la no arquivo `affineHacker.py` disponível em <https://www.nostarch.com/crackingcodes/> para economizar tempo. Certifique-se de que `dictionary.txt` e `pyperclip.py`, `affineCipher.py`, `detectEnglish.py` e `cryptomath.py` estejam no mesmo diretório que `affineHacker.py`.

`affineHacker.py`

```
1. # Affine Cipher Hacker
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip, affineCipher, detectEnglish, cryptomath
5
6. SILENT_MODE = False
7
8. def main ():
```

```
9. # Você pode querer copiar e colar este texto do código-fonte em
10. # https://www.nostarch.com/crackingcodes/.
11. myMessage = """5QG9ol3La6QI93! XQxaia6faQL9QdaQG1 !! axQARLa
!! A
uaRLQADQALQG93! xQxaGaAfaQ1QX3o1RQARL9Qda!
AafARuQLX1LQALQI1
iQX3o1RN "Q-5! 1RQP36ARu" """
12
13. hackedMessage = hackAffine (myMessage)
14
15. se hackedMessage! = Nenhum:
16. # O texto original é exibido na tela. Para a conveniência de
17. # o usuário, copiamos o texto do código para a área de transferência:
18. print ('Copiando a mensagem hackeada para a área de transferência:')
19. print (hackedMessage)
20. pyperclip.copy (hackedMessage)
21. else:
22. print ('Falha ao hackar criptografia')
23
24
25. def hackAffine (mensagem):
26. print ('Hacking ...')
27
28. # Os programas em Python podem ser interrompidos a qualquer momento
pressionando Ctrl-C (em
29. # Windows) ou Ctrl-D (no macOS e no Linux):
30. print ('(Pressione Ctrl-C ou Ctrl-D para sair a qualquer momento.)')
31
32. # Brute-force percorrendo todas as chaves possíveis:
33. para chave no intervalo (len (affineCipher.SYMBOLS) ** 2):
34. keyA = affineCipher.getKeyParts (chave) [0]
35. if cryptomath.gcd (keyA, len (affineCipher.SYMBOLS))! = 1:
36. continue
37
38. decryptedText = affineCipher.decryptMessage (chave, mensagem)
39. se não SILENT_MODE:
40. print ('Chave Tentei% s ... (% s)% (key, decryptedText [: 40]))
```

```

41.
42. se detectEnglish.isEnglish (decryptedText):
43. # Verifique com o usuário se a chave descriptografada foi encontrada:
44. print ()
45. print ('Possível corte de criptografia:')
46. print ('Chave:% s'% (chave))
47. print ('Mensagem descriptografada:' + decryptedText [: 200])
48. print ()
49. print ('Digite D para terminar, ou apenas pressione Enter para continuar
hacking: ')
50. response = input ('>')
51
52. if response.strip (). Upper () . Startswith ('D'):
53. return decryptedText
54. retorno Nenhum
55
56
57. # Se affineHacker.py for executado (em vez de importado como um módulo),
chame
58. # a função main () :
59. se __name__ == '__main__':
60. main ()

```

Execução de Amostra do Programa Hacker de Cifra Afim

Pressione F5 no editor de arquivos para executar o programa *affineHacker.py* ; a saída deve ficar assim:

Hacking ...

(Pressione Ctrl-C ou Ctrl-D para sair a qualquer momento.)

Tentei a Chave 95 ... (U & '<3dJ ^ Gjx'-3 ^ MS'Sj0jxuj'G3'% j '<mMMjS'g)

Tentei Chave 96 ... (T% &; 2cI) Fiw &, 2] LR & Ri / iwti & F2 & \$ i &; lLLiR & f)

Tentei a chave 97 ... (S%: 1bH \ Ehv% + 1 \ KQ% Qh.hvsh% E1% # h%: kKKhQ% e)

- recorte -

Tentei a Chave 2190 ... (? ^ =! - +. 32 # 0 = 5-3 * "!" # 1 # 04 # = 2- = # =! ~ ** # "!")

Tentei a chave 2191 ... (^ BNLOTSDQ ^ VNTKC ^ CDRDQUD ^ SN ^ AD ^

B@KKDC ^ H)

Tentei Key 2192 ... ("Um computador mereceria ser chamado i)

Hack de criptografia possível:

Chave: 2192

Mensagem descriptografada: "Um computador mereceria ser chamado de inteligente se

poderia enganar um humano a acreditar que era humano. "-Alan Turing

Digite D para pronto, ou apenas pressione Enter para continuar o hacking:

> d

Copiando mensagem hackeada para a área de transferência:

"Um computador mereceria ser chamado de inteligente se pudesse enganar um humano

em acreditar que era humano. "- Alan Turing

Vamos dar uma olhada mais de perto em como funciona o programa hacker de criptografia afim.

Configurando módulos, constantes e a função main ()

O programa hacker de criptografia afim tem 60 linhas de comprimento porque já escrevemos muito do código que ele usa. A linha 4 importa os módulos que criamos nos capítulos anteriores:

1. # Affine Cipher Hacker
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar pyperclip, affineCipher, detectEnglish, cryptomath
- 5
6. SILENT_MODE = False

Quando você executa o programa hacker de cifra afim, você verá que ele produz muitos resultados à medida que funciona em todas as possíveis descodificações. No entanto, a impressão de toda essa saída desacelera o programa. Se você quiser acelerar o programa, defina a variável SILENT_MODE na linha 6 como True para impedir que todas as mensagens sejam impressas.

Em seguida, configuramos a função main () :

8. def main ():
9. # Você pode querer copiar e colar este texto do código-fonte em
10. # <https://www.nostarch.com/crackingcodes/>.

```
11. myMessage = """ "5QG9ol3La6QI93! XQxaia6faQL9QdaQG1 !! axQARLa
!! A
uaRLQADQALQG93! xQxaGaAfaQ1QX3o1RQARL9Qda!
AafARuQLX1LQALQI1
iQX3o1RN "Q-5! 1RQP36ARu" """
12
13. hackedMessage = hackAffine (myMessage)
```

O texto cifrado a ser hackeado é armazenado como uma string em myMessage na linha 11, e essa string é passada para a função hackAffine () , que veremos na próxima seção. O valor de retorno desta chamada é uma string da mensagem original se o texto cifrado foi hackeado ou o valor None se o hack falhou.

O código nas linhas 15 a 22 verifica se hackedMessage foi definido como Nenhum :

```
15. se hackedMessage!= Nenhum:
16. # O texto original é exibido na tela. Para a conveniência de
17. # o usuário, copiamos o texto do código para a área de transferência:
18. print ('Copiando a mensagem hackeada para a área de transferência:')
19. print (hackedMessage)
20. pyperclip.copy (hackedMessage)
21. else:
22. print ('Falha ao hackar criptografia')
```

Se hackedMessage não for igual a None , a mensagem será impressa na tela na linha 19 e copiada para a área de transferência na linha 20. Caso contrário, o programa simplesmente imprime feedback ao usuário de que não foi possível hackear o texto cifrado. Vamos dar uma olhada em como a função hackAffine () funciona.

A função de hacking de código afim

A função hackAffine () começa na linha 25 e contém o código para descriptografia. Começa imprimindo algumas instruções para o usuário:

```
25. def hackAffine (mensagem):
26. print ('Hacking ...')
27
28. # Os programas em Python podem ser interrompidos a qualquer momento
pressionando Ctrl-C (em
```

29. # Windows) ou Ctrl-D (no macOS e no Linux):

30. print ('(Pressione Ctrl-C ou Ctrl-D para sair a qualquer momento.)')

O processo de decodificação pode demorar um pouco, portanto, se o usuário quiser sair do programa mais cedo, ele poderá pressionar ctrl -C (no Windows) ou ctrl -D (no macOS e no Linux).

Antes de continuarmos com o restante do código, você precisa aprender sobre o operador do expoente.

O operador do expoente

Um operador matemático útil que você precisa saber para entender o programa hacker de criptografia afim (além dos operadores básicos + , - , * , / e //) é o *operador de expoente* (**). O operador do expoente aumenta um número para o poder de outro número. Por exemplo, dois ao poder de cinco seriam $2^{**} 5$ em Python. Isto é equivalente a dois multiplicados por si cinco vezes: $2 * 2 * 2 * 2 * 2$. Ambas as expressões, $2^{**} 5$ e $2 * 2 * 2 * 2 * 2$, avaliam o inteiro 32 .

Digite o seguinte no shell interativo para ver como o operador ** funciona:

```
>>> 5 ** 2  
25  
>>> 2 ** 5  
32  
>>> 123 ** 10  
792594609605189126649
```

A expressão $5^{**} 2$ é avaliada como 25 porque 5 multiplicada por si mesma é equivalente a 25 . Da mesma forma, $2^{**} 5$ retorna 32 porque 2 multiplicado por si mesmo cinco vezes é 32 .

Vamos voltar ao código-fonte para ver o que o operador ** faz no programa.

Calculando o número total de chaves possíveis

A linha 33 usa o operador ** para calcular o número total de chaves possíveis:

32. # Brute-force percorrendo todas as chaves possíveis:

33. para chave no intervalo (len (affineCipher.SYMBOLS) ** 2):

34. keyA = affineCipher.getKeyParts (chave) [0]

Sabemos que existem, no máximo, len (affineCipher.SYMBOLS) possíveis inteiros para os inteiros possíveis de Key A e len (affineCipher.SYMBOLS) para a Key B. Para obter o intervalo inteiro de chaves possíveis, multiplicamos esses

valores juntos. Como estamos multiplicando o mesmo valor por si só, podemos usar o operador ** na expressão len (affineCipher.SYMBOLS) ** 2 .

A linha 34 chama a função getKeyParts () que usamos em *affineCipher.py* para dividir uma única chave inteira em dois inteiros. Neste exemplo, estamos usando a função para obter a parte da chave A da chave que estamos testando. Lembre-se de que o valor de retorno desta chamada de função é uma tupla de dois inteiros: um para a Chave A e um para a Chave B. A linha 34 armazena o primeiro inteiro da tupla na chaveA colocando o [0] após a chamada da função hackAffine () .

Por exemplo, affineCipher.getKeyParts (key) [0] avalia a tupla e o índice (42, 22) [0] , que então avalia como 42 , o valor no índice 0 da tupla. Isso obtém apenas a parte Key A do valor de retorno e a armazena na variável keyA . A parte da Chave B (o segundo valor na tupla retornada) é ignorada porque não precisamos da Chave B para calcular se a Chave A é válida. As linhas 35 e 36 verificam se o código A é uma Chave A válida para a cifra afim e, se não, o programa continua para a próxima chave a ser tentada. Para entender como a execução retorna ao início do loop, você precisa aprender sobre a instrução continue .

A continuação da declaração

A instrução continue usa a palavra-chave continue por si só e não usa parâmetros. Usamos uma instrução continue dentro de um tempo ou loop. Quando uma instrução continue é executada, a execução do programa imediatamente salta para o início do loop para a próxima iteração. Isso também acontece quando a execução do programa atinge o final do bloco do loop. Mas uma instrução continue faz com que a execução do programa retorne ao início do loop antes que ele atinja o final do loop.

Digite o seguinte no shell interativo:

```
>>> para i na faixa (3):
```

```
... imprimir (i)
```

```
... print ('Olá!')
```

```
...
```

```
0
```

```
Olá!
```

```
1
```

```
Olá!
```

2

Olá!

O loop for faz um loop através do objeto range , e o valor em i torna-se cada inteiro de 0 até, mas não incluindo, 3 . Em cada iteração, a chamada de função de impressão ('Hello!') Exibe Hello! na tela.

Agora, contraste para loop com o próximo exemplo, que é o mesmo do exemplo anterior, exceto por ter uma instrução continue antes da linha de impressão ('Hello!') .

>>> para i na faixa (3):

... imprimir (i)

... continua

... print ('Olá!')

...

0

1

2

Observe que o Hello! nunca é impresso, porque a instrução continue faz com que a execução do programa retorne ao início do loop for para a próxima iteração e a execução nunca chegue à linha de impressão ('Hello!') .

Uma instrução continue é frequentemente colocada dentro do bloco de uma instrução if para que a execução continue no início do loop sob certas condições. Vamos voltar ao nosso código para ver como ele usa a instrução continue para ignorar a execução, dependendo da chave usada.

Usando continuar para pular código

No código-fonte, a linha 35 usa a função gcd () no módulo cryptomath para determinar se a Chave A é relativamente primo para o tamanho do conjunto de símbolos:

35. if cryptomath.gcd (keyA, len (affineCipher.SYMBOLS))! = 1:

36. continue

Lembre-se de que dois números são relativamente primos se seu maior divisor comum (GCD) for 1. Se a Chave A e o tamanho do conjunto de símbolos não forem relativamente primos, a condição na linha 35 é True e a instrução continue na linha 36 é executada. Isso faz com que a execução do programa retorne ao início do loop para a próxima iteração. Como resultado, o programa ignora a

chamada para `decryptMessage()` na linha 38 se a chave é inválida e continua a tentar outras chaves até encontrar a chave correta.

Quando o programa encontra a chave certa, a mensagem é descriptografada chamando `decryptMessage()` com a chave na linha 38:

```
38. decryptedText = affineCipher.decryptMessage (chave, mensagem)
39. se não SILENT_MODE:
40. print ('Chave Tentei% s ... (% s)% (key, decryptedText [: 40]))
```

Se `SILENT_MODE` foi definido como `False`, a mensagem de chave de teste é impressa na tela, mas se ela estiver definida como `True`, a chamada de `print()` na linha 40 será ignorada.

Em seguida, a linha 42 usa a função `isEnglish()` do módulo `detectEnglish` para verificar se a mensagem descriptografada é reconhecida como inglês:

```
42. se detectEnglish.isEnglish (decryptedText):
43. # Verifique com o usuário se a chave descriptografada foi encontrada:
44. print ()
45. print ('Possível corte de criptografia:')
46. print ('Chave:% s'% (chave))
47. print ('Mensagem descriptografada:' + decryptedText [: 200])
48. print ()
```

Se a chave de descriptografia errada foi usada, a mensagem descriptografada pareceria com caracteres aleatórios e `isEnglish()` retornaria `False`. Mas se a mensagem descriptografada for reconhecida como legível em inglês (pelos padrões da função `isEnglish()`), o programa a exibirá para o usuário.

Exibimos um trecho da mensagem descriptografada que é reconhecida como em inglês, porque a função `isEnglish()` pode erroneamente identificar texto como inglês, mesmo que não tenha encontrado a chave correta. Se o usuário decidir que essa é realmente a descriptografia correta, ele pode digitar D e, em seguida, pressionar Enter .

```
49. print ('Digite D para terminar, ou apenas pressione Enter para continuar
hacking: ')
50. response = input ('>')
51
52. if response.strip ().Upper ().Startswith ('D'):
53. return decryptedText
```

Caso contrário, o usuário pode simplesmente pressionar enter para retornar uma string em branco da chamada input () , e a função hackAffine () continuaria tentando mais chaves.

A partir do recuo no início da linha 54, você pode ver que esta linha é executada após a conclusão do loop for na linha 33:

54. retorno Nenhum

Se o loop for terminar e atingir a linha 54, ele passou por todas as chaves de descriptografia possíveis sem encontrar a correta. Neste ponto, a função hackAffine () retorna o valor None para sinalizar que não foi bem sucedido em hackear o texto cifrado.

Se o programa tivesse encontrado a chave correta, a execução teria retornado anteriormente da função na linha 53 e nunca atingido a linha 54.

Chamando a função main ()

Se rodarmos o *affineHacker.py* como um programa, a variável especial `__name__` será configurada para a string '`__main__`' ao invés de '`affineHacker`' . Neste caso, chamamos a função main () .

57. # Se affineHacker.py for executado (em vez de importado como um módulo), chame

58. # a função main ():

59. se `__name__ == '__main__'`:

60. `main()`

Isso conclui o programa affine cipher hacking.

Resumo

Este capítulo é bastante curto porque não introduz nenhuma nova técnica de hacking. Como você viu, desde que o número de chaves possíveis seja de apenas alguns milhares, não demorará muito para os computadores usarem todas as chaves possíveis e usarem a função `isEnglish()` para procurar a chave certa.

Você aprendeu sobre o operador expoente (`**`), o que aumenta um número para o poder de outro número. Você também aprendeu como usar a instrução `continue` para enviar a execução do programa de volta ao início do loop, em vez de esperar até que a execução chegue ao final do bloco.

Convenientemente, já escrevemos muito do código usado para o hacker de

criptografia afim em *affineCipher.py* , *detectEnglish.py* e *cryptomath.py* . O truque da função main () nos ajuda a reutilizar o código em nossos programas.

No [Capítulo 16](#) , você aprenderá sobre a simples codificação de substituição, na qual os computadores não podem usar força bruta. O número de chaves possíveis para essa cifra é mais de trilhões de trilhões! Um único laptop não poderia passar por uma fração dessas chaves durante a nossa vida, o que torna a cifra imune a um ataque de força bruta.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. O que $2^{**} 5$ avalia para?
2. O que $6^{**} 2$ avalia para?
3. O que o código a seguir imprime?

para eu na faixa (5):

se eu == 2:

continuar

imprimir (i)

4. A função main () de *affineHacker.py* é chamada se outro programa executar o import *affineHacker* ?

16

PROGRAMANDO A CIPRA DE SUBSTITUIÇÃO SIMPLES

“A internet é a ferramenta mais libertadora que a humanidade já inventou e também a melhor para a vigilância. Não é um nem o outro. São os dois.

John Perry Barlow, co-fundador da Electronic Frontier Foundation



No [Capítulo 15](#) , você aprendeu que a cifra afim tem cerca de mil chaves

possíveis, mas que os computadores ainda podem usar força bruta em todos eles facilmente. Precisamos de uma cifra com tantas chaves possíveis que nenhum computador possa forçar a força bruta por todas elas.

A *simples cifra de substituição* é uma dessas cifras que é efetivamente invulnerável a um ataque de força bruta porque possui um número enorme de chaves possíveis. Mesmo se o seu computador pudesse experimentar um trilhão de chaves a cada segundo, ainda levaria 12 milhões de anos para ele experimentar cada um! Neste capítulo, você irá escrever um programa para implementar a simples cifra de substituição e aprender algumas funções e métodos de string úteis do Python também.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- O método de lista `sort()`
- Livrar-se de caracteres duplicados de uma string
- Funções de invólucro
- Os métodos de string `isupper()` e `islower()`

Como funciona o código de substituição simples

Para implementar a simples cifra de substituição, escolhemos uma letra aleatória para criptografar cada letra do alfabeto, usando cada letra apenas uma vez. A chave para a simples cifra de substituição é sempre uma seqüência de 26 letras do alfabeto em ordem aleatória. Existem 403,291,461,126,605,635,584,000,000 diferentes ordenações de chaves possíveis para a simples cifra de substituição. Isso é um monte de chaves! Mais importante, esse número é tão grande que é impossível forçar a força bruta. (Para ver como esse número foi calculado, acesse <https://www.nostarch.com/crackingcodes/>.)

Vamos tentar usar a simples cifra de substituição com papel e lápis primeiro. Para este exemplo, vamos criptografar a mensagem "Attack at dawn" (Ataque ao amanhecer) usando a chave VJZBGNFEPLITMXDWKQUCRYAHSO. Primeiro, escreva as letras do alfabeto e a tecla correspondente embaixo de cada letra, como na [Figura 16-1](#).

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O

Figura 16-1: Letras de criptografia para a chave de exemplo

Para criptografar uma mensagem, encontre a letra no texto simples na linha

superior e substitua-a pela letra na linha inferior. A criptografa para V , T criptografa para C , C criptografa para Z e assim por diante. Assim, a mensagem "Ataque ao amanhecer" criptografa para "Vccvzi vc bvax".

Para descriptografar a mensagem criptografada, encontre a letra no texto cifrado na linha inferior e substitua-a pela letra correspondente na linha superior. V decriptografa para A , C descriptografa para T , Z descriptografa para C e assim por diante.

Ao contrário da cifra de César, na qual a linha de baixo se desloca, mas permanece em ordem alfabética, na simples cifra de substituição, a linha de baixo está completamente embaralhada. Isso resulta em muito mais chaves possíveis, o que é uma enorme vantagem de usar a simples cifra de substituição. A desvantagem é que a chave tem 26 caracteres e é mais difícil de memorizar. Você pode precisar anotar a chave, mas se fizer isso, certifique-se de que ninguém mais a leia!

Código-fonte para o programa de codificação de substituição simples

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File** .

Digite o seguinte código no editor de arquivos e salve-o como *simpleSubCipher.py* . Certifique-se de colocar o arquivo *pyperclip.py* no mesmo diretório que o arquivo *simpleSubCipher.py* . Pressione F5 para executar o programa.

```
simpleSub  
Cipher.py
```

```
1. # Cifra de Substituição Simples  
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)  
3  
4. importar pyperclip, sys, random  
5  
6  
7. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
8  
9. def main ():  
10. myMessage = 'Se um homem é oferecido um fato que vai contra o seu  
instintos, ele vai examiná-lo de perto, e a menos que a evidência
```

é esmagadora, ele se recusará a acreditar. Se, por outro mão, ele é oferecido algo que oferece uma razão para agir de acordo com seus instintos, ele vai aceitá-lo mesmo no menor evidência. A origem dos mitos é explicada dessa maneira.

-Bertrand Russell

```
11. myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'  
12. myMode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.  
13  
14. se keyIsValid (myKey):  
15.     sys.exit ('Há um erro na chave ou no conjunto de símbolos.')  
16. if myMode == 'encriptar':  
17.     translated = encryptMessage (myKey, myMessage)  
18. elif myMode == 'decifrar':  
19.     translated = decryptMessage (myKey, myMessage)  
20. print ('Usando a tecla% s'% (myKey))  
21. print ('A mensagem% sed é:% (myMode))  
22. print (tradução)  
23. pyperclip.copy (tradução)  
24. print ()  
25. print ('Esta mensagem foi copiada para a área de transferência.')  
26  
27  
28. def keyIsValid (chave):  
29.     keyList = lista (chave)  
30.     lettersList = list (LETRAS)  
31.     keyList.sort ()  
32.     lettersList.sort ()  
33  
34. return keylist == letrasLista  
35  
36  
37. def encryptMessage (key, message):  
38.     return translateMessage (chave, mensagem, 'criptografar')  
39  
40.  
41. def decryptMessage (chave, mensagem):  
42.     return translateMessage (chave, mensagem, 'decifrar')
```

```
43
44
45. def translateMessage (chave, mensagem, modo):
46.     traduzido = "
47.     charsA = LETRAS
48.     charsB = chave
49.     if mode == 'descriptografar':
50.         # Para descriptografar, podemos usar o mesmo código que a criptografia.
Nós
51.         # só precisa trocar onde a chave e as LETRAS são usadas.
52.         charsA, charsB = charsB, charsA
53
54.     # Loop através de cada símbolo na mensagem:
55.     para o símbolo na mensagem:
56.     if symbol.upper () em charsA:
57.         # Criptografar / descriptografar o símbolo:
58.         symIndex = charsA.find (symbol.upper ())
59.         if symbol.isupper ():
60.             traduzido + = charsB [symIndex] .upper ()
61.         else:
62.             traduzido + = charsB [symIndex] .lower ()
63.         else:
64.             # Símbolo não está em LETRAS; apenas adicione:
65.             traduzido + = símbolo
66
67.     retornar traduzido
68
69
70. def getRandomKey ():
71.     key = list (LETTERS)
72.     random.shuffle (chave)
73.     return " .join (chave)
74
75
76. if __name__ == '__main__':
77.     main ()
```

Execução de amostra do programa de codificação de substituição simples

Quando você executa o programa *simpleSubCipher.py*, a saída criptografada deve ficar assim:

Usando a chave LFWOAYUISVKMNXPBDCRJTQEGHZ

A mensagem criptografada é:

Syl nlx sr pyyacao l ylwj eiswi upar lulsxrj isr srxjsxwjr, ia esmm
rwctjsxsza sj wmptramh, lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia esmm
caytra

jp famsaqa sj. Sy, px jia pjiac ilxo, ia sr pyyacao rpnajisxu eiswi lyppcor
l calppx ypc lwjsxu sx lwwpcolxwa jp isr srxjsxwjr, ia esmm lwwabj sj aqax
px jia rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.

-Facjclxo Ctrramm

Esta mensagem foi copiada para a área de transferência.

Observe que, se a letra do texto original for minúscula, ela será minúscula no texto cifrado. Da mesma forma, se a letra é maiúscula no texto original, ela é maiúscula no texto cifrado. A simples cifra de substituição não criptografa espaços ou sinais de pontuação e simplesmente retorna esses caracteres como estão.

Para descriptografar esse texto cifrado, cole-o como o valor da variável myMessage na linha 10 e altere myMode para a string 'decrypt'. Quando você executa o programa novamente, a saída de descriptografia deve ficar assim:

Usando a chave LFWOAYUISVKMNXPBDCRJTQEGHZ

A mensagem descriptografada é:

Se um homem é oferecido um fato que vai contra seus instintos, ele vai escrutiná-lo de perto, e a menos que a evidência seja esmagadora, ele recusará acreditar. Se, por outro lado, ele é oferecido algo que oferece uma razão para agir de acordo com seus instintos, ele vai aceitá-lo mesmo na menor evidência. A origem dos mitos é explicada dessa maneira.

- Bertrand Russell

Esta mensagem foi copiada para a área de transferência.

Configurando módulos, constantes e a função main()

Vamos ver as primeiras linhas do código-fonte do programa de codificação de substituição simples.

```
1. # Cifra de Substituição Simples
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip, sys, random
5
6
7. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

A linha 4 importa os módulos `pyperclip`, `sys` e `random`. A variável constante `LETTERS` é configurada para uma cadeia de todas as letras maiúsculas, que é o conjunto de símbolos para o programa de codificação de substituição simples.

A função `main()` em *simpleSubCipher.py*, que é semelhante à função `main()` dos programas de codificação nos capítulos anteriores, é chamada quando o programa é executado pela primeira vez. Ele contém as variáveis que armazenam a mensagem, a chave e o modo usado para o programa.

9. `def main():`

```
10. myMessage = 'Se um homem é oferecido um fato que vai contra o seu
    instintos, ele vai examiná-lo de perto, e a menos que a evidência
    é esmagadora, ele se recusará a acreditar. Se, por outro
    mão, ele é oferecido algo que oferece uma razão para agir
    de acordo com seus instintos, ele vai aceitá-lo mesmo no
    menor evidência. A origem dos mitos é explicada dessa maneira.
```

-Bertrand Russell

```
11. myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
12. myMode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.
```

As chaves para simples cifras de substituição são fáceis de serem erradas porque são bastante longas e precisam ter todas as letras do alfabeto. Por exemplo, é fácil digitar uma chave que está faltando uma letra ou uma chave que tenha a mesma letra duas vezes. A função `keyIsValid()` garante que a chave é utilizável pelas funções de criptografia e descriptografia, e a função sai do programa com uma mensagem de erro se a chave não for válida:

```
14. se keyIsValid (myKey):
15.     sys.exit ('Há um erro na chave ou no conjunto de símbolos.')
```

Se a linha 14 retornar False de keyIsValid () , myKey conterá uma chave inválida e a linha 15 encerrará o programa.

As linhas 16 a 19 verificam se a variável myMode está configurada para 'encrypt' ou 'decrypt' e chama encryptMessage () ou decryptMessage () de acordo:

16. if myMode == 'encriptar':
17. translated = encryptMessage (myKey, myMessage)
18. elif myMode == 'decifrar':
19. translated = decryptMessage (myKey, myMessage)

O valor de retorno de encryptMessage () e decryptMessage () é uma cadeia de caracteres da mensagem criptografada ou descriptografada armazenada na variável traduzida .

A linha 20 imprime a chave que foi usada na tela. A mensagem criptografada ou descriptografada é impressa na tela e também copiada para a área de transferência.

20. print ('Usando a tecla% s'% (myKey))
21. print ('A mensagem% sed é:% ' (myMode))
22. print (tradução)
23. pyperclip.copy (tradução)
24. print ()
25. print ('Esta mensagem foi copiada para a área de transferência.')

A linha 25 é a última linha de código na função main () , portanto a execução do programa retorna após a linha 25. Quando a chamada main () é feita na última linha do programa, o programa sai.

Em seguida, veremos como a função keyIsValid () usa o método sort () para testar se a chave é válida.

O método de lista sort ()

As listas têm um método sort () que reorganiza os itens da lista em ordem numérica ou alfabética. Essa capacidade de classificar itens em uma lista é útil quando você precisa verificar se duas listas contêm os mesmos itens, mas não listá-los na mesma ordem.

Em *simpleSubCipher.py* , um valor de string de chave de substituição simples é válido apenas se tiver cada um dos caracteres no conjunto de símbolos sem letras

duplicadas ou ausentes. Podemos verificar se um valor de string é uma chave válida, classificando-o e verificando se é igual às LETTERS ordenadas. Mas porque podemos classificar apenas listas, não seqüências de caracteres (lembre-se de que strings são imutáveis, ou seja, seus valores não podem ser alterados), obteremos versões de lista dos valores da string passando-os para list () . Então, depois de classificar essas listas, podemos comparar as duas para ver se são iguais ou não. Embora LETTERS já esteja em ordem alfabética, vamos classificá-lo porque vamos expandi-lo para conter outros caracteres mais tarde.

```
28. def keyIsValid (chave):  
29.     keyList = lista (chave)  
30.     lettersList = list (LETRAS)  
31.     keyList.sort ()  
32.     lettersList.sort ()
```

A string na chave é passada para list () na linha 29. O valor da lista retornado é armazenado em uma variável chamada keyList .

Na linha 30, a variável constante LETTERS (que contém a string 'ABCDEFGHIJKLMNPQRSTUVWXYZ') é passada para list () , que retorna a lista no seguinte formato: ['A', 'B', 'C', 'D', E, F, G, H, I, J, K, L, M, N, O, P, Q R, S, T, U, V, W, X, Y, Z.

Nas linhas 31 e 32, as listas em keyList e lettersList são então classificadas em ordem alfabética, chamando o método list () sort sobre elas. Observe que, semelhante ao método de lista append () , o método de lista sort () modifica a lista no lugar e não possui um valor de retorno.

Quando classificados, os valores keyList e lettersList *devem* ser os mesmos, porque keyList era simplesmente os caracteres em LETTERS com a ordem embaralhada. A linha 34 verifica se os valores keyList e lettersList são iguais:

```
34. return keylist == letrasLista
```

Se keyList e lettersList forem iguais, você pode ter certeza de que keyList e o parâmetro key não possuem nenhum caractere duplicado, porque LETTERS não possui duplicatas. Nesse caso, a linha 34 retorna True . Mas se keyList e lettersList não corresponderem, a chave é inválida e a linha 34 retorna False .

Funções de invólucro

O código de criptografia e o código de descriptografia no programa

simpleSubCipher.py são quase idênticos. Quando você tem dois pedaços de código muito semelhantes, é melhor colocá-los em uma função e chamá-lo duas vezes, em vez de digitar o código duas vezes. Isso não apenas economiza tempo, mas, mais importante, evita a introdução de erros ao copiar e colar o código. Também é vantajoso, porque se houver algum bug no código, você só precisa corrigir o bug em um lugar, e não em vários lugares.

As funções de invólucro ajudam a evitar a necessidade de inserir código duplicado envolvendo o código de outra função e retornando o valor retornado pela função empacotada. Muitas vezes, a função wrapper faz uma ligeira mudança nos argumentos ou valor de retorno da função empacotada. Caso contrário, não haveria necessidade de empacotamento porque você poderia simplesmente chamar a função diretamente.

Vejamos um exemplo do uso de funções de wrapper em nosso código para entender como elas funcionam. Nesse caso, `encryptMessage ()` e `decryptMessage ()` nas linhas 37 e 41 são as funções do wrapper:

```
37. def encryptMessage (key, message):
38.     return translateMessage (chave, mensagem, 'criptografar')
39
40.
41. def decryptMessage (chave, mensagem):
42.     return translateMessage (chave, mensagem, 'decifrar')
```

Cada uma dessas funções de wrapper chama o `translateMessage ()`, que é a função wrapped, e retorna o valor que o `translateMessage ()` retorna. (Vamos examinar a função `translateMessage ()` na próxima seção.) Como ambas as funções do wrapper usam a mesma função `translateMessage ()`, precisamos modificar apenas essa função em vez das funções `encryptMessage ()` e `decryptMessage ()` se precisar fazer alterações na cifra.

Com essas funções de wrapper, alguém que importa o programa *simpleSubCipher.py* pode chamar as funções nomeadas `encryptMessage ()` e `decryptMessage ()` da mesma maneira que pode fazer com todos os outros programas de criptografia neste livro. As funções do wrapper têm nomes claros que informam aos outros que usam as funções o que eles fazem sem precisar examinar o código. Como resultado, se quisermos compartilhar nosso código, outros poderão usá-lo mais facilmente.

Outros programas podem criptografar uma mensagem em várias cifras

importando os programas de codificação e chamando suas funções encryptMessage () , como mostrado aqui:

```
import affineCipher, simpleSubCipher, transpositionCipher  
- recorte -  
ciphertext1 = affineCipher.encryptMessage (encKey1, 'Hello!')  
ciphertext2 = transpositionCipher.encryptMessage (encKey2, 'Hello!')  
ciphertext3 = simpleSubCipher.encryptMessage (encKey3, 'Hello!')
```

A consistência de nomenclatura é útil, porque torna mais fácil para alguém familiarizado com um dos programas de criptografia usar os outros programas de criptografia. Por exemplo, você pode ver que o primeiro parâmetro é sempre a chave e o segundo parâmetro é sempre a mensagem, que é a convenção usada para a maioria dos programas de criptografia neste livro. Usar a função translateMessage () em vez de funções separadas encryptMessage () e decryptMessage () seria inconsistente com os outros programas.

Vamos ver a função translateMessage () a seguir.

A função translateMessage ()

A função translateMessage () é usada para criptografia e decriptografia.

```
45. def translateMessage (chave, mensagem, modo):  
46.     traduzido = "  
47.     charsA = LETRAS  
48.     charsB = chave  
49.     if mode == 'descriptografar':  
50.         # Para descriptografar, podemos usar o mesmo código que a criptografia.  
Nós  
51.         # só precisa trocar onde a chave e as LETRAS são usadas.  
52.         charsA, charsB = charsB, charsA
```

Observe que translateMessage () possui a chave e a mensagem de parâmetros, mas também um terceiro parâmetro chamado mode . Quando chamamos translateMessage () , a chamada na função encryptMessage () passa 'encrypt' para o parâmetro mode e a chamada na função decryptMessage () passa a 'decrypt' . É assim que a função translateMessage () sabe se deve criptografar ou descriptografar a mensagem transmitida a ela.

O processo de criptografia real é simples: para cada letra no parâmetro de mensagem , a função pesquisa o índice dessa letra em LETTERS e substitui o

caractere pela letra nesse mesmo índice no parâmetro- chave . A descriptografia faz o oposto: procura o índice na chave e substitui o caractere pela letra no mesmo índice em LETTERS .

Em vez de usar LETTERS e key , o programa usa as variáveis charsA e charsB , que permitem substituir a letra em charsA pela letra no mesmo índice em charsB . Ser capaz de alterar quais valores são atribuídos a charsA e charsB torna mais fácil para o programa alternar entre criptografar e descriptografar. A linha 47 define os caracteres em charsA para os caracteres em LETTERS e a linha 48 define os caracteres em charsB para os caracteres em key .

As figuras a seguir mostram como o mesmo código pode ser usado para criptografar ou descriptografar uma letra. [A Figura 16-2](#) ilustra o processo de criptografia. A linha superior nesta figura mostra os caracteres em charsA (definido como LETTERS), a linha do meio mostra os caracteres em charsB (set to key) e a linha inferior mostra os índices inteiros correspondentes aos caracteres.

charsA	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
charsB	V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figura 16-2: Usando o índice para criptografar texto simples

O código em translateMessage () sempre procura o índice do caractere de mensagem no charsA e o substitui pelo caractere correspondente em charsB naquele índice. Então, para criptografar, apenas deixamos charsA e charsB como estão. Usando as variáveis charsA e charsB substitui o caractere em LETTERS pelo caractere em key , porque charsA é setado para LETTERS e charsB é setado para key .

Para descriptografar, os valores em charsA e charsB são alternados usando charsA, charsB = charsB, charsA na linha 52. A [Figura 16-3](#) mostra o processo de descriptografia.

charsA	V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O
charsB	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figura 16-3: Usando o índice para descriptografar o texto cifrado

Tenha em mente que o código em translateMessage () sempre substitui o caractere em charsA pelo caractere no mesmo índice em charsB . Portanto, quando a linha 52 troca os valores, o código em translateMessage () faz o processo de descriptografia em vez do processo de criptografia.

As próximas linhas de código mostram como o programa encontra o índice a ser usado para criptografia e descriptografia.

54. # Loop através de cada símbolo na mensagem:

55. para o símbolo na mensagem:

56. if symbol.upper () em charsA:

57. # Criptografar / descriptografar o símbolo:

58. symIndex = charsA.find (symbol.upper ())

O loop for na linha 55 define a variável de símbolo para um caractere na sequência de mensagens em cada iteração através do loop. Se a forma maiúscula desse símbolo existir em charsA (lembre-se de que a chave e LETTERS possuem apenas caracteres maiúsculos), a linha 58 localizará o índice da forma maiúscula do símbolo em charsA . A variável symIndex armazena esse índice.

Nós já sabemos que o método find () nunca retornaria -1 (a -1 do método find () significa que o argumento não pôde ser encontrado na string) porque a instrução if na linha 56 garante que symbol.upper () existe em charsA . Caso contrário, a linha 58 não teria sido executada.

Em seguida, usaremos cada símbolo criptografado ou decriptografado para criar a string retornada pela função translateMessage () . Mas, como a chave e as LETRAS estão em maiúsculas, precisamos verificar se o símbolo original na mensagem estava em minúsculas e, em seguida, ajustar o símbolo decriptografado ou criptografado para minúsculas, se estivesse. Para fazer isso, você precisa aprender dois métodos de string: isupper () e islower () .

Os métodos de sequência isupper () e islower ()

Os métodos isupper () e islower () verificam se uma string está em maiúscula ou minúscula.

Mais especificamente, o método de seqüência de caracteres isupper () retorna True se ambas as condições forem atendidas:

- A string tem pelo menos uma letra maiúscula.
- A string não possui letras minúsculas.

O método de seqüência de caracteres islower () retorna True se ambas as condições forem atendidas:

- A string tem pelo menos uma letra minúscula.

- A string não possui letras maiúsculas.

Caracteres que não sejam letras na cadeia de caracteres não afetam se esses métodos retornam True ou False , embora ambos os métodos sejam avaliados como False se apenas caracteres não alfabéticos existirem na cadeia de caracteres. Digite o seguinte no shell interativo para ver como esses métodos funcionam:

```
>>> 'HELLO'.isupper ()
Verdade
❶ >>> 'HELLO WORLD 123'.isupper ()
Verdade
❷ >>> 'hello'.islower ()
Verdade
>>> '123'.isupper ()
Falso
>>> ".islower ()"
Falso
```

O exemplo em ❶ retorna True porque 'HELLO WORLD 123' tem pelo menos uma letra maiúscula e nenhuma letra minúscula. Os números nessa sequência não afetam a avaliação. Em ❷ , 'hello'.islower () retorna True porque a string ' hello ' tem pelo menos uma letra minúscula e nenhuma letra maiúscula.

Vamos voltar ao nosso código para ver como ele usa os métodos de string isupper () e islower () .

Preservando Casos com isupper ()

O programa *simpleSubCipher.py* usa os métodos de string isupper () e islower () para ajudar a garantir que os casos do texto simples sejam refletidos no texto cifrado.

```
59. if symbol.isupper ():
60.     traduzido += charsB [symIndex] .upper ()
61. else:
62.     traduzido += charsB [symIndex] .lower ()
```

A linha 59 testa se o símbolo tem uma letra maiúscula. Se isso acontecer, a linha 60 concatena a versão em maiúscula do caractere em charsB [symIndex] para traduzida . Isso resulta na versão em maiúsculas do caractere chave correspondente à entrada em maiúsculas. Se o símbolo tiver uma letra

minúscula, a linha 62 concatena a versão minúscula do caractere em charsB [symIndex] para traduzida .

Se o símbolo não for um caractere no conjunto de símbolos, como '5' ou '?', a linha 59 retornaria False e a linha 62 seria executada em vez da linha 60. O motivo é que as condições para isupper () não seriam atendidas porque essas cadeias não têm pelo menos uma letra maiúscula. Neste caso, o inferior () A chamada de método na linha 62 não teria nenhum efeito na cadeia porque não tem letras. O método lower () não altera caracteres que não sejam letras como '5' e '?' . Ele simplesmente retorna os caracteres originais sem letra.

A linha 62 no bloco else é responsável por quaisquer caracteres minúsculos e não alfabéticos em nossa sequência de símbolos .

O recuo na linha 63 indica que a instrução else está emparelhada com if symbol.upper () em charsA: statement na linha 56, então a linha 63 é executada se o símbolo não estiver em LETTERS .

63. else:

64. # Símbolo não está em LETRAS; apenas adicione:

65. traduzido + = símbolo

Se o símbolo não estiver em LETTERS , a linha 65 será executada. Isso significa que não podemos criptografar ou descriptografar o caractere em símbolo , então simplesmente o concatenamos até o final da tradução como ele é.

No final da função translateMessage () , a linha 67 retorna o valor na variável traduzida , que contém a mensagem criptografada ou descriptografada:

67. retornar traduzido

Em seguida, veremos como usar a função getRandomKey () para gerar uma chave válida para a cifra simples de substituição.

Gerando uma chave aleatória

Digitar uma string para uma chave que contenha cada letra do alfabeto pode ser difícil. Para nos ajudar com isso, a função getRandomKey () retorna uma chave válida para usar. As linhas 71 a 73 misturam aleatoriamente os caracteres na constante LETTERS .

70. def getRandomKey ():

71. key = list (LETTERS)

72. random.shuffle (chave)

```
73. return " ".join(chave)
```

NOTA

Leia “[Alocar aleatoriamente uma string](#)” na [página 123](#) para obter uma explicação de como misturar uma string usando os métodos `list()`, `random.shuffle()` e `join()`.

Para usar a função `getRandomKey()`, precisamos alterar a linha 11 de `myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'` para esta:

```
11. myKey = getRandomKey()
```

Como a linha 20 em nosso programa de codificação de substituição simples imprime a chave que está sendo usada, você poderá ver a chave retornada pela função `getRandomKey()`.

Chamando a função `main()`

As linhas 76 e 77 no final do programa chamam `main()` se `simpleSubCipher.py` estiver sendo executado como um programa em vez de ser importado como um módulo por outro programa.

```
76. if __name__ == '__main__':
```

```
77. main()
```

Isso conclui nosso estudo do programa de codificação de substituição simples.

Resumo

Neste capítulo, você aprendeu como usar o método de lista `sort()` para ordenar itens em uma lista e como comparar duas listas ordenadas para verificar se há caracteres duplicados ou ausentes de uma string. Você também aprendeu sobre os métodos de string `isupper()` e `islower()`, que verificam se um valor de string é composto de letras maiúsculas ou minúsculas. Você aprendeu sobre as funções do wrapper, que são funções que chamam outras funções, geralmente adicionando apenas pequenas alterações ou argumentos diferentes.

A simples cifra de substituição tem muitas chaves possíveis para a força bruta. Isso faz com que seja impermeável às técnicas que você usou para hackear programas de codificação anteriores. Você terá que fazer programas mais inteligentes para quebrar esse código.

No [Capítulo 17](#), você aprenderá como hackear a simples cifra de substituição. Em vez de forçar brute através de todas as chaves, você usará um algoritmo mais

inteligente e sofisticado.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Por que um ataque de força bruta não pode ser usado contra uma simples codificação de substituição, mesmo com um poderoso supercomputador?

2. O que a variável de spam contém depois de executar este código?

```
spam = [4, 6, 2, 8]
```

```
spam.sort()
```

3. O que é uma função de wrapper?

4. O que 'hello'.islower() avalia para?

5. O que 'HELLO 123'.isupper() avalia para?

6. O que '123'.islower() avalia para?

17

ENGANANDO A CIPRA DE SUBSTITUIÇÃO SIMPLES

“A criptografia é fundamentalmente um ato privado.

O ato de criptografia, na verdade, remove informações do domínio público. Até as leis contra a criptografia alcançam apenas as fronteiras de uma nação e o braço de sua violência”.

—Eric Hughes, “Um Manifesto de Cypherpunk” (1993)



No [Capítulo 16](#), você aprendeu que a simples cifra de substituição é impossível de ser quebrada usando força bruta, porque ela tem muitas chaves possíveis. Para hackar a simples cifra de substituição, precisamos criar um programa mais

sofisticado que use valores de dicionário para mapear as possíveis letras de deciptação de um texto cifrado. Neste capítulo, escreveremos um programa desse tipo para restringir a lista de possíveis saídas de descriptografia à lista correta.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Padrões de palavras, candidatos, cartas de descriptografia em potencial e mapeamentos de cifras
- Expressões regulares
- O método regex `sub()`

Usando padrões do Word para descriptografar

Em ataques de força bruta, tentamos cada chave possível para verificar se ela pode descriptografar o texto cifrado. Se a chave estiver correta, a decodificação resultará em inglês legível. Mas, analisando primeiro o texto cifrado, podemos reduzir o número de chaves possíveis para tentar e talvez até encontrar uma chave completa ou parcial.

Vamos supor que o texto original seja composto principalmente de palavras em um arquivo de dicionário em inglês, como o que usamos no [Capítulo 11](#). Embora um texto cifrado não seja feito de palavras inglesas reais, ele ainda conterá grupos de letras divididos por espaços, assim como palavras em frases regulares. Vamos chamar essas *palavras-chave* neste livro. Em uma cifra de substituição, cada letra do alfabeto tem exatamente uma letra de criptografia correspondente única. Vamos chamar as letras nos *codificadores* de texto cifrado. Como cada letra de texto simples pode criptografar apenas uma cifra, e não estamos criptografando espaços nessa versão da cifra, o texto simples e o texto cifrado compartilharão os mesmos *padrões de palavras*.

Por exemplo, se tivéssemos o texto simples MISSISSIPPI SPILL, o texto cifrado correspondente poderia ser RJBBIJJXXJ BXJHH. O número de letras na primeira palavra do texto simples e a primeira palavra cifrada são as mesmas. O mesmo é verdadeiro para a segunda palavra de texto simples e a segunda palavra cifrada. O texto simples e o texto cifrado compartilham o mesmo padrão de letras e espaços. Observe também que as letras que se repetem no texto original repetem o mesmo número de vezes e nos mesmos lugares que o texto cifrado.

Poderíamos, portanto, assumir que uma palavra cifrada corresponde a uma

palavra no arquivo do dicionário em inglês e que seus padrões de palavras seriam correspondentes. Então, se pudermos encontrar em qual palavra no dicionário a palavra cifrada é decodificada, podemos descobrir a decriptografia de cada cifra da palavra. E, se descobrirmos decifrações da cifra usando essa técnica, poderemos decifrar a mensagem inteira.

Encontrando Padrões do Word

Vamos examinar o padrão de palavras da palavra-chave HGHHU. Você pode ver que a palavra-chave tem certas características, que a palavra de texto simples original deve compartilhar. Ambas as palavras devem ter o seguinte em comum.

1. Eles devem ter cinco letras de comprimento.
2. A primeira, terceira e quarta letras devem ser as mesmas.
3. Eles devem ter exatamente três letras diferentes; a primeira, segunda e quinta letras devem ser todas diferentes.

Vamos pensar em palavras na língua inglesa que se encaixem nesse padrão. *Puppy* é uma dessas palavras, que tem cinco letras de comprimento (P, U, P, P, Y) e usa três letras diferentes (P, U, Y) organizadas nesse mesmo padrão (P para a primeira, terceira e quarta letra); U para a segunda letra; e Y para a quinta letra). *Mamãe*, *bobby*, *calmaria* e *babá* também se encaixam no padrão. Essas palavras, juntamente com qualquer outra palavra no arquivo de dicionário em inglês que corresponda aos critérios, são todas possíveis descrições de HGHHU.

Para representar um padrão de palavras de uma maneira que o programa possa entender, faremos cada padrão em um conjunto de números separados por pontos que indicam o padrão das letras.

Criar padrões de palavras é fácil: a primeira letra obtém o número 0 e a primeira ocorrência de cada letra diferente obtém o próximo número. Por exemplo, o padrão de palavras para *gato* é 0.1.2 e o padrão de palavras para *classificação* é 0.1.2.3.3.4.5.4.0.2.6.4.7.8.

Em cifras de substituição simples, não importa qual chave é usada para criptografar, uma palavra de texto simples e sua palavra cifrada *sempre* têm o mesmo padrão de palavras. O padrão de palavras para a palavra-chave HGHHU é 0.1.0.0.2, o que significa que o padrão de palavras do texto simples correspondente a HGHHU também é 0.1.0.0.2.

Encontrando Cartas Potenciais de Decodificação

Para descriptografar o HGHHU, precisamos encontrar todas as palavras em um arquivo de dicionário em inglês cujo padrão de palavras também seja 0.1.0.0.2. Neste livro, vamos chamar as palavras de texto simples que têm o mesmo padrão de palavras da palavra cifrada que os *candidatos* a essa palavra-chave. Aqui está uma lista de candidatos para o HGHHU:

- cachorro
- mamãe
- bobby
- calmarias
- babá

Usando padrões de palavras, podemos adivinhar para quais criptografias de texto simples as deciframentas, que chamaremos de *possíveis letras de descriptografia* da cifra. Para quebrar uma mensagem criptografada com a simples cifra de substituição, precisamos encontrar todas as letras de descriptografia potenciais de cada palavra na mensagem e determinar as letras de descriptografia reais através do processo de eliminação. [A Tabela 17-1](#) lista as possíveis letras de descriptografia para o HGHHU.

Tabela 17-1: Cartas de Decodificação Potencial dos Cifradores em HGHHU

Cifradores **H G H H você**

**Letras de decodificação em
potencial** P você P P Y

 M O M M Y

 B O B B Y

 eu você eu eu S

 N UMA N N Y

A seguir, um *mapeamento de cifra* criado usando a [Tabela 17-1](#) :

1. H tem as letras de descriptografia em potencial P, M, B, L e N.
2. G tem as letras de descriptografia em potencial U, O e A.

3. U tem as letras potenciais de decodificação Y e S.
4. Todos os outros codificadores além de H, G e U não possuem letras de descriptografia potenciais neste exemplo.

Um mapeamento de cifra mostra todas as letras do alfabeto e suas possíveis cartas de descriptografia. À medida que começamos a coletar mensagens criptografadas, encontraremos possíveis letras de descriptografia para cada letra do alfabeto, mas como apenas os criptogramas H, G e U faziam parte de nosso exemplo de texto cifrado, não temos as possíveis letras de descriptografia de outros codificadores.

Observe também que U tem apenas duas possíveis letras de decodificação (Y e S) porque há sobreposições entre os candidatos, muitos dos quais terminam na letra Y. *Quanto mais sobreposições houver, menos letras de descriptografia potenciais existirão e mais facilmente será para descobrir o que essa cifra de decifra descriptografa.*

Para representar a [Tabela 17-1](#) no código Python, usaremos um valor de dicionário para representar os mapeamentos de calendários da seguinte maneira (os pares de valores-chave para 'H' , 'G' e 'U' estão em negrito):

```
{'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': ['U ','O ','A '],  
'H': ['P', 'M', 'B', 'L', 'N'] , 'eu': [], 'J': [], 'K': [], 'L': [], 'M': [],  
'N': [], 'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': ['Y',  
'S'] , 'V': [], 'W': [], 'X': [], 'Y': [], 'Z': []}
```

Este dicionário tem 26 pares de valores-chave, uma chave para cada letra do alfabeto e uma lista de possíveis cartas de descriptografia para cada letra. Ele mostra letras de descriptografia em potencial para as teclas 'H' , 'G' e 'U' . As outras chaves têm listas vazias, [] , para valores, porque elas não têm letras de descriptografia potenciais até o momento.

Se pudermos reduzir o número de possíveis cartas de descriptografia de uma cifra a apenas uma letra cruzando referências de outras palavras criptografadas, podemos encontrar o que essa cifra descriptografa. Mesmo que não consigamos resolver todos os 26 cifradores, poderemos hackear a maioria dos mapeamentos de cifras para descriptografar a maior parte do texto cifrado.

Agora que abordamos alguns dos conceitos básicos e terminologia que usaremos neste capítulo, vamos dar uma olhada nos passos envolvidos no processo de hacking.

Visão geral do processo de hacking

Hackear a simples cifra de substituição é bem fácil usando padrões de palavras. Podemos resumir os principais passos do processo de hacking da seguinte forma:

1. Encontre o padrão de palavras para cada palavra cifrada no texto cifrado.
2. Encontre os candidatos da palavra em inglês que cada palavra-chave poderia decifrar.
3. Crie um dicionário mostrando letras de descriptografia em potencial para cada cipherletter para atuar como o mapeamento de cifras para cada palavra cifrada.
4. Combine os mapeamentos do cipherletter em um único mapeamento, que chamaremos de *mapeamento com interseção*.
5. Remova qualquer criptografia resolvida do mapeamento combinado.
6. Descriptografar o texto cifrado com os criptografados resolvidos.

Quanto mais palavras criptografadas em um texto cifrado, mais provável é que os mapeamentos se sobreponham uns aos outros e menos letras de descriptografia em potencial para cada cifra. Isso significa que, na cifra de substituição simples, *quanto mais longa a mensagem de texto cifrado, mais fácil será hackear*.

Antes de mergulhar no código fonte, vamos ver como podemos facilitar as duas primeiras etapas do processo de hacking. Usaremos o arquivo de dicionário que usamos no [Capítulo 11](#) e um módulo chamado *wordPatterns.py* para obter o padrão de palavra para cada palavra no arquivo do dicionário e classificá-los em uma lista.

Os módulos padrão do Word

Para calcular padrões de palavras para cada palavra no arquivo de *dicionário dictionary.txt*, faça download do *makeWordPatterns.py* em <https://www.nostarch.com/crackingcodes/>. Certifique-se de que este programa e o *dicionário.txt* estejam na pasta onde você salvará o programa *simpleSubHacker.py* deste capítulo.

O programa *makeWordPatterns.py* tem uma função *getWordPattern()* que pega uma string (como 'filhote') e retorna seu padrão de palavras (como '0.1.0.0.2').

Quando você executa o `makeWordPatterns.py`, ele deve criar o módulo Python `wordPatterns.py`. O módulo contém uma única declaração de atribuição de variável, conforme mostrado aqui, e tem mais de 43.000 linhas de extensão:

```
allPatterns = {'0.0.1': ['EEL'],
'0.0.1.2': ['EELS', 'OOZE'],
'0.0.1.2.0': ['EERIE'],
«0.0.1.2.3»: ['AARON', 'LLOYD', 'OOZED'],
- recorte -
```

A variável `allPatterns` contém um valor de dicionário com as strings de padrão de palavra como chaves e uma lista de palavras em inglês que correspondem ao padrão como seus valores. Por exemplo, para encontrar todas as palavras em inglês com o padrão `0.1.2.1.3.4.5.4.6.7.8`, insira o seguinte no shell interativo:

```
>>> import wordPatterns
>>> wordPatterns.allPatterns ['0.1.2.1.3.4.5.4.6.7.8']
['BENEFICIÁRIO', 'HOMOGENEIDADE', 'MOTOCICLETAS']
```

No dicionário `allPatterns`, a chave '`0.1.2.1.3.4.5.4.6.7.8`' tem o valor da lista `['BENEFICIARY', 'HOMOGENEITY', 'MOTORCYCLES']`, que contém três palavras em inglês com este padrão de palavras específico.

Agora vamos importar o módulo `wordPatterns.py` para começar a construir o programa simples de substituição de hackers!

NOTA

Se você receber uma mensagem de erro `ModuleNotFoundError` ao importar `wordPatterns` no shell interativo, insira o seguinte no shell interativo primeiro:

```
>>> import sys
>>> sys.path.append (' nome_da_pasta ')
```

Substitua `name_of_folder` pela localização em que `wordPatterns.py` é salvo. Isso informa ao shell interativo para procurar por módulos na pasta que você especificar.

Código fonte do programa de substituição de substituição simples

Abra uma janela do editor de arquivos selecionando **Arquivo ▶ Novo arquivo**. Digite o seguinte código no editor de arquivos e salve-o como `simpleSubHacker.py`. Certifique-se de colocar os arquivos `pyperclip.py`, `simpleSubCipher.py` e `wordPatterns.py` no mesmo diretório que

simpleSubHacker.py . Pressione F5 para executar o programa.

*simpleSub
Hacker.py*

```
1. # Cifra de Substituição Simples
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar os, re, copiar, pyperclip, simpleSubCipher, wordPatterns,
makeWordPatterns
5
6
7
8
9
10. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. nonLettersOrSpacePattern = re.compile ('[^ A-Z \s]')
12
13. def main ():
14.     message = «Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr
sxrjsxwjr, ia esmm rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa
sr pqaceiamnsxu, ia esmm caytra jp famsaqa sj. Sy, px jia pjiac
ilxo, eu sr pyyacao rpnajisxu eiswi lyppcor l calrpx ypc lwjsxu sx
lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax px jia
rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.
-Facjclxo Ctrramm '
15
16. # Determine as possíveis traduções válidas de texto cifrado:
17. print ('Hacking ...')
18. letterMapping = hackSimpleSub (mensagem)
19
20. # Exibe os resultados para o usuário:
21. print ('Mapeamento:')
22. print (letterMapping)
23. print ()
24. print ('Texto cifrado original:')
25. imprimir (mensagem)
26. print ()
```

```
27. print ('Copiando a mensagem hackeada para a área de transferência:')
28. hackedMessage = decryptWithCipherletterMapping (mensagem,
letterMapping)
29. pyperclip.copy (hackedMessage)
30. print (hackedMessage)
31
32
33. def getBlankCipherletterMapping ():
34. # Retorna um valor de dicionário que é um mapeamento de cifra em branco:
35. return {'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': []
'H': [], 'eu': [], 'J': [], 'K': [], 'L': [], 'M': [], 'N': [],
'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': [],
'V W X Y Z': []}
36
37
38. def addLettersToMapping (letterMapping, cipherword, candidate):
39. # O parâmetro letterMapping usa um valor de dicionário que
40. # armazena um mapeamento cipherletter, que é copiado pela função.
41. # O parâmetro da palavra-chave é um valor de string da palavra do texto
cifrado.
42. # O parâmetro candidato é uma possível palavra em inglês que o
43. # cipherword poderia decifrar para.
44
45. # Esta função adiciona as letras no candidato como potencial
46. # letras de descriptografia para os cífractores no cipherletter
47. # mapeamento.
48
49.
50. para i no intervalo (len (cipherword)):
51. se candidato [i] não em letterMapping [cipherword [i]]:
52. letterMapping [cipherword [i]]. Append (candidato [i])
53
54
55
56. def intersectMappings (mapA, mapB):
57. # Para interceptar dois mapas, crie um mapa em branco e adicione apenas
58. # possíveis cartas de descriptografia, se existirem em ambos os mapas:
```

```
59. intersectedMapping = getBlankCipherletterMapping ()
60. para cartas em LETRAS:
61
62. # Uma lista vazia significa "qualquer letra é possível". Neste caso apenas
63. # copie o outro mapa inteiramente:
64. se mapA [carta] == []:
65.     intersectedMapping [letter] = copy.deepcopy (mapB [letra])
66. elif mapB [carta] == []:
67.     intersectedMapping [letter] = copy.deepcopy (mapA [letra])
68. else:
69.     # Se uma letra no mapa [letra] existe no mapa [letra],
70.     # adicione essa letra ao intersectedMapping [letter]:
71.     para mappedLetter no mapA [letter]:
72.         se mappedLetter no mapB [letter]:
73.             intersectedMapping [carta] .append (mappedLetter)
74
75. retornar intersectedMapping
76
77
78. def removeSolvedLettersFromMapping (letterMapping):
79.     # Cipherletters no mapeamento que mapeiam para apenas uma letra são
80.     # "resolvido" e pode ser removido das outras letras.
81.     # Por exemplo, se 'A' mapear para possíveis letras ['M', 'N'] e 'B'
82.     # mapeia para ['N'], então sabemos que 'B' deve mapear para 'N', então
83.     # podemos
84.     # remove 'N' da lista do que 'A' pode mapear. Então 'A' então mapeia
85.     # para ['M']. Note que agora que 'A' mapeia para apenas uma letra, podemos
86.     # remove 'M' da lista de letras para todas as outras letras.
87.     # (É por isso que existe um loop que continua reduzindo o mapa.)
88.     loopAgain = Verdadeiro
89.     while loopAgain:
90.         # Primeiro, suponha que não vamos repetir novamente:
91.         loopAgain = Falso
92
93.         # solvedLetters será uma lista de letras maiúsculas que têm um
94.         # e apenas um mapeamento possível em letterMapping:
```

```
95. solvedLetters = []
96. para cipherletter em LETTERS:
97. if len (letterMapping [cipherletter]) == 1:
98. solvedLetters.append (letterMapping [cipherletter] [0])
99
100. # Se uma carta for resolvida, então não pode ser um potencial
101. # carta de decriptação para uma letra diferente de texto cifrado, então nós
102. # deve removê-lo dessas outras listas:
103. para caligrafia em LETRAS:
104. para s emListas solucionadas:
105. if len (letterMapping [cipherletter])! = 1 e s em
letterMapping [cipherletter]:
106. letterMapping [cipherletter] .remove (s)
107. if len (letterMapping [cipherletter]) == 1:
108. # Uma nova carta está agora resolvida, então dê um loop novamente:
109. loopAgain = True
110. carta de retorno
111
112
113. def hackSimpleSub (mensagem):
114. intersectedMap = getBlankCipherletterMapping ()
115. cipherwordList = nonLettersOrSpacePattern.sub (",
message.upper ()). split ()
116. para cipherword in cipherwordList:
117. # Obter um novo mapeamento de cifra para cada palavra de texto cifrado:
118. candidateMap = getBlankCipherletterMapping ()
119
120. wordPattern = makeWordPatterns.getWordPattern (palavra-chave)
121. if wordPattern not em wordPatterns.allPatterns:
122. continue # Esta palavra não estava em nosso dicionário, então continue.
123
124. # Adicione as letras de cada candidato ao mapeamento:
125. para candidato em wordPatterns.allPatterns [wordPattern]:
126. addLettersToMapping (candidateMap, cipherword, candidate)
127
128. # Intersecte o novo mapeamento com o mapeamento interseccional
existente:
```

```
129. intersectedMap = intersectMappings (intersectedMap, candidateMap)
130
131. # Remova quaisquer letras resolvidas das outras listas:
132. return removeSolvedLettersFromMapping (intersectedMap)
133
134
135. def decryptWithCipherletterMapping (texto cifrado, letterMapping):
136. # Retorna uma string do texto cifrado descriptografado com o mapeamento
de letras,
137. # com quaisquer letras decifradas ambíguas substituídas por um sublinhado.
138
139. # Primeiro crie uma subchave simples a partir do mapeamento
letterMapping:
140. key = ['x'] * len (LETRAS)
141. para caligrafia em LETRAS:
142. if len (letterMapping [cipherletter]) == 1:
143. # Se houver apenas uma letra, adicione-a à chave:
144. keyIndex = LETTERS.find (letterMapping [cipherletter] [0])
145. key [keyIndex] = caligrafia
146. else:
147. ciphertext = ciphertext.replace (cipherletter.lower (), '_')
148. ciphertext = ciphertext.replace (cipherletter.upper (), '_')
149. key = " .join (chave)
150
151. # Com a chave que criamos, descriptografar o texto cifrado:
152. return simpleSubCipher.decryptMessage (chave, texto cifrado)
153
154.
155. if __name__ == '__main__':
156. main ()
```

Execução de amostra do programa de substituição de substituição simples

Quando você executa este programa, ele tenta hackear o texto cifrado na variável de mensagem . Sua saída deve ficar assim:

Hacking ...

Mapeamento:

```
{'A': ['E'], 'B': ['Y', 'P', 'B'], 'C': ['R'], 'D': [], 'E': ['W'], 'F': ['B', 'P'], 'G': ['B', 'Q', 'X', 'P', 'Y'], 'H': ['P', 'Y', 'K', 'X', 'B'], 'I': ['H'], 'J': ['T'], 'K': [], 'L': ['A'], 'M': ['L'], 'N': ['M'], 'O': ['D'], 'P': ['O'], 'Q': ['V'], 'R': ['S'], 'S': ['eu'], 'T': ['U'], 'U': ['G'], 'V': [], 'W': ['C'], 'X': ['N'], 'Y': ['F'], 'Z': ['Z']}
```

Texto cifrado original:

Syl nlx sr pyyacao l ylwj eiswi upar lulsxrj isr srxjsxwjr, ia esmm
rwctjsxsza sj wmptramh, lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia esmm
caytra

jp famsaqa sj. Sy, px jia pjiac ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor
l calppx ypc lwjsxu sx lwwpcolxwa jp isr srxjsxwjr, ia esmm lwwabj sj aqax
px jia rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.

-Facjclxo Ctrramm

Copiando mensagem hackeada para a área de transferência:

Se um homem é oferecido um fato que vai contra seus instintos, ele vai
escrutiná-lo closel_, e a menos que a evidência é esmagadora, ele irá recusar
para acreditar. Se, por outro lado, ele é oferecido algo que oferece
uma razão para agir de acordo com seus instintos, ele aceitará até mesmo
na menor evidência. A origem de m_ths é obtida neste wa_.

-ertrand Russell

Agora vamos explorar o código fonte em detalhes.

Configurando Módulos e Constantes

Vamos ver as primeiras linhas do programa simples de substituição de hackers.
A linha 4 importa sete módulos diferentes, mais do que qualquer outro programa
até agora. A variável global LETTERS na linha 10 armazena o conjunto de
símbolos, que consiste nas letras maiúsculas do alfabeto.

1. # Cifra de Substituição Simples
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar os, re, copiar, pyperclip, simpleSubCipher, wordPatterns,
makeWordPatterns
- recorte -
10. LETRAS = 'ABCDEFGHIJKLMNPQRSTUVWXYZ'

O re módulo é o módulo de expressão regular, que permite a manipulação sofisticada de strings usando expressões regulares. Vamos ver como as expressões regulares funcionam.

Encontrando Caracteres com Expressões Regulares

Expressões regulares são sequências que definem um padrão específico que corresponde a determinadas cadeias. Por exemplo, a string '[^ AZ \ s]' na linha 11 é uma expressão regular que diz ao Python para localizar qualquer caractere que não seja uma letra maiúscula de A a Z ou um caractere de espaço em branco (como espaço, tabulação ou caractere de nova linha).

```
11. nonLettersOrSpacePattern = re.compile ('[^ AZ \ s]')
```

A função `re.compile ()` cria um objeto de padrão de expressão regular (abreviado como *objeto regex* ou *objeto de padrão*) que o `re module` pode usar. Usaremos esse objeto para remover quaisquer caracteres que não sejam letras do texto cifrado em “ [A função `hackSimpleSub \(\)`](#) ” na [página 241](#) .

Você pode executar muitas manipulações sofisticadas de strings com expressões regulares. Para saber mais sobre expressões regulares, acesse <https://www.nostarch.com/crackingcodes/> .

Configurando a função main ()

Como com os programas de hackers anteriores neste livro, a função `main ()` armazena o texto cifrado na variável `message` e a linha 18 passa essa variável para a função `hackSimpleSub ()` :

```
13. def main ():
```

```
14.     message = «Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr  
sxrjsxwjr, ia esmm rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa  
sr pqaceiamnsxu, ia esmm caytra jp famsaqa sj. Sy, px jia pjiac  
ilxo, eu sr pyyacao rpnajisxu eiswi lyypcor l calrpx ypc lwjsxu sx  
lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax px jia  
rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.  
-Facjclxo Ctrramm '
```

```
15
```

```
16. # Determine as possíveis traduções válidas de texto cifrado:
```

```
17. print ('Hacking ...')
```

```
18. letterMapping = hackSimpleSub (mensagem)
```

Em vez de retornar a mensagem descriptografada ou None, se não conseguir descriptografá-la, hackSimpleSub () retorna um mapeamento cruzado de cifra com as letras descriptografadas removidas. (Veremos como criar um mapeamento intersectado em “ [Intersecting Two Mappings](#) ” na [página 234.](#)) Esse mapeamento intersectado de cifra é então passado para decryptWithCipherletterMapping () para descriptografar o texto cifrado armazenado na mensagem em um formato legível, já que você veja em mais detalhes em “ [Descriptografar a Mensagem](#) ” na [página 243](#) .

O mapeamento de cifra armazenado em letterMapping é um valor de dicionário que possui 26 cadeias maiúsculas de uma letra como chaves que representam os codificadores. Ele também lista as letras maiúsculas de possíveis letras de descriptografia para cada cipherletter como os valores do dicionário. Quando cada cipherletter tem apenas uma letra potencial de descriptografia associada a ela, temos um mapeamento totalmente resolvido e podemos descriptografar qualquer texto cifrado usando a mesma cifra e chave.

Cada mapeamento cifra gerado depende do texto cifrado usado. Em alguns casos, teremos apenas um mapeamento parcialmente resolvido em que alguns criptografadores não têm descriptografias em potencial e outros criptografadores têm várias descriptografias em potencial. Textos cifrados mais curtos que não contenham todas as letras do alfabeto têm maior probabilidade de resultar em mapeamentos incompletos.

Exibindo Resultados de Hacking para o Usuário

O programa chama a função print () para exibir letterMapping , a mensagem original e a mensagem descriptografada na tela:

```
20. # Exibe os resultados para o usuário:  
21. print ('Mapeamento:')  
22. print (letterMapping)  
23. print ()  
24. print ('Texto cifrado original:')  
25. imprimir (mensagem)  
26. print ()  
27. print ('Copiando a mensagem hackeada para a área de transferência:')  
28. hackedMessage = decryptWithCipherletterMapping (mensagem,  
letterMapping)  
29. pyperclip.copy (hackedMessage)
```

30. print (hackedMessage)

A linha 28 armazena a mensagem descriptografada na variável hackedMessage , que é copiada para a área de transferência e impressa na tela para que o usuário possa compará-la à mensagem original. Usamos decryptWithCipherletterMapping () para encontrar a mensagem descriptografada, que é definida posteriormente no programa.

Em seguida, vamos ver todas as funções que criam os mapeamentos da cifra.

Criando um Mapeamento de Cifras

O programa precisa de um mapeamento de cifras para cada palavra cifrada no texto cifrado. Para criar um mapeamento completo, precisaremos de várias funções auxiliares. Uma dessas funções auxiliares irá configurar um novo mapeamento de cifra, para que possamos chamá-lo para cada cifra.

Outra função terá uma palavra cifrada, seu atual mapeamento de letras e uma palavra de descriptografia candidata para encontrar todas as palavras de descriptografia candidatas. Vamos chamar essa função para cada palavra cifrada e cada candidato. A função adicionará todas as letras de descriptografia da palavra candidata ao mapeamento de letras da palavra cifrada e retornará o mapeamento de letras.

Quando temos mapeamentos de letras para várias palavras do texto cifrado, usaremos uma função para mesclá-las. Em seguida, usaremos uma função auxiliar final para resolver a descriptografia de tantos codificadores quanto possível, combinando uma letra de descriptografia com cada cifra. Como observado, nem sempre seremos capazes de resolver todos os criptografadores, mas você descobrirá como lidar com esse problema em “ [Descriptografando a Mensagem](#) ” na [página 243](#) .

Criando um mapeamento em branco

Primeiro, precisamos criar um mapeamento de cifra em branco.

33. def getBlankCipherletterMapping ():

34. # Retorna um valor de dicionário que é um mapeamento de cifra em branco:

35. return {'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': []
'H': [], 'eu': [], 'J': [], 'K': [], 'L': [], 'M': [], 'N': [],
'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': [],
'V W X Y Z': []}

Quando chamada, a função `getBlankCipherletterMapping()` retorna um dicionário com as chaves configuradas para strings de um caractere das 26 letras do alfabeto.

Adicionando Letras a um Mapeamento

Para adicionar letras a um mapeamento, definimos a função `addLettersToMapping()` na linha 38.

38. def `addLettersToMapping(letterMapping, cipherword, candidate):`

Essa função usa três parâmetros: um mapeamento de cifra (`letterMapping`), uma palavra cifrada para mapear (`palavra cifrada`) e uma palavra de descriptografia candidata para a qual a palavra cifrada poderia descriptografar (`candidato`). A função mapeia todas as letras em candidatos para a cifra na posição de índice correspondente na palavra cifrada e adiciona essa carta ao `letterMapping`, se ainda não estiver lá.

Por exemplo, se 'PUPPY' for o candidato para a palavra cifrada 'HGHHU' , a função `addLettersToMapping()` adicionará o valor 'P' à chave 'H' em `letterMapping` . Em seguida, a função passa para a próxima letra e acrescenta 'U' ao valor da lista emparelhado com a tecla 'G' , e assim por diante.

Se a letra já estiver na lista de possíveis letras de descriptografia, então `addLettersToMapping()` não adicionará essa letra à lista novamente. Por exemplo, em 'PUPPY', ele ignoraria a adição de 'P' à tecla 'H' das próximas duas instâncias de 'P', porque já está lá. Por fim, a função altera o valor para a tecla 'U', de modo que tem 'Y' em sua lista de possíveis letras de descriptografia.

O código em `addLettersToMapping()` assume que `len(cipherword)` é o mesmo que `len(candidate)` porque devemos passar apenas uma palavra cifrada e um par `candidato` com padrões de palavras correspondentes.

Em seguida, o programa itera sobre cada índice na string em `cipherword` para verificar se uma letra já foi adicionada à lista de possíveis cartas de descriptografia:

50. para `i` no intervalo (`len(cipherword)`):

51. se `candidato[i]` não em `letterMapping[cipherword[i]]`:

52. `letterMapping[cipherword[i]].Append(candidate[i])`

Usaremos a variável `i` para iterar através de cada letra da palavra cifrada e sua correspondente letra de descriptografia em potencial por meio da indexação.

Podemos fazer isso porque a possível letra de deciptação a ser adicionada é a candidata [i] para a palavra-código cifrada [i] . Por exemplo, se a palavra cifrada era 'HGHHU' e o candidato era 'PUPPY' , eu começaria no índice 0 , e usariamos cipherword [0] e candidate [0] para acessar as primeiras letras de cada string. Então a execução seguiria para a declaração if na linha 51.

A instrução if verifica se a possível letra de descriptografia, candidata [i] , ainda não está na lista de possíveis letras de descriptografia da cipherletter e não o adiciona se já estiver na lista. Ele faz isso acessando a cifra no mapeamento com letterMapping [cipherword [i]] , porque a palavra cifrada [i] é a chave no letterMapping que precisa ser acessada. Essa verificação evita letras duplicadas na lista de possíveis cartas de descriptografia.

Por exemplo, o primeiro 'P' em 'PUPPY' pode ser adicionado ao letterMapping na primeira iteração do loop, mas quando i é igual a 2 na terceira iteração, o 'P' do candidato [2] não seria ser adicionado ao mapeamento porque já foi adicionado na primeira iteração.

Se a letra de descriptografia em potencial não estiver no mapeamento, a linha 52 adicionará a nova letra, candidata [i] , à lista de possíveis letras de descriptografia no mapeamento da carta cifrada em letterMapping [cipherword [i]] .

Lembre-se de que, como o Python transmite uma cópia da referência a um dicionário passado para o parâmetro, em vez de uma cópia do próprio dicionário, qualquer alteração feita no letterMap nessa função também será executada fora da função addLettersToMapping () . Isso ocorre porque ambas as cópias da referência ainda se referem ao mesmo dicionário passado para o parâmetro letterMapping na chamada para addLettersToMapping () na linha 126.

Depois de percorrer todos os índices em cipherword , a função é feita adicionando letras ao mapeamento na variável letterMapping . Agora vamos ver como o programa compara esse mapeamento com o de outras palavras-chave para verificar sobreposições.

Interseção de dois mapeamentos

A função hackSimpleSub () usa a função intersectMappings () para obter dois mapeamentos de cifra enviados como seus parâmetros mapA e mapB e retornar um mapeamento mesclado de mapA e mapB . A função intersectMappings () instrui o programa a combinar mapA e mapB , criar um mapa em branco e

adicionar as possíveis cartas de descriptografia ao mapa em branco somente se elas existirem nos *dois* mapas para evitar duplicatas.

56. def intersectMappings (mapA, mapB):
57. # Para interceptar dois mapas, crie um mapa em branco e adicione apenas
58. # possíveis cartas de descriptografia, se existirem em ambos os mapas:
59. intersectedMapping = getBlankCipherletterMapping ()

Primeiro, a linha 59 cria um mapeamento cipherletter para armazenar o mapeamento mesclado chamando getBlankCipherletterMapping () e armazenando o valor retornado na variável intersectedMapping .

O loop for na linha 60 faz um loop através das letras maiúsculas na variável constante LETTERS e usa a variável letter como as chaves dos dicionários mapA e mapB :

60. para cartas em LETRAS:

61

62. # Uma lista vazia significa "qualquer letra é possível". Neste caso apenas
63. # copie o outro mapa inteiramente:

64. se mapA [carta] == []:

65. intersectedMapping [letter] = copy.deepcopy (mapB [letra])

66. elif mapB [carta] == []:

67. intersectedMapping [letter] = copy.deepcopy (mapA [letra])

A linha 64 verifica se a lista de possíveis letras de descriptografia do mapa A está em branco. Uma lista em branco significa que essa cifra pode potencialmente decifrar *qualquer* letra. Nesse caso, o mapeamento da cifra interseccional apenas copia a lista de cartas de descriptografia em potencial do *outro* mapeamento. Por exemplo, se a lista de possíveis letras de descriptografia no mapa A estiver em branco, a linha 65 definirá a lista do mapeamento com intersecção como uma cópia da lista no mapB e vice-versa na linha 67. Observe que, se as listas de ambos os mapeamentos estiverem em branco, a condição na linha 64 ainda é True e, em seguida, a linha 65 simplesmente copia a lista em branco no mapB para o mapeamento interseccional.

O resto do bloco na linha 68 lida com o caso em que nenhum mapa nem mapB está em branco:

68. else:
69. # Se uma letra em mapA [letter] existir no mapB [letter],

```
70. # adicione essa letra a intersectedMapping [letter]:  
71. para mappedLetter em mapA [letter]:  
72. if mappedLetter em mapB [letra]:  
73. intersectedMapping [letra] .append (mappedLetter)  
74  
75. retornar intersectedMapping
```

Quando os mapas não estão em branco, a linha 71 percorre as cadeias de letras maiúsculas na lista em mapA [letter] . A linha 72 verifica se a letra maiúscula em mapA [letter] também existe na lista de strings de letras maiúsculas no mapB [letter] . Se isso acontecer, então intersectedMapping [letra] na linha 73 adiciona essa letra comum à lista de possíveis letras de descriptografia.

Após o para loop que começou na linha 60 foi concluída, o mapeamento cipherletter em intersectedMapping só deve ter os potenciais letras de descriptografia que existem nas listas de potenciais letras de decodificação de ambos MAPA e mapB . A linha 75 retorna este mapeamento de cifra completamente cruzado. Em seguida, vamos ver um exemplo de saída de um mapeamento cruzado.

Como funciona o auxiliar de mapeamento de letras

Agora que definimos as funções auxiliares de mapeamento de letras, vamos tentar usá-las no shell interativo para entender melhor como elas funcionam juntas. Vamos criar um mapa cipherletter cruzado para o texto cifrado 'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC' , que contém apenas três palavras-chave. Faremos isso criando um mapeamento para cada palavra e combinando os mapeamentos.

Importe o *simpleSubHacker.py* para o shell interativo:

```
>>> import simpleSubHacker
```

Em seguida, chamamos getBlankCipherletterMapping () para criar um mapeamento de letras em branco e armazenar esse mapeamento em uma variável chamada letterMapping1 :

```
>>> letterMapping1 = simpleSubHacker.getBlankCipherletterMapping ()  
>>> letterMapping1  
{'A': [], 'C': [], 'B': [], 'E': [], 'D': [], 'G': [], 'F': [], 'eu': [],  
'H': [], 'K': [], 'J': [], 'M': [], 'L': [], 'O': [], 'N': [], 'Q': [],  
'P': [], 'S': [], 'R': [], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [],
```

```
' X ': [],' Z ': []}
```

Vamos começar a hackear a primeira palavra-chave, 'OLQIHXIRCKGNZ' .

Primeiro, precisamos obter o padrão de palavras para essa palavra cifrada chamando a função `getWordPattern ()` do módulo `makeWordPattern` , como mostrado aqui:

```
>>> import makeWordPatterns  
>>> makeWordPatterns.getWordPattern ('OLQIHXIRCKGNZ')  
0.1.2.3.4.5.3.6.7.8.9.10.11
```

Para descobrir quais palavras em inglês no dicionário têm o padrão de palavras 0.1.2.3.4.5.3.6.7.8.9.10.11 (ou seja, para descobrir os candidatos para a palavra-chave 'OLQIHXIRCKGNZ') , importamos o módulo `wordPatterns` e procuramos acima deste padrão:

```
>>> import wordPatterns  
>>> candidates = wordPatterns.allPatterns ['0.1.2.3.4.5.3.6.7.8.9.10.11']  
>>> candidatos  
['UNCOMFORTABLE', 'UNCOMFORTABLY']
```

Duas palavras em inglês correspondem ao padrão de palavra para 'OLQIHXIRCKGNZ' ; portanto, as únicas duas palavras que a primeira palavra cifrativa poderia decodificar são 'DESCONFORTÁVEIS' e 'INCRIVELMENTE' . Essas palavras são nossos candidatos, portanto, vamos armazená-los na variável `candidates` (não confundir com o parâmetro `candidato` na função `addLettersToMapping ()`) como uma lista.

Em seguida, precisamos mapear suas letras para as letras da palavra cifrada usando `addLettersToMapping ()` . Primeiro, vamos mapear 'UNCOMFORTABLE' acessando o primeiro membro da lista de candidatos , assim:

```
>>> letterMapping1 = simpleSubHacker.addLettersToMapping (letterMapping1,  
'OLQIHXIRCKGNZ', candidatos [0])  
>>> letterMapping1  
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [], 'eu': [  
'O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'], 'N': [  
'L'], 'Q': ['C'], 'P': [], 'S': [], 'R': ['R'], 'U': [], 'T': [], 'W': [],  
'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

A partir do valor `letterMapping1` , você pode ver que as letras em

'OLQIHXIRCKGNZ' mapeiam as letras em 'UNCOMFORTABLE' : 'O' mapeia para ['U'] , 'L' mapeia para ['N'] , 'Q' maps para ['C'] e assim por diante.

Mas, como as letras em 'OLQIHXIRCKGNZ' também podem decifrar 'UNCOMFORTABLY' , também precisamos adicioná-las ao mapeamento da cifra. Digite o seguinte no shell interativo:

```
>>> letterMapping1 = simpleSubHacker.addLettersToMapping (letterMapping1,  
'OLQIHXIRCKGNZ', candidatos [1])  
>>> letterMapping1  
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [],  
'eu': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'],  
'N': ['L'], 'Q': ['C'], 'P': [], 'S': [], 'R': ['R'], 'U': [], 'T': [],  
'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E', 'Y']}
```

Observe que não muito mudou em letterMapping1, exceto que o mapeamento de cifra em letterMapping1 agora tem o mapa 'Z' em 'Y' além de 'E' . Isso porque addLettersToMapping () adiciona a letra à lista somente se a letra ainda não estiver lá.

Agora temos um mapeamento de cifras para a primeira das três palavras-chave. Precisamos obter um novo mapeamento para a segunda palavra-chave, 'PLQRZKBZB ' , e repetir o processo:

```
>>> letterMapping2 = simpleSubHacker.getBlankCipherletterMapping ()  
>>> wordPat = makeWordPatterns.getWordPattern ('PLQRZKBZB')  
>>> candidatos = wordPatterns.allPatterns [wordPat]  
>>> candidatos  
['CONVERSES', 'INCREASES', 'PORTENDED', 'UNIVERSES']  
>>> para candidato em candidatos:  
... letterMapping2 = simpleSubHacker.addLettersToMapping (letterMapping2,  
'PLQRZKBZB', candidate)  
...  
>>> letterMapping2  
{'A': [], 'C': [], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': [], 'F': [], 'eu':  
[], 'H': [], 'K': ['R', 'A', 'N'], 'J': [], 'M': [], 'L': ['O', 'N'], 'O': [],  
'N': [], 'Q': ['N', 'C', 'R', 'I'], 'P': ['C', 'eu', 'P', 'U'], 'S': [], 'R':  
['V', 'R', 'T'], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': [], 'Z':  
['E']}
```

Em vez de inserir quatro chamadas para addLettersToMapping () para cada uma

dessas quatro palavras candidatos, podemos escrever um loop que passa a lista de candidatos e pede addLettersToMapping () em cada um deles. Isso conclui o mapeamento da cifra da segunda cifra.

Em seguida, precisamos obter a interseção dos mapeamentos da cifra em letterMapping1 e letterMapping2 passando-os para intersectMappings () . Digite o seguinte no shell interativo:

```
>>> intersectedMapping = simpleSubHacker.intersectMappings  
(letterMapping1,  
letterMapping2)  
>>> intersectedMapping  
{'A': [], 'C': ['T'], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': ['B'], 'F': [],  
'eu': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'],  
'N': ['L'], 'Q': ['C'], 'P': ['C', 'eu', 'P', 'U'], 'S': [], 'R': ['R'],  
'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': 'E']}
```

Agora, a lista de potenciais letras de descriptografia para qualquer cipherletter no mapeamento entrecortada deve ser apenas os potenciais letras de descriptografia que estão em *ambos* letterMapping1 e letterMapping2 .

Por exemplo, a lista em intersectedMapping para a tecla 'Z' é apenas ['E'] porque letterMapping1 tinha ['E', 'Y'] mas letterMapping2 tinha apenas ['E'] .

Em seguida, repetimos todas as etapas anteriores para a terceira palavra-chave, 'MPBKSSIPLC' , da seguinte maneira:

```
>>> letterMapping3 = simpleSubHacker.getBlankCipherletterMapping()  
>>> wordPat = makeWordPatterns.getWordPattern('MPBKSSIPLC')  
>>> candidatos = wordPatterns.allPatterns[wordPat]  
>>> para i no intervalo(len(candidatos)):  
... . letterMapping3 = simpleSubHacker.addLettersToMapping(letterMapping3,  
'MPBKSSIPLC', candidatos[i])  
...  
>>> letterMapping3  
{'A': [], 'C': ['Y', 'T'], 'B': ['M', 'S'], 'E': [], 'D': [], 'G': [],  
'F': [], 'eu': ['E', 'O'], 'H': [], 'K': ['eu', 'A'], 'J': [], 'M': ['A', 'D'],  
'L': ['L', 'N'], 'O': [], 'N': [], 'Q': [], 'P': ['D', 'T'], 'S': ['T', 'P'],  
'R': [], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': [], 'Z': []}
```

Digite o seguinte no shell interativo para interceptar letterMapping3 com

intersectedMapping , que é o mapeamento intersectado de letterMapping1 e letterMapping2 :

```
>>> intersectedMapping = simpleSubHacker.intersectMappings  
(intersectedMapping,  
letterMapping3)  
>>> intersectedMapping  
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], 'F': [],  
'eu': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['A', 'D'], 'L': ['N'],  
'O': ['U'], 'N': ['L'], 'Q': ['C'], 'P': ['eu'], 'S': ['T', 'P'], 'R': ['R'] ,  
'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z E']}
```

Neste exemplo, podemos encontrar soluções para as chaves que possuem apenas um valor em sua lista. Por exemplo, 'K' descriptografa para 'A' . Mas note que a tecla 'M' pode decodificar para 'A' ou ' D' . Como sabemos que 'K' descriptografa para 'A' , podemos deduzir que a chave 'M' deve descriptografar para 'D' , não 'A' . Afinal, se a letra resolvida é usada por uma cifra, ela não pode ser usada por outra cifra, porque a simples cifra de substituição criptografa uma letra de texto simples para exatamente uma cifra.

Vejamos como a função removeSolvedLettersFromMapping () localiza essas letras resolvidas e as remove da lista de possíveis letras de descriptografia. Nós precisaremos do intersectedMapping que acabamos de criar, então não feche a janela IDLE ainda.

Identificando cartas resolvidas em mapeamentos

A função removeSolvedLettersFromMapping () procura por qualquer cipherletter no parâmetro letterMapping que tenha apenas uma possível letra de descriptografia. Esses codificadores são considerados resolvidos, o que significa que quaisquer outros codificadores com essa letra resolvida em sua lista de descriptografia em potencial letras não podem decifrar a esta carta. Isso poderia causar uma reação em cadeia, porque quando uma possível carta de descriptografia é removida de outras listas de possíveis cartas de descriptografia contendo apenas duas letras, o resultado poderia ser uma nova cifra de correção resolvida. O programa lida com essa situação fazendo um loop e removendo a carta recém-resolvida de todo o mapeamento da cifra.

78. def removeSolvedLettersFromMapping (letterMapping):

- recorte -

88. loopAgain = True

```
89. while loopAgain:  
90. # Primeiro, suponha que não  
focaremos novamente: 91. loopAgain = False
```

Como uma referência a um dicionário é passada para o parâmetro letterMapping , esse dicionário conterá as alterações feitas na função removeSolvedLettersFromMapping () , mesmo depois que a função retornar. A linha 88 cria loopAgain , uma variável que contém um valor booleano, que determina se o código precisa fazer um loop novamente quando encontrar outra letra resolvida.

Se o loopAgain variável é definida para Verdadeiro na linha 88, a execução do programa entra no enquanto laço na linha 89. No início do ciclo, linha 91 conjuntos loopAgain para Falso . O código assume que esta é a última iteração através da enquanto laço na linha 89. O loopAgain variável somente é definida para Verdadeiro se o programa encontrar um novo cipherletter resolvido durante esta iteração.

A próxima parte do código cria uma lista de codificadores que possuem exatamente uma possível letra de descriptografia. Estas são as letras resolvidas que serão removidas do mapeamento.

```
93. # solvedLetters será uma lista de letras maiúsculas que tenham um
```

```
94. # e somente um mapeamento possível em letterMapping:
```

```
95. solvedLetters = []
```

```
96. para cipherletter em LETTERS:
```

```
97. if len (letterMapping [cipherletter]) == 1:
```

```
98. solvedLetters.append (letterMapping [cipherletter] [0])
```

O loop for na linha 96 passa por todos os 26 códigos de criptografia possíveis e examina a lista de cartas de descriptografia em potencial para essa cipherletter (isto é, a lista em letterMapping [cipherletter]) do mapeamento do catálogo .

A linha 97 verifica se o comprimento dessa lista é 1 . Se for, sabemos que há apenas uma letra que a cifra pode decifrar e a cifra é solucionada. A linha 98 adiciona a letra de descriptografia resolvida à lista solvedLetters . A letra solucionada está sempre em letterMapping [cipherletter] [0] porque letterMapping [cipherletter] é uma lista de possíveis letras de descriptografia que tem apenas um valor de cadeia no índice 0 da lista.

Após o loop for anterior que iniciou na linha 96 terminar, a variável

solvedLetters deve conter uma lista de todas as descriptografia de uma cipherletter. A linha 98 armazena essas seqüências descriptografadas em solvedLetters como uma lista.

Neste ponto, o programa é feito identificando todas as letras resolvidas. Em seguida, ele verifica se eles estão listados como possíveis cartas de descriptografia para outros criptógrafos e os remove.

Para fazer isso, o loop for na linha 103 percorre todos os 26 criuletters possíveis e examina a lista de cartas de descriptografia em potencial do mapeamento de cifra.

103. para cipherletter em LETTERS:

104. para s em solvedLetters:

105. if len (letterMapping [cipherletter])! = 1 es em
letterMapping [cipherletter]:

106. letterMapping [cipherletter] .remove (s)

107. se len (letterMapping [cipherletter]) == 1:

108. # Uma nova letra está agora resolvida, então faça um loop novamente:

109. loopAgain = True

110. return letterMapping

Para cada cipherletter examinada, a linha 104 percorre as letras em SoledLetters para verificar se alguma delas existe na lista de possíveis letras de descriptografia para letterMapping [cipherletter] .

A linha 105 verifica se uma lista de possíveis letras de descriptografia não é resolvida, verificando se len (letterMapping [cipherletter])! = 1 e verificando se a letra resolvida existe na lista de possíveis letras de descriptografia. Se ambos os critérios forem atendidos, essa condição retornará True e a linha 106 removerá a letra resolvida s da lista de possíveis letras de descriptografia.

Se essa remoção deixar apenas uma letra na lista de possíveis letras de descriptografia, a linha 109 definirá a variável loopAgain como True para que o código possa remover essa carta recém-resolvida do mapeamento de cifra na próxima iteração do loop.

Depois da enquanto laço na linha 89 passou por uma iteração completa sem loopAgain sendo definida como verdadeira , o programa vai além do laço e linha 110 retorna o mapeamento cipherletter armazenados em letterMapping .

A variável letterMapping deve agora conter um mapeamento de cifras

parcialmente ou potencialmente resolvido.

Testando a função removeSolvedLetterFromMapping ()

Vamos ver o removeSolvedLetterFromMapping () em ação, testando-o no shell interativo. Retornar para a janela do shell interativo que você abriu quando criou o intersectedMapping . (Se você fechou a janela, não se preocupe, basta digitar novamente as instruções em “ [Como funcionam as funções auxiliares de mapeamento de letras](#) ” na [página 235](#) e, em seguida, siga este exemplo.)

Para remover as letras resolvidas do intersectedMapping , insira o seguinte no shell interativo:

```
>>> letterMapping = simpleSubHacker.removeSolvedLettersFromMapping ( intersectedMapping )
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'eu': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['D'], 'L': ['N'], 'O':
['U'], 'N': ['L'], 'Q': ['C'], 'P': ['eu'], 'S': ['P'], 'R': ['R'], 'U': [],
'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

Quando você remover as letras resolvidas do intersectedMapping , observe que 'M' agora tem apenas uma letra potencial de descriptografia, 'D' , que é o que previmos que seria o caso. Agora, cada cipherletter tem apenas uma letra de descriptografia em potencial, portanto, podemos usar o mapeamento da cifra para começar a descriptografar. Precisamos retornar a esse exemplo de shell interativo mais uma vez, portanto, mantenha a janela aberta.

A função hackSimpleSub ()

Agora que você viu como as funções getBlankCipherletterMapping () , addLettersToMapping () , intersectMappings () e removeSolvedLettersFromMapping () manipulam os mapeamentos da cifra que você as passa, vamos usá-las em nosso programa *simpleSubHacker.py* para descriptografar uma mensagem.

A linha 113 define a função hackSimpleSub () , que recebe uma mensagem de texto cifrado e usa as funções auxiliares de mapeamento de letras para retornar um mapeamento de cifras parcial ou totalmente resolvido:

```
113. def hackSimpleSub (mensagem):
114.     intersectedMap = getBlankCipherletterMapping ()
115.     cipherwordList = nonLettersOrSpacePattern.sub (",
```

message.upper ()). Split ()

Na linha 114, criamos um novo mapeamento de cifra que armazenamos na variável intersectedMap . Esta variável irá eventualmente conter os mapeamentos cruzados de cada uma das palavras-chave.

Na linha 115, removemos quaisquer caracteres que não sejam letras da mensagem . O objeto regex em nonLettersOrSpacePattern corresponde a qualquer string que não seja um caractere de letra ou espaço em branco. O método sub () é chamado em uma expressão regular e recebe dois argumentos. A função pesquisa a sequência no segundo argumento em busca de correspondências e substitui essas correspondências pela sequência no primeiro argumento. Em seguida, ele retorna uma string com todas essas substituições. Neste exemplo, o método sub () diz ao programa para percorrer a sequência de mensagens em maiúsculas e substituir todos os caracteres que não são letras por uma cadeia em branco ("). Isso faz sub () retornar uma string com todos os caracteres de pontuação e número removidos, e essa string é armazenada na variável cipherwordList .

Depois que a linha 115 é executada, a variável cipherwordList deve conter uma lista de cadeias maiúsculas das palavras-chave individuais anteriormente na mensagem .

O loop for na linha 116 atribui cada cadeia na lista de mensagens à variável da palavra-chave . Dentro desse loop, o código cria um mapa em branco, obtém os candidatos da palavra-código, adiciona as letras dos candidatos a um mapeamento de cifra e, em seguida, cruza esse mapeamento com intersectedMap .

116. para cipherword in cipherwordList:

117. # Obtenha um novo mapeamento de cifra para cada palavra de texto cifrado:

118. candidateMap = getBlankCipherletterMapping ()

120. wordPattern = makeWordPatterns.getWordPattern (cipherword)

121. if wordPattern não em wordPatterns.allPatterns:

122. continue # Essa palavra não estava em nosso dicionário, então continue.

124. # Adicione as letras de cada candidato ao mapeamento:

125. para candidato em wordPatterns.allPatterns [wordPattern]:

126. addLettersToMapping (candidateMap, cipherword, candidate)

128. # Increte o novo mapeamento com o mapeamento interseccional

existente:

129. intersectedMap = intersectMappings (intersectedMap, candidateMap)

A linha 118 obtém um novo mapeamento de cifra em branco da função getBlankCipherletterMapping () e armazena-a na variável candidateMap .

Para encontrar os candidatos para a palavra cifrada atual, a linha 120 chama getWordPattern () no módulo makeWordPatterns . Em alguns casos, a palavra cifrada pode ser um nome ou uma palavra muito incomum que não existe no dicionário; nesse caso, seu padrão de palavras provavelmente não existirá em wordPatterns . Se o padrão de palavras da palavra criptografada não existir nas chaves do dicionário wordPatterns.allPatterns , a palavra de texto simples original não existe no arquivo do dicionário. Nesse caso, a palavra cifrada não recebe um mapeamento, e a instrução continue na linha 122 retorna à próxima cifra na lista na linha 116.

Se a execução atingir a linha 125, saberemos que a palavra padrão existe em wordPatterns.allPatterns . Os valores no dicionário allPatterns são listas de strings das palavras inglesas com o padrão em wordPattern . Como os valores estão na forma de uma lista, usamos um loop for para iterar sobre eles. A variável candidata é configurada para cada uma dessas cadeias de palavras em inglês em cada iteração do loop.

O loop for na linha 125 chama addLettersToMapping () na linha 126 para atualizar o mapeamento cipherletter em candidateMap usando as letras em cada um dos candidatos. A função addLettersToMapping () modifica a lista diretamente, então candidateMap é modificado pelo tempo que a chamada de função retorna.

Depois que todas as letras nos candidatos são adicionadas ao mapeamento cipherletter em candidateMap , a linha 129 intercepta o candidateMap com intersectedMap e retorna o novo valor de intersectedMap .

Neste ponto, a execução do programa retorna ao início do loop for na linha 116 para criar um novo mapeamento para a próxima palavra cifrada na lista cipherwordList , e o mapeamento para a próxima cifra também é cruzado com intersectedMap . O loop continua mapeando as palavras-chave até atingir a última palavra em cipherWordList .

Quando temos o último mapeamento de cifra cruzada que contém os mapeamentos de todas as palavras-chave no texto cifrado, passamos para

removeSolvedLettersFromMapping () na linha 132 para remover quaisquer letras resolvidas.

131. # Remova quaisquer letras resolvidas das outras listas:

132. return removeSolvedLettersFromMapping (intersectedMap)

O mapeamento cipherletter retornado de removeSolvedLettersFromMapping () é então retornado para a função hackSimpleSub () . Agora, temos parte da solução da criptografia, para que possamos começar a descriptografar a mensagem.

O método da string replace ()

O método de string replace () retorna uma nova string com caracteres substituídos. O primeiro argumento é a substring para procurar, e o segundo argumento é a string para substituir essas subsequências. Digite o seguinte no shell interativo para ver um exemplo:

```
>>> 'mississippi'.replace ('s', 'X')
'miXXiXXippi'
>>> 'cão'.replace ('d', 'bl')
'blog'
>>> 'jogger'.replace ('ger', 's')
'jogs'
```

Usaremos o método de string replace () em decryptMessage () no programa *simpleSubHacker.py* .

Descriptografar a mensagem

Para descriptografar nossa mensagem, usaremos a função simpleSubstitutionCipher.decryptMessage () que já programamos em *simpleSubstitutionCipher.py* . Mas simpleSubstitutionCipher.decryptMessage () descriptografa usando apenas as chaves, não os mapeamentos de letras, portanto não podemos usar a função diretamente. Para resolver esse problema, criaremos uma função decryptWithCipherletterMapping () que usa um mapeamento de letras, converte o mapeamento em uma chave e passa a chave e a mensagem para simpleSubstitutionCipher.decryptMessage () . A função decryptWithCipherletterMapping () irá retornar uma string descriptografada. Lembre-se de que as chaves de substituição simples são strings de 26 caracteres e o caractere no índice 0 da string de chave é o caractere criptografado de A, o caractere no índice 1 é o caractere criptografado para B e assim por diante.

Para converter um mapeamento em uma saída de descriptografia, podemos ler

facilmente, primeiro criaremos uma chave de espaço reservado, que terá a seguinte aparência: ['x', 'x', 'x', 'x', 'x' , x, x x ',' x ',' x ',' x]. A minúscula 'x' pode ser usada na chave de espaço reservado porque a chave real usa apenas letras maiúsculas. (Você pode usar qualquer caractere que não seja uma letra maiúscula como espaço reservado.) Como nem todas as letras terão uma descriptografia, precisamos ser capazes de distinguir entre partes da lista de chaves que foram preenchidas com as letras de descriptografia e aquelas em que a descriptografia não foi resolvida. O 'x' indica letras que não foram resolvidas.

Vamos ver como tudo isso vem junto no código-fonte:

```
135. def decryptWithCipherletterMapping (ciphertext, letterMapping):
136. # Retorna uma string do texto cifrado descriptografado com o mapeamento
de letras,
137. # com qualquer letra decifrada ambígua substituída por um sublinhado.
138
139. # Primeiro, crie uma subchave simples a partir do mapeamento
letterMapping:
140. key = ['x'] * len (LETTERS)
141. para a caligrafia em LETTERS:
142. if len (letterMapping [cipherletter]) == 1:
143. # Se houver apenas uma letra, adicione-a à chave:
144. keyIndex = LETTERS.find (letterMapping [cipherletter] [0])
145. key [keyIndex] = cipherletter
```

A linha 140 cria a lista de marcadores de posição, replicando a lista de itens individuais ['x'] 26 vezes. Porque LETRAS é uma seqüência das letras do alfabeto, len (LETRAS) avalia para 26 . Quando usado em uma lista inteiro, o operador de multiplicação (*) executa a replicação de lista.

O loop for na linha 141 verifica cada uma das letras em LETTERS para a variável cipherletter e, se a cipherletter for resolvida (isto é, letterMapping [cipherletter] tem apenas uma letra), substitui o espaço reservado 'x' por essa letra .

O letterMapping [cipherletter] [0] na linha 144 é a letra de descriptografia, e keyIndex é o índice da letra de descriptografia em LETTERS , que é retornada da chamada find () . A linha 145 define esse índice na lista de chaves para a letra de descriptografia.

No entanto, se o cipherletter não tiver uma solução, a função insere um sublinhado para esse cipherletter para indicar quais caracteres permanecem sem solução. A linha 147 substitui as letras minúsculas na cipherletter por um sublinhado, e a linha 148 substitui as letras maiúsculas por um sublinhado:

146. mais:

147. ciphertext = ciphertext.replace (cipherletter.lower (), '_')

148. texto cifrado = ciphertext.replace (cipherletter.upper (), '_')

Após substituir todas as partes na lista em key pelas letras solucionadas, a função combina a lista de strings em uma única string usando o método join () para criar uma chave de substituição simples. Esta cadeia é passada para a função decryptMessage () no programa *simpleSubCipher.py*.

149. key = " .join (chave)

150

151. # Com a chave que criamos, decriptografe o texto cifrado:

152. return simpleSubCipher.decryptMessage (key, ciphertext)

Finalmente, a linha 152 retorna a string de mensagem descriptografada da função decryptMessage (). Agora temos todas as funções que precisamos para encontrar um mapeamento de letras cruzadas, hackear uma chave e descriptografar uma mensagem. Vamos ver um exemplo rápido de como essas funções funcionam no shell interativo.

Descriptografando no Shell Interativo

Vamos voltar ao exemplo que usamos em “[Como funcionam as funções auxiliares de mapeamento de letras](#)” na [página 235](#). Usaremos a variável intersectedMapping que criamos em nossos exemplos de shell anteriores para descriptografar a mensagem de texto cifrado 'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC'.

Digite o seguinte no shell interativo:

```
>>> simpleSubHacker.decryptWithCipherletterMapping ('OLQIHXIRCKGNZ  
PLQRZKBZB  
MPBKSSIPLC', intersectedMapping) NÃO COMPORTAVEL  
AUMENTA A DESAPONTAMENTO
```

O texto cifrado descriptografa a mensagem “Desconfortável aumenta decepciona”. Como você pode ver, a função decryptWithCipherletterMapping () funcionou perfeitamente e retornou a string totalmente descriptografada. Mas

este exemplo não mostra o que acontece quando não temos todas as letras que aparecem no texto cifrado resolvido. Para ver o que acontece quando estamos perdendo a descriptografia de uma cifra, vamos remover a solução para os criptografadores 'M' e 'S' do intersectedMapping usando as seguintes instruções:

```
>>> intersectedMapping ['M'] = []
>>> intersectedMapping ['S'] = []
```

Em seguida, tente descriptografar o texto cifrado com intersectedMapping novamente:

```
>>> simpleSubHacker.decryptWithCipherletterMapping ('OLQIHXIRCKGNZ
PLQRZKBZB
MPBKSSIPLC', intersectedMapping)
INCOMBRANTES INCOMBUSTÍVEIS _ISA__OINT
```

Desta vez, parte do texto cifrado não foi decifrado. Os codificadores sem uma letra de descriptografia foram substituídos por sublinhados.

Este é um texto cifrado curto para hackear. Normalmente, as mensagens criptografadas seriam muito mais longas. (Este exemplo foi especificamente escolhido para ser hackable. Mensagens tão curtas quanto este exemplo geralmente não podem ser hackeadas usando o método word pattern.) Para hackear criptografias mais longas, você precisará criar um mapeamento cipherletter para cada palavra cifrada nas mensagens mais longas e depois interceptá-los todos juntos. A função hackSimpleSub () chama as outras funções do nosso programa para fazer exatamente isso.

Chamando a função main ()

As linhas 155 e 156 chamam a função main () para executar *simpleSubHacker.py* se estiver sendo executada diretamente, em vez de ser importada como um módulo por outro programa Python:

```
155. if __name__ == '__main__':
156.     main()
```

Isso completa nossa discussão sobre todas as funções que o programa *simpleSubHacker.py* usa.

NOTA

Nossa abordagem de hacking funciona somente se os espaços não estiverem criptografados. Você pode expandir o conjunto de símbolos para que o

programa de criptografia criptografe espaços, números e caracteres de pontuação, bem como letras, dificultando ainda mais (mas não impossíveis) as mensagens criptografadas. Hackear tais mensagens envolveria a atualização das freqüências não apenas de letras, mas de todos os símbolos do conjunto de símbolos. Isso torna o hacking mais complicado, razão pela qual este livro criptografou apenas letras.

Resumo

Ufa! O programa *simpleSubHacker.py* é bastante complicado. Você aprendeu a usar o mapeamento de cifra para modelar as possíveis letras de descriptografia para cada letra de texto cifrado. Você também aprendeu como diminuir o número de chaves possíveis, adicionando letras potenciais ao mapeamento, interceptando-as e removendo as letras resolvidas de outras listas de possíveis cartas de descriptografia. Em vez de forçar 403, 403, 91, 126, 126, 605, 635, 584, 000, 000 chaves possíveis, você pode usar algum código Python sofisticado para descobrir a maioria (senão todas) da chave de substituição simples original.

A principal vantagem da simples cifra de substituição é o grande número de chaves possíveis. A desvantagem é que é relativamente fácil comparar palavras-chave a palavras em um arquivo de dicionário para determinar quais criptografias descriptografam a quais letras. No [Capítulo 18](#), exploraremos uma codificação de substituição polialfabética mais poderosa, chamada de cifra de Vigenère, que foi considerada impossível de se quebrar por várias centenas de anos.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Qual é o padrão de palavras para a palavra *olá*?
2. O *mamute* e os *óculos de proteção* têm o mesmo padrão de palavras?
3. Qual palavra poderia ser a palavra de texto simples possível para a palavra-chave *PYYACAO? Alegado, eficientemente ou poodle?*

18

PROGRAMAÇÃO DO CIPHER VIGENÈRE

“Acreditei então, e continuo a acreditar agora, que os benefícios para nossa

segurança e liberdade de criptografia amplamente disponível superam de longe o dano inevitável que vem do seu uso por criminosos e terroristas.”

—Matt Blaze, AT & T Labs, setembro de 2001



O criptógrafo italiano Giovan Battista Bellaso foi a primeira pessoa a descrever a cifra de Vigenère em 1553, mas foi batizada com o nome do diplomata francês Blaise de Vigenère, uma das muitas pessoas que reinventaram a cifra nos anos subsequentes. Era conhecido como “le chiffre indéchiffrable”, que significa “a cifra indecifrável”, e permaneceu ininterrupta até o polímata britânico Charles Babbage quebrá-lo no século XIX.

Como a cifra Vigenère tem muitas chaves possíveis para a força bruta, mesmo com nosso módulo de detecção de inglês, é uma das mais fortes cifras discutidas até agora neste livro. É até mesmo invencível ao ataque de padrão de palavras que você aprendeu no [Capítulo 17](#).

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Subchaves
- Construindo strings usando o processo list-append-join

Usando várias chaves de letra na criptografia Vigenère

Ao contrário da cifra de César, a cifra de Vigenère tem várias chaves. Por usar mais de um conjunto de substituições, a cifra de Vigenère é uma *cifra de substituição polialfabética*. Diferentemente da simples cifra de substituição, a análise de freqüência sozinha não derrotará a cifra de Vigenère. Em vez de usar uma chave numérica entre 0 e 25, como fizemos na cifra de César, usamos uma chave de letra para o Vigenère.

A chave Vigenère é uma série de letras, como uma única palavra em inglês, que é dividida em várias subchaves de uma única letra que criptografam letras no texto simples. Por exemplo, se usarmos uma chave Vigenère de PIZZA, a primeira subchave é P, a segunda subchave é I, a terceira e quarta subchaves são ambas Z e a quinta subchave é A. A primeira subchave criptografa a primeira

letra do texto simples , a segunda subchave criptografa a segunda letra e assim por diante. Quando chegamos à sexta letra do texto original, retornamos à *primeira* subchave.

Usar a cifra de Vigenère é o mesmo que usar múltiplas cifras de César, como mostrado na [Figura 18-1](#) . Em vez de criptografar todo o texto simples com uma cifra de César, aplicamos uma cifra diferente de César a cada letra do texto simples.

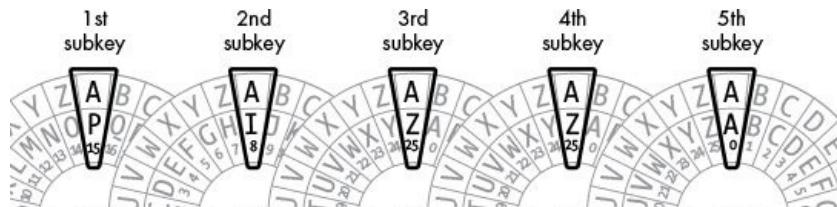


Figura 18-1: Cifras múltiplas de César combinam-se para fazer a cifra de Vigenère

Cada subchave é convertida em um inteiro e serve como uma chave de cifra de César. Por exemplo, a letra A corresponde à chave Cifra César 0. A letra B corresponde à chave 1 e assim por diante até Z para a tecla 25, como mostrado na [Figura 18-2](#) .

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Figura 18-2: Chaves cifradas e suas letras correspondentes

Vamos ver um exemplo. Segue-se a mensagem COMMON SENSE IS NOT COMMON mostrada ao lado da tecla Vigenère PIZZA. O texto simples é mostrado com a subchave correspondente que criptografa cada letra abaixo dela.

SENSO COMUM NÃO É TÃO COMUM
PIZZAPIZZAPIZZAPIZZAPIZZ

Para criptografar o primeiro C no texto sem formatação com a subchave P, criptografe-o com a cifra de César usando a chave numérica correspondente da subchave 15, que resulta na cifra-chave R e repita o processo para cada letra do texto sem formatação percorrendo as subchaves. [A Tabela 18-1](#) mostra esse processo. O inteiro para a letra e a subchave do texto plano (fornecidos entre parênteses) são somados para produzir o inteiro para a letra do texto cifrado.

Tabela 18-1: Criptografando cartas com sub-chaves Vigenère

Carta de texto	Sub-	Carta de texto
-----------------------	-------------	-----------------------

simples	chave	cifrado
C (2)	P (15)	R (17)
O (14)	Eu (8)	W (22)
M (12)	Z (25)	L (11)
M (12)	Z (25)	L (11)
O (14)	A (0)	O (14)
N (13)	P (15)	C (2)
S (18)	Eu (8)	A (0)
E (4)	Z (25)	D (3)
N (13)	Z (25)	M (12)
S (18)	A (0)	S (18)
E (4)	P (15)	T (19)
Eu (8)	Eu (8)	Q (16)
S (18)	Z (25)	R (17)
N (13)	Z (25)	M (12)
O (14)	A (0)	O (14)
T (19)	P (15)	Eu (8)
S (18)	Eu (8)	A (0)
O (14)	Z (25)	N (13)
C (2)	Z (25)	B (1)

O (14)	A (0)	O (14)
M (12)	P (15)	B (1)
M (12)	Eu (8)	U (20)
O (14)	Z (25)	N (13)
N (13)	Z (25)	M (12)

Utilizando a cifra Vigenère com a chave PIZZA (que é constituída pelas subchaves 15, 8, 25, 25, 0) encripta o texto plano SENSO COMUM NÃO É TÃO COMUM no texto cifrado RWLLOC ADMST QR MOI UM BOBUNM.

Chaves Vigenere mais longas são mais seguras

Quanto mais letras na chave Vigenère, mais forte será a mensagem criptografada contra um ataque de força bruta. PIZZA é uma má escolha para uma chave Vigenère porque tem apenas cinco letras. Uma chave com cinco letras tem $11.881.376$ combinações possíveis (porque 26 letras com a potência de 5 é $26^5 = 26 \times 26 \times 26 \times 26 \times 26 = 11.881.376$). Onze milhões de chaves são demais para um ser humano à força bruta, mas um computador pode testá-las em apenas algumas horas. Primeiro, ele tentaria descriptografar a mensagem usando a chave AAAAA e verificar se a descriptografia resultante estava em inglês. Então poderia tentar AAAAB, depois AAAAC, e assim por diante até chegar a PIZZA.

A boa notícia é que, para cada letra adicional que a chave possui, o número de chaves possíveis se multiplica por 26. Uma vez que haja quatrilhões de chaves possíveis, a cifra levaria um computador muitos anos para ser interrompida. [A Tabela 18-2](#) mostra quantas chaves possíveis existem para cada tamanho de chave.

Tabela 18-2: Número de chaves possíveis com base no comprimento da chave Vigenère

Comprimento da chave	Equação	Chaves possíveis
-----------------------------	----------------	-------------------------

1	26	= 26
---	----	------

2	26×26	= 676
3	676×26	= 17.576
4	$17,576 \times 26$	= 456.976
5	$456,976 \times 26$	= 11.881.376
6	$11.881.376 \times 26$	= 308.915.776
7	$308.915.776 \times 26$	= 8,031,810,176
8	$8,031,810,176 \times 26$	= 208.827.064.576
9	$208.827.064.576 \times 26$	= 5,429,503,678,976
10	$5,429,503,678,976 \times 26$	= 141,167,095,653,376
11	$141,167,095,653,376 \times 26$	= 3,670,344,486,987,776
12	$3,670,344,486,987,776 \times 26$	= 95,428,956,661,682,176
13	$95,428,956,661,682,176 \times 26$	= 2.481.152.873.203.736.576
14	$2.481.152.873.203.736.576 \times 26$	= 64,509,974,703,297,150,976

Com chaves com doze ou mais letras, torna-se impossível para um mero laptop quebrá-las em um período de tempo razoável.

Escolhendo uma chave que impede ataques de dicionário

Uma chave Vigenère não precisa ser uma palavra real como PIZZA. Pode ser qualquer combinação de letras de qualquer tamanho, como a chave de doze letras DURIWKNMFICK. De fato, não usar uma palavra que possa ser encontrada no dicionário é melhor. Mesmo que a palavra RADIOLOGISTS seja também uma chave de doze letras mais fácil de lembrar do que DURIWKNMFICK, um criptoanalista pode antecipar que o criptógrafo está usando uma palavra em inglês como chave.

A tentativa de ataque de força bruta usando todas as palavras inglesas no dicionário é conhecida como *ataque de dicionário*. Existem 95,428,956,661,682,176 possíveis chaves de doze letras, mas existem apenas 1800 palavras de doze letras em nosso arquivo de dicionário. Se usarmos uma palavra de doze letras do dicionário como uma chave, seria mais fácil para a força bruta do que uma chave aleatória de três letras (que tem 17.576 chaves possíveis).

É claro que o criptógrafo tem uma vantagem em que o criptoanalista não conhece o comprimento da chave Vigenère. Mas o criptoanalista podia tentar todas as teclas de uma só letra, depois todas as de duas letras, e assim por diante, o que ainda lhes permitiria encontrar uma palavra-chave no dicionário muito rapidamente.

Código Fonte do Programa de Cifras Vigenère

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos, salve-o como *vigenereCipher.py* e certifique-se de que *pyperclip.py* esteja no mesmo diretório. Pressione F5 para executar o programa.

vigenereCipher.py

```
1. # Vigenere Cipher (Cifra de Substituição Polialfabética)
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. import pyperclip
5
6. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
7
8. def main():
9.     # Este texto pode ser baixado em https://www.nostarch.com/
10.    códigos de cracking /:
11.    myMessage = """ Alan Mathison Turing foi um matemático britânico,
12.    lógico, criptoanalista e cientista da computação. """
13.    myKey = 'ASIMOV'
14.    myMode = 'encrypt' # Defina como 'encriptar' ou 'decifrar'.
15.    translated = encryptMessage (myKey, myMessage)
```

```
16. elif myMode == 'decifrar':  
17.     translated = decryptMessage (myKey, myMessage)  
18  
19.     print ('% sed mensagem:% (myMode.title ()))  
20.     impressão (tradução)  
21.     pyperclip.copy (tradução)  
22.     print ()  
23.     print ('A mensagem foi copiada para a área de transferência.')  
24  
25  
26. def encryptMessage (key, message):  
27.     return translateMessage (chave, mensagem, 'criptografar')  
28.  
29  
30. def decryptMessage (chave, mensagem):  
31.     return translateMessage (chave, mensagem, 'descriptografar')  
32  
33  
34. def translateMessage (chave, mensagem, modo):  
35.     translated = [] # Armazena a string de mensagem criptografada /  
descriptografada.  
36  
37.     keyIndex = 0  
38.     key = key.upper ()  
39  
40.     para símbolo na mensagem: # Loop através de cada símbolo na mensagem.  
41.     num = LETTERS.find (symbol.upper ())  
42.     if num! = -1: # -1 significa que symbol.upper () não foi encontrado em  
LETTERS.  
43.     if mode == 'encriptar':  
44.         num + = LETTERS.find (key [keyIndex]) # Adicionar se estiver  
criptografando.  
45.     modo elif == 'decifrar':  
46.         num - = LETTERS.find (key [keyIndex]) # Subtrair se  
descriptografar.  
47  
48.     num% = len (LETTERS) # Lida com qualquer embrulho.
```

```
49.
50. # Adicione o símbolo criptografado / decriptografado ao final da tradução:
51. if symbol.isupper ():
52.     translated.append (LETTRAS [num])
53. elif symbol.islower ():
54.     translated.append (LETTERS [num.] .Lower ())
55
56. keyIndex += 1 # Vai para a próxima letra da chave.
57. se keyIndex == len (chave):
58.     keyIndex = 0
59. else:
60.     # Anexar o símbolo sem criptografar / descriptografar:
61.     translated.append (símbolo)
62
63. return " ".join (tradução)
64
65
66. # Se vigenereCipher.py for executado (em vez de importado como um
módulo), chame
67. # a função main ():
68. if __name__ == '__main__':
69.     main ()
```

Exemplo de Execução do Programa de Cifra Vigenère

Quando você executa o programa, sua saída ficará assim:

Mensagem criptografada:

Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi, lgouqdaf, kdmktsvmztsl,
izr
xoexghzr kkusitaaf.

A mensagem foi copiada para a área de transferência.

O programa imprime a mensagem criptografada e copia o texto criptografado para a área de transferência.

Configurando módulos, constantes e a função main ()

O início do programa tem os comentários usuais descrevendo o programa, uma declaração de importação para o módulo pyperclip e uma variável chamada LETTERS que contém uma string de cada letra maiúscula. A função main ()

para a cifra Vigenère é como as outras funções main () deste livro: ela começa definindo as variáveis para message , key e mode .

```
1. # Vigenere Cipher (Cifra de Substituição Polialfabética)
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar pyperclip
5
6. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
7
8. def main():
9. # Este texto pode ser baixado em https://www.nostarch.com/
códigos de cracking /:
10. myMessage = """ Alan Mathison Turing foi um matemático britânico,
lógico, criptoanalista e cientista da computação. """
11. myKey = 'ASIMOV'
12. myMode = 'encrypt' # Defina como 'encriptar' ou 'decifrar'.
13
14. if myMode == 'encriptar':
15.     translated = encryptMessage (myKey, myMessage)
16. elif myMode == 'decifrar':
17.     translated = decryptMessage (myKey, myMessage)
18
19. print ('% sed mensagem:% (myMode.title ()))
20. impressão (tradução)
21. pyperclip.copy (tradução)
22. print ()
23. print ('A mensagem foi copiada para a área de transferência.')
```

O usuário define essas variáveis nas linhas 10, 11 e 12 antes de executar o programa. A mensagem criptografada ou descriptografada (dependendo do que myMode está definido) é armazenada em uma variável chamada traduzida para que possa ser impressa na tela (linha 20) e copiada para a área de transferência (linha 21).

Construindo Strings com o Processo List-Append-Join

Quase todos os programas deste livro construiram uma string com código de alguma forma. Ou seja, o programa cria uma variável que inicia como uma

string em branco e, em seguida, adiciona caracteres usando a concatenação de strings. Isso é o que os programas de criptografia anteriores fizeram com a variável traduzida . Abra o shell interativo e digite o seguinte código:

```
>>> edifício = "
>>> para c em 'Hello world!':
>>> construção + = c
>>> print (construção)
```

Este código percorre cada caractere na string 'Hello world!' e concatena-o ao final da string armazenada no edifício . No final do loop, o edifício contém a string completa.

Embora a concatenação de strings pareça uma técnica simples, ela é muito ineficiente em Python. É muito mais rápido começar com uma lista em branco e, em seguida, usar o método de lista append () . Quando você terminar de criar a lista de strings, você pode converter a lista em um único valor de string usando o método join () . O código a seguir faz a mesma coisa que o exemplo anterior, mas mais rápido. Digite o código no shell interativo:

```
>>> construção = []
>>> para c em 'Hello world!':
>>> building.append (c)
>>> edifício = " .join (construção)
>>> print (construção)
```

Usar essa abordagem para criar strings em vez de modificar uma string resultará em programas muito mais rápidos. Você pode ver a diferença ao sincronizar as duas abordagens usando time.time () . Abra uma nova janela do editor de arquivos e digite o seguinte código:

stringTest.py

tempo de importação

```
startTime = time.time ()
para teste no intervalo (10000):
edifício =
para i no intervalo (10000):
edifício += 'x'
print ('Concatenação de cadeia:', (time.time () - startTime))
```

```
startTime = time.time ()  
para teste no intervalo (10000):  
edifício = []  
para i no intervalo (10000):  
building.append ('x')  
edifício = " .join (construção)  
print ('List anexando:', (time.time () - startTime))
```

Salve este programa como *stringTest.py* e execute-o. A saída será algo como isto:

Concatenação de cordas: 40.317070960998535

Lista anexar: 10.488219022750854

O programa *stringTest.py* define um *startTime* variável como o horário atual, executa código para anexar 10.000 caracteres a uma string usando concatenação e, em seguida, imprime o tempo necessário para concluir a concatenação. Em seguida, o programa redefine *startTime* para a hora atual, executa o código para usar o método de anexação de lista para criar uma cadeia de caracteres do mesmo tamanho e, em seguida, imprime o tempo total necessário para concluir. No meu computador, a utilização de concatenação de cadeias de caracteres para criar 10.000 cadeias com 10.000 caracteres cada demorava cerca de 40 segundos, mas o uso do processo *list-append-join* para executar a mesma tarefa levou apenas 10 segundos. Se o seu programa constrói muitas strings, o uso de listas pode tornar seu programa muito mais rápido.

Usaremos o processo *list-append-join* para criar strings para os programas restantes neste livro.

Criptografando e Decriptografando a Mensagem

Como o código de criptografia e descriptografia é basicamente o mesmo, criaremos duas funções de invólucro chamadas *encryptMessage ()* e *decryptMessage ()* para a função *translateMessage ()*, que conterá o código real para criptografar e descriptografar.

```
26. def encryptMessage (key, message):  
27.     return translateMessage (chave, mensagem, 'criptografar')  
28.  
29  
30. def decryptMessage (chave, mensagem):
```

31. return translateMessage (chave, mensagem, 'descriptografar')

A função translateMessage () constrói a cadeia criptografada (ou descriptografada), um caractere por vez. A lista em traduzidos armazena esses caracteres para que eles possam ser unidos quando o desenvolvimento da cadeia de caracteres for concluído.

34. def translateMessage (chave, mensagem, modo):

35. translated = [] # Armazena a string de mensagem criptografada / descriptografada.

36

37. keyIndex = 0

38. key = key.upper ()

Tenha em mente que a cifra de Vigenère é apenas a cifra de César, exceto que uma chave diferente é usada dependendo da posição da letra na mensagem. A variável keyIndex , que controla qual subchave usar, começa em 0 porque a letra usada para criptografar ou descriptografar o primeiro caractere da mensagem é a chave [0] .

O programa assume que a chave está em todas as letras maiúsculas. Para garantir que a chave seja válida, a linha 38 chama upper () na tecla .

O restante do código em translateMessage () é semelhante ao código de cifra de César:

40. para símbolo na mensagem: # Loop através de cada símbolo na mensagem.

41. num = LETTERS.find (symbol.upper ())

42. if num! = -1: # -1 significa que symbol.upper () não foi encontrado em LETTERS.

43. if mode == 'encriptar':

44. num + = LETTERS.find (key [keyIndex]) # Adicionar se estiver criptografando.

45. modo elif == 'decifrar':

46. num - = LETTERS.find (key [keyIndex]) # Subtrair se descriptografar.

O loop for na linha 40 define os caracteres em mensagem para o símbolo da variável em cada iteração do loop. A linha 41 encontra o índice da versão maiúscula do símbolo em LETTERS , que é como traduzimos uma letra em um número.

Se num não estiver definido como -1 na linha 41, a versão em maiúsculas do símbolo foi encontrada em LETTERS (o que significa que o símbolo é uma letra). A variável keyIndex rastreia qual subchave usar e a subchave é sempre a chave que o [keyIndex] avalia.

Claro, isso é apenas uma única letra. Precisamos encontrar o índice desta letra em LETTERS para converter a subchave em um inteiro. Este inteiro é então adicionado (se encriptado) ao número do símbolo na linha 44 ou subtraído (se decifrado) ao número do símbolo na linha 46.

No código de cifra de Caesar, verificamos se o novo valor de num era menor que 0 (caso em que adicionamos len (LETTERS) a ele) ou se o novo valor de num era len (LETTERS) ou maior (nesse caso , nós subtraímos len (LETTERS) disto). Essas verificações manipulam os casos de envolvimentos.

No entanto, existe uma maneira mais simples de lidar com esses dois casos. Se modificamos o inteiro armazenado em num por len (LETTERS) , podemos realizar o mesmo cálculo em uma única linha de código:

48. num% = len (LETTERS) # Lida com qualquer embrulho.

Por exemplo, se num era -8 , gostaríamos de adicionar 26 (isto é, len (LETTERS)) a ele para obter 18 , e isso pode ser expresso como -8% 26 , que é avaliado como 18 . Ou, se num for 31 , queremos subtrair 26 para obter 5 e 31% 26 para 5 . A aritmética modular na linha 48 lida com ambos os casos envolventes.

O caractere criptografado (ou descriptografado) existe em LETTERS [num] . No entanto, queremos que o caso do caractere criptografado (ou descriptografado) corresponda ao caso original do símbolo .

50. # Adicione o símbolo criptografado / descriptografado ao final da tradução:

51. if symbol.isupper ():

52. translated.append (LETRAS [num])

53. elif symbol.islower ():

54. translated.append (LETTERS [num.] .Lower ())

Portanto, se o símbolo é uma letra maiúscula, a condição na linha 51 é True e a linha 52 acrescenta o caractere em LETTERS [num] à tradução porque todos os caracteres em LETTERS já estão em maiúsculas.

No entanto, se o símbolo for uma letra minúscula, a condição na linha 53 será True e a linha 54 anexará o formato minúsculo de LETTERS [num] à tradução .

É assim que fazemos com que a mensagem criptografada (ou descriptografada) corresponda ao invólucro original da mensagem.

Agora que traduzimos o símbolo, queremos garantir que, na próxima iteração do loop for , usemos a próxima subchave. A linha 56 incrementa keyIndex por 1, então a próxima iteração usa o índice da próxima subchave:

56. keyIndex += 1 # Vai para a próxima letra da chave.

57. se keyIndex == len (chave):

58. keyIndex = 0

No entanto, se estivéssemos na última subchave da chave, keyIndex seria igual ao tamanho da chave . A linha 57 verifica essa condição e redefine o keyIndex de volta para 0 na linha 58, se for esse o caso, de modo que a chave [keyIndex] aponte para a primeira subchave.

O recuo indica que a instrução else na linha 59 está emparelhada com a instrução if na linha 42:

59. else:

60. # Anexar o símbolo sem criptografar / descriptografar:

61. translated.append (símbolo)

O código na linha 61 é executado se o símbolo não foi encontrado na string LETTERS . Isso acontece se o símbolo for um número ou um sinal de pontuação, como '5' ou '?' . Neste caso, a linha 61 acrescenta o símbolo não modificado à tradução .

Agora que terminamos de construir a string traduzida , chamamos o método join () na string em branco:

63. return " .join (tradução)

Essa linha faz a função retornar toda a mensagem criptografada ou descriptografada quando a função é chamada.

Chamando a função main ()

As linhas 68 e 69 terminam o código do programa:

68. if __name__ == '__main__':

69. main ()

Essas linhas chamam a função main () se o programa foi executado por si próprio em vez de ser importado por outro programa que deseja usar suas

funções encryptMessage () e decryptMessage () .

Resumo

Você está perto do final deste livro, mas perceba que a cifra de Vigenère não é muito mais complicada do que a cifra de César, que foi um dos primeiros programas de codificação que você aprendeu. Com apenas algumas mudanças na cifra de César, criamos uma cifra que tem teclas exponencialmente mais possíveis do que as que podem ser forçadas a brutar.

A cifra de Vigenère não é vulnerável ao ataque de padrão de palavras do dicionário que o programa de hackers de substituição simples usa. Por centenas de anos, a cifra de Vigenère "indecifrável" manteve as mensagens em segredo, mas essa cifra também se tornou vulnerável. Nos [Capítulos 19 e 20](#), você aprenderá técnicas de análise de freqüência que lhe permitirão hackear a cifra de Vigenère.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/> .

1. Qual cifra é a cifra Vigenère semelhante, exceto que a cifra Vigenère usa várias chaves em vez de apenas uma chave?
2. Quantas chaves possíveis existem para uma chave Vigenère com um comprimento de chave de 10?
 1. Centenas
 2. Milhares
 3. Milhões
 4. Mais de um trilhão
3. Que tipo de cifra é a cifra de Vigenère?

19

ANÁLISE DE FREQUÊNCIA

“O talento inefável para encontrar padrões no caos não pode fazer o que quer que seja, a menos que ele mergulhe primeiro no caos. Se eles contêm padrões, ele não os vê agora, de qualquer maneira racional. Mas pode haver alguma parte sub-mental de sua mente que pode ir trabalhar...”

-Neal Stephenson, Cryptonomicon



Neste capítulo, você aprenderá a determinar a frequência de cada letra em inglês em um texto específico. Em seguida, você comparará essas frequências com as frequências de letras de seu texto cifrado para obter informações sobre o texto original, o que ajudará a quebrar a criptografia. Esse processo de determinar com que frequência uma letra aparece em textos simples e em textos cifrados é chamado de *análise de frequência*. Entender a análise de freqüência é um passo importante na invasão da cifra Vigenère. Usaremos a análise de freqüência de letras para quebrar a cifra de Vigenère no [Capítulo 20](#).

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Carta de frequência e ETAOIN
- A chave do método sort () e os argumentos de palavra-chave reversos
- Passando funções como valores em vez de chamar funções
- Convertendo dicionários em listas usando os métodos keys () , values () e items ()

Analisando a Frequênciа de Letras no Texto

Quando você joga uma moeda, cerca de metade do tempo sobe cara e metade do tempo que sobra coroa. Ou seja, a *frequência* de cara e coroa deve ser a mesma. Podemos representar a frequência como uma porcentagem dividindo o número total de vezes que um evento ocorre (quantas vezes invertemos as cabeças, por exemplo) pelo número total de tentativas em um evento (que é o número total de vezes que invertemos a moeda) e multiplicando o quociente por 100. Podemos aprender muito sobre uma moeda a partir de sua frequência de cara ou coroa: se a moeda é justa ou injustamente ponderada ou mesmo se é uma moeda de duas cabeças.

Também podemos aprender muito sobre um texto cifrado da frequência de suas letras. Algumas letras no alfabeto Inglês são usadas com mais frequência que outras. Por exemplo, as letras E, T, A e O aparecem com mais frequência em

palavras inglesas, enquanto as letras J, X, Q e Z aparecem com menos frequência em inglês. Usaremos essa diferença nas freqüências de letras no idioma inglês para violar mensagens criptografadas pelo Vigenère.

[A Figura 19-1](#) mostra as frequências de letras encontradas no inglês padrão. O gráfico foi compilado usando textos de livros, jornais e outras fontes.

Quando classificamos essas frequências de letras na ordem da maior frequência para o menor, E é a letra mais frequente, seguida por T, depois A e assim por diante, conforme mostrado na [Figura 19-2](#).

As seis letras mais frequentes em inglês são ETAOIN. A lista completa de cartas ordenadas por frequência é ETAOINSHRDLCUMWFGYPBVKJXQZ.

Lembre-se de que a cifra de transposição criptografa as mensagens organizando as letras do texto original em inglês em uma ordem diferente. Isso significa que as frequências das letras no texto cifrado não são diferentes daquelas no texto original. Por exemplo, E, T e A devem aparecer com mais frequência do que Q e Z em um texto cifrado de transposição.

Da mesma forma, as letras que aparecem mais frequentemente em um texto cifrado de César e um simples texto cifrado de substituição são mais provavelmente criptografadas a partir das letras inglesas mais comumente encontradas, como E, T ou A. Da mesma forma, as letras que aparecem com menos frequência É mais provável que o texto cifrado tenha sido criptografado para X, Q e Z em texto simples.

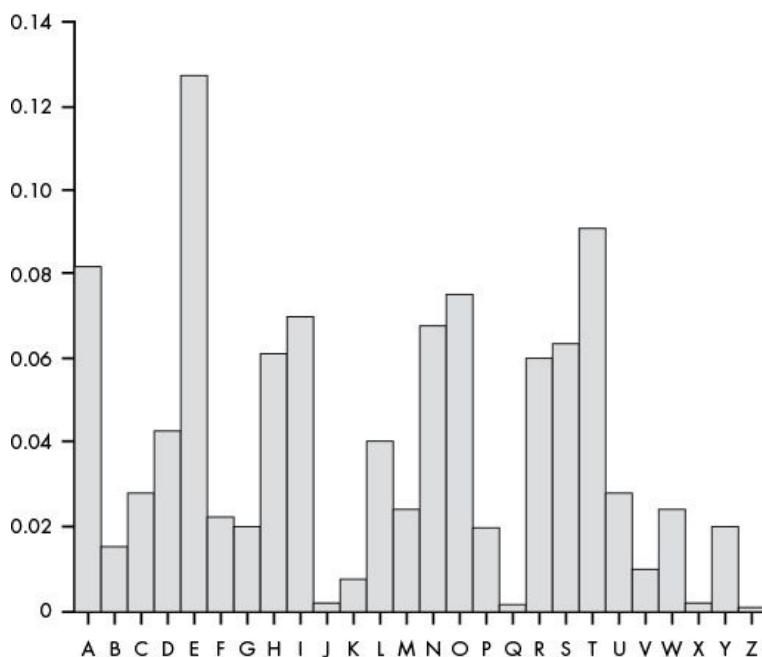


Figura 19-1: Análise de frequência de cada letra em texto típico em inglês

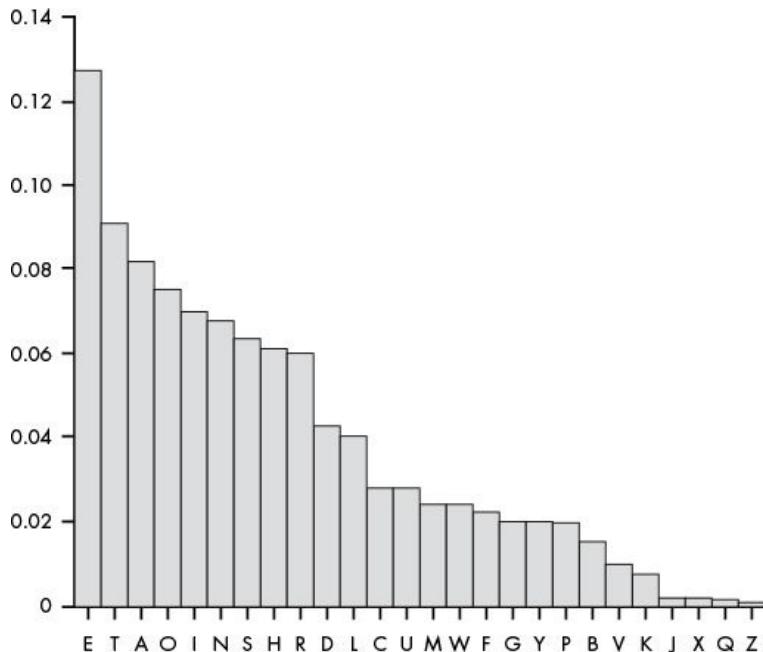


Figura 19-2: Cartas mais freqüentes e menos frequentes em texto típico em inglês

A análise de freqüência é muito útil ao hackear a cifra de Vigenère, porque nos permite forçar brutalmente cada subchave, uma de cada vez. Por exemplo, se uma mensagem foi criptografada com a chave PIZZA, precisaríamos forçar 26⁵ ou 11.881.376 chaves para encontrar a chave inteira de uma só vez. Para forçar apenas uma das cinco subchaves, no entanto, precisamos apenas de 26 possibilidades. Fazer isso para cada uma das cinco subchaves significa que precisamos apenas de força bruta de 26×5 ou 130 subchaves.

Usando a chave PIZZA, cada quinta letra da mensagem que começa com a primeira letra será criptografada com P, a cada quinta letra começando com a segunda letra com I e assim por diante. Podemos usar força bruta para a primeira subchave descriptografando cada quinta letra no texto cifrado com todas as 26 subchaves possíveis. Para a primeira subchave, descobriríamos que P produzia letras descriptografadas que correspondiam à frequência de letras do inglês mais do que as outras 25 subchaves possíveis. Este seria um forte indicador de que P foi a primeira subchave. Poderíamos então repetir isso para as outras subchaves até termos a chave inteira.

Correspondência de correspondências de letras

Para encontrar as frequências das letras em uma mensagem, usaremos um

algoritmo que simplesmente ordena as letras em uma cadeia de caracteres pela frequência mais alta para a frequência mais baixa. Em seguida, o algoritmo usa essa sequência ordenada para calcular o que este livro chama de *pontuação de correspondência de frequência*, que usaremos para determinar a similaridade da frequência de letras de uma sequência à do inglês padrão.

Para calcular a pontuação de correspondência de frequência para um texto cifrado, começamos com 0 e, em seguida, adicionamos um ponto cada vez que uma das letras inglesas mais frequentes (E, T, A, O, I, N) aparece entre as seis letras mais freqüentes do texto cifrado. Também adicionaremos um ponto à partitura toda vez que uma das letras menos frequentes (V, K, J, X, Q ou Z) aparecer entre as seis letras menos frequentes do texto cifrado.

A pontuação de correspondência de frequência para uma string pode variar de 0 (a frequência de letras da string é completamente diferente da frequência de letras em inglês) a 12 (a freqüência de letras da string é idêntica à do inglês normal). Conhecer a pontuação de correspondência de frequência de um texto cifrado pode revelar informações importantes sobre o texto original.

Calculando a pontuação da correspondência de frequência para a cifra de substituição simples

Usaremos o seguinte texto cifrado para calcular a pontuação de correspondência de frequência de uma mensagem criptografada usando a cifra de substituição simples:

Syl nlx sr pyyacao l ylwj eiswi upar lulsxrj isr srxjsxwjr, ia esmm
rwctjsxsza sj wmpframh, lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia esmm
caytra

jp famsaqa sj. Sy, px jia pjiac ilxo, ia sr pyyacao rpnajisxu eiswi lyppcor
l calppx ypc lwjsxu sx lwwpcolxwa jp isr srxjsxwjr, ia esmm lwwabj sj aqax
px jia rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.

-Facjclxo Ctrramm

Quando contamos a frequência de cada letra neste texto cifrado e classificamos da frequência mais alta para a mais baixa, o resultado é ASRXJILPWMCYOUEQNTHBFZGKVD. A é a letra mais frequente, S é a segunda letra mais frequente e assim sucessivamente para a letra D, que aparece com menos frequência.

Das seis letras mais freqüentes neste exemplo (A, S, R, X, J e I), duas dessas

letras (A e I) também estão entre as seis letras que aparecem com mais frequência no idioma inglês, que são E , T, A, O, I e N. Portanto, adicionamos dois pontos à pontuação de correspondência de frequência.

As seis letras menos frequentes no texto cifrado são F, Z, G, K, V e D. Três dessas letras (Z, K e V) aparecem no conjunto de letras que ocorrem com menor frequência, que são V, K, J, X, Q e Z. Então, adicionamos mais três pontos à pontuação. Com base na ordem de freqüência derivada desse texto cifrado, ASRXJILPWMCYOUEQNTHBFZGKVD, a pontuação de correspondência de frequência é 5, conforme mostrado na [Figura 19-3](#) .

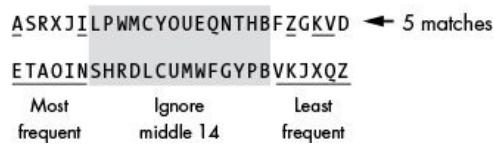


Figura 19-3: Cálculo da pontuação de correspondência de frequência da cifra de substituição simples

O texto cifrado criptografado usando uma simples cifra de substituição não terá uma pontuação de correspondência de frequência muito alta. As frequências de letras de um texto cifrado de substituição simples não correspondem às do inglês regular porque as letras de texto simples são substituídas uma por uma pelas cifradas. Por exemplo, se a letra T é criptografada para a letra J, então J seria mais provável de aparecer com freqüência no texto cifrado, mesmo que seja uma das letras que aparecem com menos frequência em inglês.

Calculando a pontuação da correspondência de frequência para a cifra de transposição

Desta vez, vamos calcular a pontuação de correspondência de frequência para um texto cifrado criptografado usando a cifra de transposição:

"Eu rc ascwuiluhnviwuetnh, osgaa gelo tipeeeee slnatsfietgi tittyne cenisl. E fnc isltn sn oa anos sd onisli, l erglei trhfmwfrogotn, l stcofiit.
aea wesn, lnc ee w, l elh eeehoer ros iol er snh nl oahsts ilasvih tvfeh
rtira id thatnie.im ei-dlmf eu thszonsisehroe, aiehcdsanahiec gv gyedsB
affahiecesd d lee onsdihsoc nin cethiTix eRneahgin re teom fbiotd n
ntacscwevhtdhnhpiwru "

As letras mais frequentes a menos frequentes neste texto cifrado são EISNTHAOCLRFDGWVMUYBPZXQJK. E é a carta mais frequente, eu é a segunda letra mais frequente e assim por diante.

As quatro letras que aparecem com mais frequência neste texto cifrado (E, I, N e T) também estão entre as letras mais frequentes no padrão Inglês (ETAOIN). Da mesma forma, as cinco letras menos frequentes no texto cifrado (Z, X, Q, J e K) também aparecem em VKJXQZ, resultando em uma pontuação total de correspondência de frequência de 9, conforme mostrado na [Figura 19-4](#).



Figura 19-4: Cálculo da pontuação da correspondência de frequência da cifra de transposição

O texto cifrado criptografado usando uma cifra de transposição deve ter uma pontuação de correspondência de frequência muito maior do que um simples texto cifrado de substituição. A razão é que, ao contrário da simples cifra de substituição, a cifra de transposição usa as mesmas letras encontradas no texto original, mas organizadas em uma ordem diferente. Portanto, a frequência de cada letra permanece a mesma.

Usando Análise de Frequência na Cifra Vigenère

Para hackar a cifra Vigenère, precisamos descriptografar as subchaves individualmente. Isso significa que não podemos confiar no uso da detecção de palavras em inglês, porque não poderemos descriptografar o suficiente da mensagem usando apenas uma subchave.

Em vez disso, descriptografamos as letras criptografadas com uma subchave e executamos a análise de frequência para determinar qual texto cifrado descriptografado produz uma frequência de letras que mais se aproxima do inglês normal. Em outras palavras, precisamos descobrir qual descriptografia tem a pontuação de correspondência de frequência mais alta, o que é uma boa indicação de que encontramos a subchave correta.

Repetimos esse processo para a segunda, terceira, quarta e quinta subchave também. Por enquanto, estamos apenas supondo que o comprimento da chave seja de cinco letras. (No [Capítulo 20](#), você aprenderá a usar o exame Kasiski para determinar o tamanho da chave.) Como há 26 descriptografias para cada subchave (o número total de letras no alfabeto) na codificação Vigenère, o computador só precisa executar $26 + 26 + 26 + 26 + 26$, ou 156, descriptografia para uma chave de cinco letras. Isto é muito mais fácil do que executar

descrições para cada combinação de subchaves possível, o que totalizaria 11.881.376 decifrações ($26 \times 26 \times 26 \times 26 \times 26$)!

Há mais etapas para hackar a cifra de Vigenère, que você aprenderá no [Capítulo 20](#) quando escrevermos o programa de hackers. Por enquanto, vamos escrever um módulo que realize análise de frequência usando as seguintes funções úteis:

`getLetterCount()` recebe um parâmetro de string e retorna um dicionário que conta quantas vezes cada letra aparece na string

`getFrequencyOrder()` Recebe um parâmetro de string e retorna uma string das 26 letras ordenadas da mais frequente para a menos frequente no parâmetro de string

`portugueseFreqMatchScore()` Obtém um parâmetro de string e retorna um inteiro de 0 a 12, indicando a pontuação de correspondência de freqüência de uma letra

Código Fonte para Frequências de Correspondência de Correspondência

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos, salve-o como *freqAnalysis.py* e certifique-se de que *pyperclip.py* esteja no mesmo diretório. Pressione F5 para executar o programa.

freqAnalysis.py

```
1. Localizador de Freqüência
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'
5. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
6
7. def getLetterCount(mensagem):
8.     # Retorna um dicionário com chaves de letras únicas e valores do
9.     # contagem de quantas vezes eles aparecem no parâmetro de mensagem:
10.    letterCount = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0,
11.                  'G': 0, 'H': 0, 'I': 0, 'J': 0, 'K': 0, 'L': 0, 'M': 0, 'N': 0,
12.                  'O': 0, 'P': 0, 'Q': 0, 'R': 0, 'S': 0, 'T': 0, 'U': 0, 'V': 0,
13.                  'W': 0, 'X': 0, 'Y': 0, 'Z': 0}
```

```
12. para carta em message.upper ():
13. se carta em letras:
14. letterCount [carta] += 1
15
16. return letterCount
17
18
19. def getItemAtIndexZero (itens):
20. itens de retorno [0]
21
22
23. def getFrequencyOrder (mensagem):
24. # Retorna uma string das letras do alfabeto dispostas na ordem da maioria
25. # freqüentemente ocorrendo no parâmetro de mensagem.
26
27. # Primeiro, pegue um dicionário de cada letra e sua contagem de frequência:
28. letterToFreq = getLetterCount (mensagem)
29
30. # Segundo, faça um dicionário de cada contagem de frequência para a (s)
letra (s)
31. # com essa frequência:
32. freqToLetter = {}
33. para cartas em LETRAS:
34. se letterToFreq [letra] não está em freqToLetter:
35. freqToLetter [letterToFreq [letra]] = [letra]
36. else:
37. freqToLetter [letterToFreq [carta]]. Append (carta)
38
39. # Terceiro, coloque cada lista de letras na ordem inversa "ETAOIN", e depois
40. # converta para uma string:
41. para freq em freqToLetter:
42. freqToLetter [freq] .sort (chave = ETAOIN.find, reverse = True)
43. freqToLetter [freq] = ".join (freqToLetter [freq])
44
45. # Quarta, converta o dicionário freqToLetter em uma lista de
46. # pares de tupla (chave, valor) e, em seguida, classificá-los:
47. freqPairs = list (freqToLetter.items ())
```

```
48. freqPairs.sort (key = getItemAtIndexZero, reverse = True)
49.
50. # Quinto, agora que as letras são ordenadas por frequência, extraia todas
51. # as letras da string final:
52. freqOrder = []
53. para o freqPair no freqPairs:
54. freqOrder.append (freqPair [1])
55
56. return " ".join (freqOrder)
57
58
59. def portugueseFreqMatchScore (message):
60. # Retorna o número de correspondências que a string na mensagem
61. O parâmetro # tem quando sua freqüência de letras é comparada ao inglês
62. # freqüência de letras. Um "jogo" é quantos dos seis mais frequentes
63. # e seis letras menos frequentes estão entre as seis mais frequentes e
64. # seis letras menos frequentes para o inglês.
65. freqOrder = getFrequencyOrder (mensagem)
66
67. matchScore = 0
68. # Encontre quantas correspondências para as seis letras mais comuns
existem:
69. para commonLetter in ETAOIN [: 6]:
70. se commonLetter em freqOrder [: 6]:
71. matchScore += 1
72. # Encontre quantas correspondências para as seis letras menos comuns que
existem:
73. para o uncommonLetter em ETAOIN [-6:]:
74. se uncommonLetter em freqOrder [-6:]:
75. matchScore += 1
76
77. return matchScore
```

Armazenando as letras na ordem ETAOIN

A linha 4 cria uma variável denominada ETAOIN , que armazena as 26 letras do alfabeto ordenadas de mais a menos frequentes:

1. Localizador de Freqüência

2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)

3

4. ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'

Naturalmente, nem todo texto em inglês reflete essa ordem exata de frequência. Você poderia facilmente encontrar um livro que tenha um conjunto de freqüências de letras em que Z é usado com mais frequência que Q. Por exemplo, o romance *Gadsby* de Ernest Vincent Wright nunca usa a letra E, que lhe dá um conjunto ímpar de freqüências de letras. Mas na maioria dos casos, inclusive em nosso módulo, a ordem ETAOIN deve ser precisa o suficiente.

O módulo também precisa de uma cadeia de todas as letras maiúsculas em ordem alfabética para algumas funções diferentes, portanto, definimos a variável constante LETTERS na linha 5.

5. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

LETTERS serve o mesmo propósito que as variáveis SYMBOLS fizeram em nossos programas anteriores: fornecer um mapeamento entre letras de strings e índices inteiros.

Em seguida, veremos como a função getLettersCount () conta a frequência de cada letra armazenada na sequência de mensagens .

Contando as Cartas em uma Mensagem

A função getLetterCount () pega a string de mensagem e retorna um valor de dicionário cujas chaves são cadeias de letras maiúsculas simples e cujos valores são inteiros que armazenam o número de vezes que a letra ocorre no parâmetro de mensagem .

A linha 10 cria a variável letterCount atribuindo a ela um dicionário que tem todas as chaves configuradas para um valor inicial de 0 :

7. def getLetterCount (mensagem):

8. # Retorna um dicionário com chaves de letras únicas e valores do

9. # contagem de quantas vezes eles aparecem no parâmetro de mensagem:

10. letterCount = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0,
'G': 0, 'H': 0, 'T': 0, 'J': 0, 'K': 0, 'L': 0, 'M': 0, 'N': 0,
'O': 0, 'P': 0, 'Q': 0, 'R': 0, 'S': 0, 'T': 0, 'U': 0, 'V': 0,
'W': 0, 'X': 0, 'Y': 0, 'Z': 0}

Nós incrementamos os valores associados às chaves até que elas representem as

contagens de cada letra, verificando cada caractere na mensagem usando um loop for na linha 12.

12. para carta em message.upper ():

13. se carta em letras:

14. letterCount [carta] += 1

O loop for percorre cada caractere na versão maiúscula da mensagem e atribui o caractere à variável letter . Na linha 13, verificamos se o caractere existe na string LETTERS , porque não queremos contar os caracteres que não são letras na mensagem . Quando a letra faz parte da cadeia LETTERS , a linha 14 incrementa o valor em letterCount [letter] .

Depois que o loop for na linha 12 terminar, o dicionário letterCount na linha 16 deve ter uma contagem mostrando com que frequência cada letra aparece na mensagem . Este dicionário é retornado de getLetterCount () :

16. return letterCount

Por exemplo, neste capítulo, usaremos a seguinte string (em https://en.wikipedia.org/wiki/Alan_Turing):

"" "Alan Mathison Turing era um matemático britânico, lógico, criptoanalista e computador

cientista. Ele foi altamente influente no desenvolvimento da ciência da computação, fornecendo

formalização dos conceitos de "algoritmo" e "computação" com a máquina de Turing. Turing

é amplamente considerado o pai da ciência da computação e da inteligência artificial. Durante

Segunda Guerra Mundial, Turing trabalhou para a Escola de Código e Cifra do Governo (GCHQ) em Bletchley Park,

Centro de quebra de código da Grã-Bretanha. Por um tempo ele foi chefe da Hut 8, a seção responsável pela

Criptanálise naval alemã. Ele inventou uma série de técnicas para quebrar cifras alemãs,

incluindo o método da bomba, uma máquina eletromecânica que poderia encontrar configurações

para a máquina Enigma. Depois da guerra, ele trabalhou no Laboratório Nacional de Física, onde

ele criou um dos primeiros projetos para um computador de programa

armazenado, o ACE. Em 1948, Turing se juntou ao Laboratório de Computação de Max Newman na Universidade de Manchester, onde ele ajudou no desenvolvimento dos computadores de Manchester e se interessou pela biologia matemática. Ele escreveu um artigo sobre a base química da morfogênese, e previu reações químicas oscilantes como a reação de Belousov-Zhabotinsky, que foi observada pela primeira vez na década de 1960. Turing's homossexualidade resultou em um processo criminal em 1952, quando os atos homossexuais ainda eram ilegal no Reino Unido. Ele aceitou tratamento com hormônios femininos (castração química) como alternativa à prisão. Turing morreu em 1954, pouco mais de duas semanas antes de completar 42 anos, de envenenamento por cianeto. Um inquérito determinou que sua morte foi suicídio; sua mãe e alguns outros acreditavam que sua morte foi acidental. Em 10 de setembro de 2009, após uma campanha na Internet, O primeiro-ministro britânico Gordon Brown fez um pedido oficial de desculpas em nome dos britânicos governo para "a maneira terrível que ele foi tratado". Em maio de 2012, a conta de um membro diante da Câmara dos Lordes, que concederia a Turing um perdão estatutário se fosse promulgada. "" "

Para esse valor de sequência, que tem 135 instâncias de A, 30 instâncias de B e assim por diante, getLetterCount () retornaria um dicionário semelhante a este:

```
{'A': 135, 'B': 30, 'C': 74, 'D': 58, 'E': 196, 'F': 37, 'G': 39, 'H': 87, 'I': 139, 'J': 2, 'K': 8, 'L': 62, 'M': 58, 'N': 122, 'O': 113, 'P': 36, 'Q': 2, 'R': 106, 'S': 89, 'T': 140, 'U': 37, 'V': 14, 'W': 30, 'X': 3, 'Y': 21, 'Z': 1}
```

Obtendo o primeiro membro de uma tupla

A função getItemAtIndexZero () na linha 19 retorna os itens no índice 0 quando uma tupla é passada para ela:

19. def getItemAtIndexZero (itens):

20. itens de retorno [0]

Mais tarde no programa, passaremos essa função para o método sort () para classificar as frequências das letras em ordem numérica. Analisaremos isso em detalhes em “ [Convertendo os itens do dicionário em uma lista classificável](#) ” na [página 275](#) .

Ordenando as Cartas na Mensagem por Freqüência

A função getFrequencyOrder () usa uma string de mensagem como argumento e retorna uma string com as 26 letras maiúsculas do alfabeto organizadas pela frequência com que aparecem no parâmetro de mensagem . Se a mensagem for legível Inglês, em vez de rabiscos aleatórios, é provável que essa string seja semelhante, se não idêntica, à string na constante ETAOIN . O código na função getFrequencyOrder () faz a maior parte do trabalho de calcular a pontuação de correspondência de frequência de uma string, que usaremos no programa de hackers da Vigenère no [Capítulo 20](#) .

Por exemplo, se passarmos a string "" "Alan Mathison Turing ..." para "getFrequencyOrder () , a função retornará a string 'ETIANORSHCLMDGFUPBWYVKXQJZ' porque E é a letra mais comum nessa string, seguida por T, então Eu, depois A e assim por diante.

A função getFrequencyOrder () consiste em cinco etapas:

1. Contando as letras na string
2. Criando um dicionário de contagens de freqüência e listas de letras
3. Classificando as listas de letras em ordem ETAOIN reversa
4. Convertendo estes dados para uma lista de tuplas
5. Convertendo a lista na string final para retornar da função getFrequencyOrder ()

Vamos dar uma olhada em cada passo.

Contando as letras com getLetterCount ()

A primeira etapa de getFrequencyOrder () chama getLetterCount () na linha 28 com o parâmetro message para obter um dicionário, chamado letterToFreq , contendo a contagem de cada letra da mensagem :

23. def getFrequencyOrder (mensagem):
24. # Retorna uma string das letras do alfabeto dispostas na ordem da maioria

25. # freqüentemente ocorrendo no parâmetro de mensagem.
26
27. # Primeiro, pegue um dicionário de cada letra e sua contagem de frequência:
28. letterToFreq = getLetterCount (mensagem)

Se passarmos a string "" "Alan Mathison Turing ..." "" como o parâmetro de mensagem , a linha 28 atribuirá o seguinte valor de dicionário a letterToFreq :

```
{'A': 135, 'C': 74, 'B': 30, 'E': 196, 'D': 58, 'G': 39, 'F': 37, 'eu': 139,  
«H»: 87, «K»: 8, «J»: 2, «M»: 58, «L»: 62, «O»: 113, «N»: 122, «Q»: 2,  
«P»: 36, «S»: 89, «R»: 106, «U»: 37, «T»: 140, «W»: 30, «V»: 14, «Y»: 21,  
'X': 3, 'Z': 1}
```

Criando um Dicionário de Contagens de Frequência e Listas de Cartas

A segunda etapa de getFrequencyOrder () cria um dicionário, freqToLetter , cujas chaves são a contagem de frequência e cujos valores são uma lista de letras com essas contagens de frequência. Enquanto o dicionário letterToFreq mapeia chaves de letras para valores de frequência, o dicionário freqToLetter mapeia chaves de frequência para a lista de valores de letras, portanto, precisaremos inverter a chave e os valores no dicionário letterToFreq . Nós invertemos as chaves e valores porque múltiplos letras podem ter a mesma contagem de frequência: "B" e "W" têm uma contagem de frequência de 30 no nosso exemplo, por isso precisamos colocá-los em um dicionário que se parece com {30: ['B', 'W'] } porque as chaves do dicionário devem ser exclusivas. Caso contrário, um valor de dicionário que se pareça com {30: 'B', 30: 'W'} substituirá simplesmente um desses pares de valores-chave pelo outro.

Para criar o dicionário freqToLetter , a linha 32 cria primeiro um dicionário em branco:

30. # Segundo, faça um dicionário de cada contagem de frequência para a (s) letra (s)
31. # com essa frequência:
32. freqToLetter = {}
33. para cartas em LETRAS:
34. se letterToFreq [letra] não está em freqToLetter:
35. freqToLetter [letterToFreq [letra]] = [letra]
36. else:
37. freqToLetter [letterToFreq [carta]]. Append (carta)

A linha 33 faz um loop sobre todas as letras em LETTERS , e a instrução if na linha 34 verifica se a freqüência da letra, ou letterToFreq [letra] , já existe como uma chave em freqToLetter . Se não, a linha 35 adiciona essa chave com uma lista da letra como o valor. Se a freqüência da letra já existir como uma chave em freqToLetter , a linha 37 simplesmente anexará a letra ao final da lista em letterToFreq [letra] .

Usando o valor de exemplo de letterToFreq criado usando a string "" "Alan Mathison Turing ... " " " , o freqToLetter deve retornar algo assim:

```
{1: ['Z'], 2: ['J', 'Q'], 3: ['X'], 135: ['A'], 8: ['K'], 139: ['I '],  
140: ['T'], 14: ['V'], 21: ['Y'], 30: ['B', 'W'], 36: ['P'], 37: ['F' , 'VOCÊ'],  
39: ['G'], 58: ['D', 'M'], 62: ['L'], 196: ['E'], 74: ['C'], 87: ['H' ]  
89: ['S'], 106: ['R'], 113: ['O'], 122: ['N']}
```

Observe que as chaves do dicionário agora contêm as contagens de frequência e seus valores contêm listas de letras com essas frequências.

Classificando as listas de letras em ordem ETAOIN reversa

A terceira etapa de getFrequencyOrder () envolve classificar as seqüências de letras em cada lista de freqToLetter . Lembre-se que freqToLetter [freq] avalia uma *lista* de letras que tem uma contagem de frequência de freq . Usamos uma lista porque é possível que duas ou mais letras tenham a mesma contagem de frequência. Nesse caso, a lista teria sequências de caracteres compostas de duas ou mais letras.

Quando várias letras têm as mesmas contagens de frequência, queremos ordenar essas letras na ordem inversa, em comparação com a ordem em que aparecem na string ETAOIN . Isso torna a ordenação consistente e minimiza a probabilidade de aumentar a pontuação de correspondência de frequência por acaso.

Por exemplo, digamos que as contagens de freqüência para as letras V, I, N e K são todas iguais para uma string que estamos tentando pontuar. Digamos também que quatro letras na string têm contagens de freqüência mais altas que V, I, N e K e dezoito letras têm contagens de freqüência mais baixas. Vou usar o x como um espaço reservado para essas letras neste exemplo. [A Figura 19-5](#) mostra o que seria a colocação dessas quatro letras na ordem ETAOIN.

xxxxINVxxxxxxxxxxxxxxxxxx ← 2 matches

ETAOINSHRDLCUMWFGYPBVKJXQZ

Most frequent	Ignore middle 14	Least frequent
---------------	------------------	----------------

Figura 19-5: A pontuação de correspondência de frequência ganhará dois pontos se as quatro letras estiverem na ordem ETAOIN.

O I e N adicionam dois pontos à pontuação de correspondência de frequência nesse caso, pois I e N estão entre as seis principais letras mais frequentes, embora não apareçam com mais frequência do que V e K nesta sequência de exemplo. Como as pontuações de correspondência de frequência variam apenas de 0 a 12, esses dois pontos podem fazer uma grande diferença! Mas, ao colocar letras de frequência idêntica na ordem ETAOIN reversa, minimizamos as chances de marcar uma letra em excesso. [A Figura 19-6](#) mostra essas quatro letras na ordem ETAOIN reversa.

```
xxxxKVNIxxxxxxxxxxxxxx ← 0 matches
ETAOINSHRDLCUMWFGYPBVKJXQZ
  Most           Ignore          Least
  frequent       middle 14      frequent
```

Figura 19-6: A pontuação de correspondência de frequência não aumentará se as quatro letras estiverem em ordem ETAOIN reversa.

Organizando as letras na ordem ETAOIN reversa, evitamos aumentar artificialmente a pontuação de correspondência de frequência por meio de uma ordenação casual de I, N, V e K. Isso também é verdade se houver dezoito letras com contagens de frequência mais altas e quatro letras com frequência mais baixa. conta, conforme mostrado na [Figura 19-7](#).

```
xxxxxxxxxxxxxxxxxKVNIxxxx ← 0 matches
ETAOINSHRDLCUMWFGYPBVKJXQZ
  Most           Ignore          Least
  frequent       middle 14      frequent
```

Figura 19-7: Inverter a ordem ETAOIN para letras menos frequentes também evita o aumento da pontuação da partida.

A ordem de classificação inversa garante que K e V não correspondam a nenhuma das seis letras menos frequentes em inglês e, novamente, evita o aumento da pontuação da correspondência de frequência em dois pontos.

Para classificar cada valor de lista no dicionário freqToLetter na ordem ETAOIN reversa, precisaremos passar um método para a função sort () do Python. Vamos ver como passar uma função ou método para outra função.

Passando Funções como Valores

Na linha 42, em vez de chamar o método find (), passamos a find como um valor para a chamada do método sort () :

42. freqToLetter [freq] .sort (chave = ETAOIN.find, reverse = True)

Podemos fazer isso porque, no Python, as funções podem ser tratadas como valores. De fato, definir uma função chamada spam é o mesmo que armazenar a definição de função em uma variável denominada spam . Para ver um exemplo, insira o seguinte código no shell interativo:

```
>>> def spam ():  
... print ('Olá!')  
...  
>>> spam ()  
Olá!  
>>> ovos = spam  
>>> ovos ()  
Olá!
```

Neste código de exemplo, definimos uma função chamada spam () que imprime a string 'Hello!' . Isso também significa que a variável spam contém a definição da função. Então copiamos a função na variável spam para os ovos variáveis. Depois de fazer isso, podemos chamar ovos () assim como podemos chamar spam () ! Observe que a instrução de atribuição *não* inclui parênteses após o spam . Em caso afirmativo, em vez disso, ele *chamaria* a função spam () e definiria os ovos da variável para o valor de retorno que é avaliado a partir da função spam () .

Como funções são valores, podemos passá-las como argumentos em chamadas de função. Digite o seguinte no shell interativo para ver um exemplo:

```
>>> def doMath (func):  
... return func (10, 5)  
...  
>>> def adicionando (a, b):  
... retorna a + b  
...  
>>> def subtraindo (a, b):  
... retornar a - b  
...  
1 >>> doMath (adicionando)  
15  
>>> doMath (subtraindo)
```

Aqui definimos três funções: doMath () , adding () e subtracting () . Quando passamos a função adicionando à chamada doMath () ❶ , estamos atribuindo a adição à variável func , e func (10, 5) está chamando adding () e passando 10 e 5 para ela. Portanto, a chamada func (10, 5) é efetivamente a mesma que a adição de chamada (10, 5) . É por isso que doMath (adicionando) retorna 15 . Da mesma forma, quando passar subtraindo para a chamada doMath () , doMath (subtraindo) retorna 5 porque func (10, 5) é o mesmo que subtrair (10, 5) .

Passando uma função para o método sort ()

Passar uma função ou método para o método sort () nos permite implementar diferentes comportamentos de ordenação. Normalmente, sort () classifica os valores em uma lista em ordem alfabética:

```
>>> spam = ['C', 'B', 'A']
>>> spam.sort()
>>> spam
['A', 'B', 'C']
```

Mas se passarmos uma função (ou método) para o argumento key keyword, os valores na lista são classificados *pelo valor de retorno da função* quando cada valor na lista é passado para essa função. Por exemplo, também podemos passar o método de string ETAOIN.find () como a chave para uma chamada sort () , da seguinte maneira:

```
>>> ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'
>>> spam.sort(key=ETAOIN.find)
>>> spam
['A', 'C', 'B']
```

Quando passamos ETAOIN.find para o método sort () , em vez de classificar as strings em ordem alfabética, o método sort () primeiro chama o método find () em cada string para que ETAOIN.find ('A') , ETAOIN. find ('B') e ETAOIN.find ('C') retornam os índices 2 , 19 e 11 , respectivamente - a posição de cada string na string ETAOIN . Em seguida, sort () usa esses índices retornados, em vez das sequências originais 'A' , 'B' e 'C' , para classificar os itens na lista de spam . É por isso que as sequências 'A' , 'B' e 'C' são classificadas como 'A' , 'C' e 'B' , refletindo a ordem em que aparecem em ETAOIN .

Invertendo as listas de letras com o método sort ()

Para ordenar as letras na ordem ETAOIN reversa, primeiro precisamos classificá-las com base na string ETAOIN atribuindo ETAOIN.find à chave . Depois que o método tiver sido chamado em todas as letras, de modo que sejam todos os índices, o método sort () classificará as letras com base em seu índice numérico.

Normalmente, a função sort () ordena qualquer lista que seja chamada em ordem alfabética ou numérica, que é conhecida como *ordem crescente* . Para classificar os itens em *ordem decrescente*, que é em ordem alfabética inversa ou numérica reversa, passamos True para o argumento de palavra-chave reversa do método sort () .

Fazemos tudo isso na linha 42:

```
39. # Terceiro, coloque cada lista de letras na ordem inversa "ETAOIN", e depois  
40. # converta para uma string:  
41. para freq em freqToLetter:  
42. freqToLetter [freq] .sort (chave = ETAOIN.find, reverse = True)  
43. freqToLetter [freq] = " .join (freqToLetter [freq])
```

Lembre-se de que, neste ponto, freqToLetter é um dicionário que armazena contagens de frequência inteiras como suas chaves e listas de seqüências de letras como seus valores. As seqüências de letras na chave freq estão sendo classificadas, e não o próprio dicionário freqToLetter . Os dicionários não podem ser classificados porque não têm ordem: não há nenhum par de valores-chave "primeiro" ou "último", pois há itens de lista.

Usando o valor de exemplo "" "Alan Mathison Turing ..." "" para o freqToLetter novamente, quando o loop terminar, este seria o valor armazenado em freqToLetter :

```
{1: 'Z', 2: 'QJ', 3: 'X', 135: 'A', 8: 'K', 139: 'T', 140: 'T', 14: 'V',  
21 : 'Y', 30: 'BW', 36: 'P', 37: 'FU', 39: 'G', 58: 'MD', 62: 'L', 196: 'E',  
74: ' C ', 87: ' H ', 89: ' S ', 106: ' R ', 113: ' O ', 122: ' N '}
```

Observe que as strings para as teclas 30 , 37 e 58 são todas ordenadas na ordem ETAOIN reversa. Antes do loop executado, os pares de valores-chave eram assim: {30: ['B', 'W'], 37: ['F', 'U'], 58: ['D', 'M'] , ...} . Após o loop, eles devem ficar assim: {30: 'BW', 37: 'FU', 58: 'MD', ...} .

A chamada do método join () na linha 43 altera a lista de cadeias em uma única cadeia. Por exemplo, o valor em freqToLetter [30] é ['B', 'W'] , que é unido como

'BW' .

Classificando as listas de dicionários por frequência

A quarta etapa de getFrequencyOrder () é classificar as cadeias de caracteres do dicionário freqToLetter pela contagem de frequência e converter as cadeias em uma lista. Tenha em mente que, como os pares de valores-chave nos dicionários não são ordenados, um valor de lista de todas as chaves ou valores em um dicionário será uma lista de itens em ordem aleatória. Isso significa que também precisaremos classificar essa lista.

Usando os métodos keys () , values () e items () Dictionary

Os métodos de dicionário keys () , values () e items () convertem cada parte de um dicionário para um tipo de dados não-dicionário. Depois que um dicionário é convertido em outro tipo de dados, ele pode ser convertido em uma lista usando a função list () .

Digite o seguinte no shell interativo para ver um exemplo:

```
>>> spam = {'cats': 10, 'dogs': 3, 'mice': 3}
>>> spam.keys()
dict_keys(['ratos', 'gatos', 'cachorros'])
>>> lista (spam.keys())
['ratos', 'gatos', 'cães']
>>> lista (spam.values())
[3, 10, 3]
```

Para obter um valor de lista de todas as chaves em um dicionário, podemos usar o método keys () para retornar um objeto dict_keys que podemos passar para a função list () . Um método de dicionário semelhante chamado values () retorna um objeto dict_values . Esses exemplos nos fornecem uma lista das chaves do dicionário e uma lista de seus valores, respectivamente.

Para obter as duas chaves e os valores, podemos usar o método de dicionário items () para retornar um objeto dict_items , o que torna os pares de valor-chave em tuplas. Podemos então passar as tuplas para list () . Digite o seguinte no shell interativo para ver isso em ação:

```
>>> spam = {'cats': 10, 'dogs': 3, 'mice': 3}
>>> list (spam.items())
[('ratos', 3), ('gatos', 10), ('cachorros', 3)]
```

Ao chamar items () e list () , convertemos os pares de valores-chave do

dicionário de spam em uma lista de tuplas. Isso é exatamente o que precisamos fazer com o dicionário freqToLetter para que possamos classificar as seqüências de letras em ordem numérica por frequência.

Convertendo os itens do dicionário em uma lista classificável

O dicionário freqToLetter tem contagem de freqüência inteira como suas chaves e listas de seqüências de caracteres de uma letra como seus valores. Para classificar as cadeias em ordem de frequência, chamamos o método items () e a função list () para criar uma lista de tuplas dos pares de valores-chave do dicionário. Então nós armazenamos esta lista de tuplas em uma variável chamada freqPairs na linha 47:

```
45. # Quarto, converta o dicionário freqToLetter em uma lista de  
46. # pares de tupla (chave, valor) e, em seguida, classifique-os:  
47. freqPairs = list (freqToLetter.items ())
```

Na linha 48, passamos o valor da função getItemAtIndexZero que definimos anteriormente no programa para a chamada do método sort () :

```
48. freqPairs.sort (key = getItemAtIndexZero, reverse = True)
```

A função getItemAtIndexZero () obtém o primeiro item em uma tupla, que neste caso é o inteiro de contagem de frequência. Isso significa que os itens em freqPairs são classificados pela ordem numérica dos inteiros de contagem de frequência. A linha 48 também passa True para o argumento de palavra-chave reversa, de modo que as tuplas são ordenadas inversamente, da maior contagem de frequência para a menor.

Continuando com o exemplo " " "Alan Mathison Turing ..." , depois que a linha 48 for executada, este seria o valor de freqPairs :

```
[(196, 'E'), (140, 'T'), (139, 'I'), (135, 'A'), (122, 'N'), (113, 'O'),  
(106, 'R'), (89, 'S'), (87, 'H'), (74, 'C'), (62, 'L'), (58, 'MD'), (39, 'G'),  
(37, 'FU'), (36, 'P'), (30, 'BW'), (21, 'Y'), (14, 'V'), (8, 'K'), (3, 'X'),  
(2, 'QJ'), (1, 'Z')]
```

A variável freqPairs é agora uma lista de tuplas ordenadas das letras mais frequentes para as menos frequentes: o primeiro valor em cada tupla é um inteiro representando a contagem de frequência, e o segundo valor é uma string contendo as letras associadas a essa contagem de frequência.

Criando uma lista das letras ordenadas

A quinta etapa de getFrequencyOrder () é criar uma lista de todas as strings da lista classificada em freqPairs . Queremos acabar com um único valor de string cujas letras estão na ordem de sua frequência, por isso não precisamos dos valores inteiros em freqPairs . A variável freqOrder começa como uma lista em branco na linha 52, e o loop for na linha 53 anexa a string no índice 1 de cada tupla em freqPairs ao final de freqOrder :

```
50. # Quinto, agora que as letras são ordenadas por frequência, extraia todas as
51. # as letras da string final:
52. freqOrder = []
53. para freqPair em freqPairs:
54.     freqOrder.append (freqPair [1])
```

Continuando com o exemplo, após o loop da linha 53 ter terminado, o freqOrder deve conter ['E', 'T', 'I', 'A', 'N', 'O', 'R', 'S', 'H', 'C', 'L', 'MD', 'G', 'FU', 'P', 'BW', 'Y', 'V', 'K', 'X', 'QJ', 'Z'] como seu valor.

A linha 56 cria uma string da lista de strings no freqOrder juntando-as usando o método join () :

```
56. return " .join (freqOrder)
```

Para o exemplo "" "Alan Mathison Turing ..." "", getFrequencyOrder () retorna a string 'ETIANORSHCLMDGFUPBWYVKXQJZ' . De acordo com essa ordenação, E é a letra mais frequente na string de exemplo, T é a segunda letra mais frequente, eu é a terceira mais frequente e assim por diante.

Agora que temos a freqüência de letras da mensagem como um valor de string, podemos compará-la com o valor de string da frequência de letras do inglês ('ETAOINSHRDLCUMWFGYPBVKJXQZ') para ver o quanto elas correspondem.

Calculando a pontuação da correspondência de frequência da mensagem

A função englishFreqMatchScore () pega uma string por message e depois retorna um inteiro entre 0 e 12 representando a pontuação da correspondência de frequência da string. Quanto mais alta a pontuação, mais próxima a freqüência da letra na mensagem coincide com a freqüência do texto normal em inglês.

```
59. def englishFreqMatchScore (mensagem):
60. # Retorna o número de correspondências que a string na mensagem
```

61. # parâmetro tem quando sua freqüência de letras é comparada com o inglês
62. # carta de freqüência. Uma "correspondência" é quantas das seis mais freqüentes são
63. # e seis letras menos frequentes estão entre as seis mais freqüentes e
64. # seis letras menos frequentes para o inglês.
65. freqOrder = getFrequencyOrder (mensagem)

O primeiro passo no cálculo da pontuação de correspondência de frequência é obter a ordem de frequência da mensagem chamando a função `getFrequencyOrder ()`, que fazemos na linha 65. Armazenamos a sequência ordenada na variável `freqOrder`.

A variável `matchScore` começa em 0 na linha 67 e é incrementada pelo loop `for` começando na linha 69, que compara as seis primeiras letras da string `ETAOIN` e as seis primeiras letras do `freqOrder`, dando um ponto para cada letra que elas têm em comum:

67. `matchScore = 0`

68. # Encontre quantas correspondências para as seis letras mais comuns existem:

69. para `commonLetter` em `ETAOIN [: 6]`:

70. if `commonLetter` em `freqOrder [: 6]`:

71. `matchScore += 1`

Lembre-se de que a fatia `[: 6]` é igual a `[0: 6]`, portanto, as linhas 69 e 70 dividem as seis primeiras letras das cadeias `ETAOIN` e `freqOrder`, respectivamente. Se qualquer uma das letras E, T, A, O, I ou N também estiver nas seis primeiras letras da string `freqOrder`, a condição na linha 70 será True e a linha 71 incrementará `matchScore`.

As linhas 73 a 75 são semelhantes às linhas 69 a 71, exceto neste caso, elas verificam se as *últimas* seis letras na string `ETAOIN` (V, K, J, X, Q e Z) estão nas *últimas* seis letras no `freqOrder`. Se forem, o `scoreScore` é incrementado.

72. # Encontre quantas correspondências para as seis letras menos comuns existem:

73. para `uncommonLetter` em `ETAOIN [-6:]`:

74. if `uncommonLetter` in `freqOrder [-6:]`:

75. `matchScore += 1`

A linha 77 retorna o inteiro em `matchScore`:

77. return matchScore

Ignoramos as 14 letras no meio da ordem de frequência ao calcular a pontuação de correspondência de frequência. As frequências dessas letras do meio são muito semelhantes entre si para fornecer informações significativas.

Resumo

Neste capítulo, você aprendeu como usar a função `sort()` para classificar os valores da lista em ordem alfabética ou numérica e como usar os argumentos de chave reversa e chave para classificar os valores da lista de diferentes maneiras. Você aprendeu como converter dicionários em listas usando os métodos de dicionário `keys()`, `values()` e `items()`. Você também aprendeu que pode passar funções como valores em chamadas de função.

No [Capítulo 20](#), usaremos o módulo de análise de frequência que escrevemos neste capítulo para hackar a cifra de Vigenère!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. O que é análise de frequência?
2. Quais são as seis letras mais usadas em inglês?
3. O que a variável de spam contém depois de executar o código a seguir?

spam = [4, 6, 2, 8]

spam.sort(reverse = true)

4. Se a variável spam contiver um dicionário, como você pode obter um valor de lista das chaves no dicionário?

20

ATACANDO A CIGANEIRA DO VIGÉNIO

“A privacidade é um direito humano inerente e um requisito para manter a condição humana com dignidade e respeito.”

Bruce Schneier, criptógrafo, 2006



Existem dois métodos para hackar a cifra de Vigenère. Um método usa um ataque de *dicionário* de força bruta para tentar cada palavra no arquivo de dicionário como a chave Vigenère, que funciona somente se a chave for uma palavra em inglês, como RAVEN ou DESK. O segundo método, mais sofisticado, usado pelo matemático do século 19 Charles Babbage, funciona mesmo quando a chave é um grupo aleatório de letras, como VUWFE ou PNFJ. Neste capítulo, vamos escrever programas para hackear a cifra Vigenère usando ambos os métodos.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- Ataques de dicionário
- Exame Kasiski
- Fatores Calculadores
- O tipo de dados set e a função set ()
- O método da lista extend ()
- A função itertools.product ()

Usando um ataque de dicionário à força bruta da criptografia Vigenère

Primeiro usaremos o ataque do dicionário para hackear a cifra de Vigenère. O arquivo de *dicionário dictionary.txt* (disponível no site deste livro em <https://www.nostarch.com/crackingcodes/>) tem aproximadamente 45.000 palavras em inglês. Demora menos de cinco minutos para o meu computador executar todas essas descrições para uma mensagem do tamanho de um parágrafo longo. Isso significa que, se uma palavra em inglês for usada para criptografar um texto codificado Vigenère, o texto cifrado é vulnerável a um ataque de dicionário. Vamos dar uma olhada no código fonte de um programa que usa um ataque de dicionário para hackar a cifra Vigenère.

Código Fonte do Programa Vigenère Dictionary Hacker

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**.

Digite o seguinte código no editor de arquivos e salve-o como *vigenereDictionaryHacker.py* . Certifique-se de colocar os *arquivos detectEnglish.py* , *vigenereCipher.py* e *pyperclip.py* no mesmo diretório do arquivo *vigenereDictionaryHacker.py* . Em seguida, pressione F5 para executar o programa.

vigenere
Dicionário
Hacker.py

```
1. # Vigenere Cipher Dictionary Hacker
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. import detectEnglish, vigenereCipher, pyperclip
5
6. def main():
7.     ciphertext = """ Tzx isz eccjxkg nfq lol mys bbqq I lxcz. """
8.     hackedMessage = hackVigenereDictionary (texto cifrado)
9
10.    se hackedMessage! = Nenhum:
11.        print ('Copiando a mensagem hackeada para a área de transferência:')
12.        print (hackedMessage)
13.        pyperclip.copy (hackedMessage)
14.    mais:
15.        print ('Falha ao hackar criptografia')
16
17
18.    def hackVigenereDictionary (texto cifrado):
19.        fo = open ('dictionary.txt')
20.        palavras = fo.readlines ()
21.        fo.close ()
22
23.    por palavra em linhas:
24.    word = word.strip () # Remove a nova linha no final.
25.    decryptedText = vigenereCipher.decryptMessage (palavra, texto cifrado)
26.    if detectEnglish.isEnglish (decryptedText, wordPercentage = 40):
27.        # Verifique com o usuário para ver se a chave descriptografada foi
encontrada:
```

```
28. print ()  
29. print ('Possível quebra de criptografia:')  
30. print ('Chave' + str (palavra) + ':' + decryptedText [: 100])  
31. print ()  
32. print ('Digite D para terminar, ou apenas pressione Enter para continuar  
quebra:')  
33. response = input ('>')  
34  
35. if response.upper (). Startswith ('D'):  
36. return decryptedText  
37  
38. if __name__ == '__main__':  
39. main ()
```

Exemplo de Execução do Programa Vigenère Dictionary Hacker

Quando você executa o programa *vigenereDictionaryHacker.py* , a saída deve ficar assim:

Possível quebra de criptografia:

Chave ASTROLOGIA: Os recados não são os qnks que eu conto.

Digite D para terminar ou pressione Enter para continuar quebrando:

>

Possível quebra de criptografia:

ASTRONOMIA-chave: Os verdadeiros segredos não são os que eu conto.

Digite D para terminar ou pressione Enter para continuar quebrando:

> d

Copiando mensagem hackeada para a área de transferência:

Os verdadeiros segredos não são os que eu conto.

A primeira palavra-chave que o programa sugere (ASTROLOGY) não funciona, então o usuário pressiona enter para deixar o programa de hackers continuar até encontrar a chave de descriptografia correta (ASTRONOMY).

Sobre o Programa Vigenère Dictionary Hacker

Como o código-fonte do programa *vigenereDictionaryHacker.py* é semelhante aos programas de hackers anteriores neste livro, não vou explicá-lo linha por linha. Resumidamente, a função *hackVigenereDictionary ()* tenta usar cada palavra no arquivo do dicionário para descriptografar o texto cifrado, e quando o texto descriptografado se parece com o inglês (de acordo com o módulo

`detectEnglish`), ele imprime a descriptografia e avisa o usuário para sair ou continuar.

Note que este programa usa o método `readlines()` nos objetos de arquivo retornados de `open()` :

`20. palavras = fo.readlines()`

Ao contrário do método `read()` , que retorna o conteúdo completo do arquivo como uma única string, o método `readlines()` retorna uma lista de strings, em que cada string é uma única linha do arquivo. Como há uma palavra em cada linha do arquivo de dicionário, a variável de `palavras` contém uma lista de todas as palavras em inglês de *Aarhus a Zurique* .

O resto do programa, das linhas 23 a 36, funciona de forma semelhante ao programa de transposição de cifra-hacking no [Capítulo 12](#) . Um loop for iterará sobre cada palavra na lista de `palavras` , descriptografará a mensagem com a palavra como chave e, em seguida, chamará `detectEnglish.isEnglish()` para ver se o resultado é um texto em inglês comprehensível.

Agora que escrevemos um programa que hackea a cifra Vigenère usando um ataque de dicionário, vamos ver como hackar a cifra de Vigenère mesmo quando a chave é um grupo aleatório de letras em vez de uma palavra do dicionário.

Usando o Kasiski Examination para encontrar o tamanho da chave

O exame Kasiski é um processo que podemos usar para determinar o tamanho da chave Vigenère usada para criptografar um texto cifrado. Podemos, então, usar a análise de freqüência para quebrar cada uma das subchaves de forma independente. Charles Babbage foi a primeira pessoa a ter quebrado a cifra de Vigenère usando esse processo, mas ele nunca publicou seus resultados. Seu método foi publicado mais tarde por Friedrich Kasiski, um matemático do início do século XX que se tornou o homônimo do método. Vamos dar uma olhada nos passos envolvidos no exame de Kasiski. Estes são os passos que o nosso programa de hackers Vigenère vai seguir.

Encontrando Seqüências Repetidas

O primeiro passo do exame de Kasiski é encontrar todos os conjuntos repetidos de pelo menos três letras no texto cifrado. Essas seqüências repetidas poderiam ser as mesmas letras de texto simples criptografadas usando as mesmas subchaves da chave Vigenère. Por exemplo, se você criptografou o texto sem formatação O GATO ESTÁ FORA DO SACO com a chave SPILLTHEBEANS,

você obteria:

O CATISOUTOF O SACO
SPI LLTHEBEANS SPI LLT
LWM NLMPWPYTBX LWM MLZ

Observe que as letras LWM repetem duas vezes. A razão é que, no texto cifrado, o LWM é a palavra de texto simples que é criptografada usando as mesmas letras da chave - SPI - porque a chave se repete no segundo. O número de letras desde o início do primeiro LWM até o início do segundo LWM, que chamaremos de *espaçamento*, é 13. Isso sugere que a chave usada para este texto cifrado é de 13 letras. Apenas olhando as seqüências repetidas, você pode descobrir o tamanho da chave.

No entanto, na maioria dos textos cifrados, a chave não se alinha convenientemente com uma seqüência repetida de letras, ou a chave pode repetir mais de uma vez entre seqüências repetidas, significando que o número de letras entre as letras repetidas seria igual a um múltiplo do tamanho da chave em vez da própria chave. Para tentar resolver esses problemas, vamos examinar um exemplo mais longo em que não sabemos qual é a chave.

Quando removemos as não-letras no texto cifrado PPQCA XQVEKG YBNKMAZU YBNGBAL JON I TSZM JYIM VRAG VOHT VRAU C TKSG. Se parecer com a string mostrada na [Figura 20-1](#), será semelhante à string mostrada na [Figura 20-1](#). A figura também mostra as seqüências repetidas nesta string - VRA, AZU e YBN - e o número de letras entre cada par de seqüências.

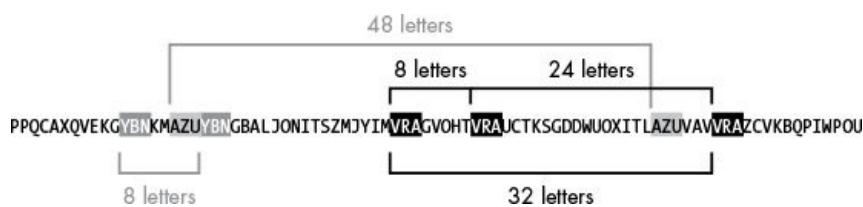


Figura 20-1: As seqüências repetidas na string de exemplo

Neste exemplo, existem vários comprimentos de chave possíveis. O próximo passo do exame Kasiski é calcular todos os fatores dessas contagens para diminuir os possíveis comprimentos de chave.

Obtendo Fatores de Espaçamentos

Os espaçamentos entre as seqüências são 8, 8, 24, 32 e 48 no exemplo. Mas os fatores dos espaçamentos são mais importantes que os espaçamentos.

Para ver o motivo, veja a mensagem THEDOGANDTHECAT na [Tabela 20-1](#) e tente criptografá-la com a tecla de nove letras ABCDEFGHI e a tecla de três letras XYZ. Cada tecla se repete pelo tamanho da mensagem.

Tabela 20-1: Criptografando THEDOGANDTHECAT com duas chaves diferentes

	Criptografando com ABCDEFGHI	Criptografando com XYZ
Mensagem de texto simples	O DOGAND THE CAT	O DOGAND THE CAT
Chave (repetindo)	ABC DEFGHI ABC DEF	XYZ XYZXYZ XYZ XYZ
Texto cifrado	TIG GSLGUL TIG FEY	QFD AMFXLC QFD ZYS

As duas chaves produzem dois textos cifrados diferentes, conforme esperado. É claro que o hacker não saberá a mensagem original ou a chave, mas verá no texto cifrado **TIG GSLGUL TIG FEY** que a sequência TIG aparece no índice 0 e no índice 9. Como $9 - 0 = 9$, o espaçamento entre estes seqüências é 9, o que parece indicar que a chave original era uma chave de nove letras; neste caso, essa indicação está correta.

No entanto, o texto codificado **QFD AMFXLC QFD ZYS** também produz uma sequência repetida (QFD) que aparece no índice 0 e no índice 9. O espaçamento entre essas sequências também é 9, indicando que a chave usada neste texto codificado também tinha nove letras. Mas sabemos que a chave tem apenas três letras: XYZ.

As sequências repetidas ocorrem quando as mesmas letras na mensagem (THE no nosso exemplo) são criptografadas com as mesmas letras da chave (ABC e XYZ no nosso exemplo), o que acontece quando as letras semelhantes na mensagem e na tecla “alinham” e criptografar para a mesma seqüência. Esse alinhamento pode acontecer em qualquer múltiplo do tamanho real da chave (como 3, 6, 9, 12 e assim por diante), e é por isso que a tecla de três letras pode produzir uma seqüência repetida com um espaçamento de 9.

Portanto, o comprimento da chave possível não se deve apenas ao espaçamento,

mas a qualquer fator desse espaçamento. Os fatores de 9 são 9, 3 e 1. Portanto, se você encontrar seqüências repetidas com um espaçamento de 9, você deve considerar que a chave poderia ser de comprimento 9 ou 3. Podemos ignorar 1 porque uma cifra de Vigenère com uma A chave do boletim é apenas a cifra de César.

O passo 2 do exame de Kasiski envolve encontrar cada um dos fatores de espaçamento (excluindo 1), como mostrado na [Tabela 20-2](#).

Tabela 20-2: Fatores de cada espaçamento

Espaçamento Fatores

8	2, 4, 8
24	2, 4, 6, 8, 12, 24
32	2, 4, 8, 16
48	2, 4, 6, 8, 12, 24, 48

Coletivamente, os números 8, 8, 24, 32 e 48 têm os seguintes fatores: 2, 2, 2, 2, 4, 4, 4, 4, 6, 6, 8, 8, 8, 12, 12, 16, 24, 24 e 48.

A chave é mais provável de ser os fatores que ocorrem com mais freqüência, o que você pode determinar contando. Como 2, 4 e 8 são os fatores de ocorrência mais freqüentes dos espaçamentos, eles são os comprimentos mais prováveis da chave de Vigenère.

Obtendo todas as enésimas letras de uma string

Agora que temos comprimentos possíveis da chave Vigenère, podemos usar essa informação para descriptografar a subchave de uma mensagem por vez. Para este exemplo, vamos supor que o tamanho da chave seja 4. Se não conseguirmos quebrar esse texto cifrado, podemos tentar novamente supondo que o tamanho da chave seja 2 ou 8.

Como a chave é percorrida para criptografar o texto original, um comprimento de chave de 4 significaria que, a partir da primeira letra, cada quarta letra do texto cifrado é criptografada usando a primeira subchave, toda quarta letra a

partir da segunda letra do texto original criptografado usando a segunda subchave e assim por diante. Usando essas informações, formaremos strings a partir do texto cifrado das letras que foram criptografadas pela mesma subchave. Primeiro vamos identifique o que cada quarta letra da string seria se começássemos de letras diferentes. Então, combinaremos as letras em uma única string. Nestes exemplos, vamos negrito a cada quarta letra.

Identifique cada quarta letra começando com a *primeira* letra:

P PQC A XQV E KGY B NKM A ZUY B NGB A LJO N SUA Z MJY A MVR
A GVO H O R A G U R A D E X A R
K BQP I WPO U

Em seguida, encontramos cada quarta letra começando com a *segunda* letra:

P P QCA X QVE K GYB N KMA Z UYB N GBA L JON I TSZ M JYI M VRA
VRA VRA U CTK S GDD W UOX I TLA Z UVA V VRA Z CV
K B QPI W POU

Então, fazemos o mesmo, começando pela *terceira* e *quarta* letra até chegarmos ao tamanho da subchave que estamos testando. [A Tabela 20-3](#) mostra as cadeias combinadas das letras em negrito para cada iteração.

Tabela 20-3: Cordas de cada quarta letra

Começando **Corda**
com

Primeira carta PAEBABANZIAHAKDXAAKIU

Segunda carta PXKNZNLIMMGTUSWIZVZBW

Terceira carta QQGKUGJTJVVCUTUVCQP

Quarta carta CVYMYBOSYRORTDOLVRVPO

Usando a análise de freqüência para quebrar cada subchave

Se adivinhamos o tamanho correto da chave, cada uma das quatro cadeias que criamos na seção anterior teria sido criptografada com uma subchave. Isso

significa que quando uma seqüência de caracteres é descriptografada com a subchave correta e passa pela análise de frequência, as letras descriptografadas provavelmente terão uma alta pontuação de correspondência de frequência em inglês. Vamos ver como esse processo funciona usando a primeira string, PAEBABANZIAHAKDXAAKIU , como exemplo.

Primeiro, descriptografamos a cadeia de caracteres 26 vezes (uma vez para cada uma das 26 subchaves possíveis) usando a função de descriptografia Vigenère no [Capítulo 18](#) , vigenereCipher.decryptMessage () . Em seguida, testamos cada string descriptografada usando a função de análise de frequência em inglês no [Capítulo 19](#) , freqAnalysis.englishFreqMatchScore () . Execute o seguinte código no shell interativo:

```
>>> import freqAnalysis, vigenereCipher  
>>> para subchave em 'ABCDEFGHIJKLMNPQRSTUVWXYZ':  
... decryptedMessage = vigenereCipher.decryptMessage (subchave,  
'PAEBABANZIAHAKDXAAKIU')  
... print (subchave, decryptedMessage,  
freqAnalysis.englishFreqMatchScore ( decryptedMessage ))  
...  
A PAEBABANZIAHAKDXAAKIU 2  
B OZDAZAZMYHZGZJCWZZJHT 1  
- recorte -
```

[A Tabela 20-4](#) mostra os resultados.

Tabela 20-4: Pontuação da Partida de Frequência em Inglês para Cada Decodificação

Sub-chave	Descriptografia	Pontuação do jogo de frequência em inglês
'UMA'	'PAEBABANZIAHAKDXAAKIU'	2
'B'	'OZDAZAZMYHZGZJCWZZJHT'	1
'C'	'NYCZYZYLXGYFYIBVYYYIGS'	1
'D'	'MXBYXXWKWXEXHAUXXXHFR'	0
'E'	'LWAXWXWJVVEWDWGZTWWGEQ'	1

'F'	'KVZWVWVIUDVCVFYSVVVFDP'	0
'G'	'JUYVUVUHTCUBUEXRUUUECO'	1
'H'	'ITXUTUTGSBTATDWQTTDBN'	1
'EU'	'HSWTSTSFRASZSCVPSSSCAM'	2
'J'	'GRVSRSREQZRYRBUORRRBZL'	0
'K'	'FQURQRQDPYQXQATNQQQAYK'	1
'EU'	'EPTQPQPCOXPWPZSMPPPZXJ'	0
'M'	'DOSPOPOBNWOVOYRLOOOYWI'	1
'N'	'CNRONONAMVNUNXQKNNNXVH'	2
'O'	'BMQNMNMZLUMTMWPJMMMWUG'	1
'P'	'ALPMLMLYKTLSLVOILLLVTF'	1
'Q'	'ZKOLKLKXJSRKUNHKKKUSE'	0
'R'	'YJNKJKJWIRJQJTMGJJTRD'	1
'S'	'XIMJIJIVHQIPISLFIIISQC'	1
'T'	'WHЛИIHUGPHOHRKEHHHRPB'	1
'VOCÊ'	'VGKHGHGTFOGNGQJDGGGQOA'	1
'V'	'UFJGFGFSENFMPICFFPNZ'	1
'W'	'TEIFEFERDMELEOHBEEEOMY'	2
'X'	'SDHEDEDQCLDKDNGADDDNLX'	2
'Y'	'RCGDCDCPBKCJCMFZCCCMKW'	0

'Z'	'QBFCBCBOAJBIBLEYBBBLJV'	0
-----	--------------------------	---

As subchaves que produzem decodificações com a frequência mais próxima correspondem ao inglês são provavelmente a subchave real. Na [Tabela 20-4](#), as subchaves 'A', 'T', 'N', 'W' e 'X' resultam nas pontuações de correspondência de frequência mais altas para a primeira string. Observe que essas pontuações são baixas em geral porque não há texto cifrado suficiente para nos dar uma grande amostra de texto, mas elas funcionam bem o suficiente para esse exemplo.

O próximo passo é repetir esse processo para as outras três strings para encontrar as subchaves mais prováveis. [A Tabela 20-5](#) mostra os resultados finais.

Tabela 20-5: Subchaves mais prováveis para as cadeias de exemplo

Cadeia de texto cifrado	Subchaves mais prováveis
-------------------------	--------------------------

PAEBABANZIAHAKDXAAAKIU A, eu, N, W, X

PXKNZNLIMMGKTUSWIZVZBW Eu, Z

QQGKUGJTJVVVCUTUVCQP C

CVYMYBOSYRORTDOLRVPO K, N, R, V, Y

Como há cinco subchaves possíveis para a primeira subchave, duas para a segunda subchave, uma para a terceira subchave e cinco para a quarta subchave, o número total de combinações é 50 (que obtemos da multiplicação de todas as subchaves possíveis $5 \times 2 \times 1 \times 5$). Em outras palavras, precisamos forçar 50 chaves possíveis. Mas isso é muito melhor do que o uso de força bruta através de $26 \times 26 \times 26 \times 26$ (ou 456.976) chaves possíveis, nossa tarefa não foi a de restringir a lista de possíveis subchaves. Esta diferença torna-se ainda maior se a chave Vigenère for mais longa!

Força Forçada Através das Possíveis Chaves

Para forçar a chave bruta, vamos tentar todas as combinações das subchaves prováveis. Todas as 50 combinações possíveis de subchaves são listadas da seguinte maneira:

AICK	IICK	NICK	WICK	XICK
AICN	IICN	NICN	WICN	XICN
AICR	IICR	NICR	WICR	XICR
AICV	IICV	NICV	WICV	XICV
AICY	IICY	NICY	WICY	XICY
AZCK	IZCK	NZCK	WZCK	XZCK
AZCN	IZCN	NZCN	WZCN	XZCN
AZCR	IZCR	NZCR	WZCR	XZCR
AZCV	IZCV	NZCV	WZCV	XZCV
AZCY	IZCY	NZCY	WZCY	XZCY

O passo final em nosso programa de hackers Vigenere será testar todas as 50 dessas chaves de decriptação no texto cifrado completo para ver qual produz texto em inglês legível. Isso deve revelar que a chave do texto cifrado “PPQCA XQVEKG...” é WICK.

Código Fonte do Programa Vigenère Hacking

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Certifique-se de que os arquivos *detectEnglish.py* , *freqAnalysis.py* , *vigenereCipher.py* e *pyperclip.py* estejam no mesmo diretório que o arquivo *vigenereHacker.py* . Em seguida, digite o seguinte código no editor de arquivos e salve-o como *vigenereHacker.py* . Pressione F5 para executar o programa.

O texto cifrado na linha 17 deste programa é difícil de copiar do livro. Para evitar erros de digitação, copie e cole-o no site do livro em <https://www.nostarch.com/crackingcodes/> . Você pode verificar se há diferenças entre o texto em seu programa e o texto do programa neste livro usando a ferramenta de comparação on-line no site do livro.

vigenere Hacker.py

1. # Vigenere Cipher Hacker
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. importar ferramentas, re
5. import vigenereCipher, pyperclip, freqAnalise, detectEnglish
- 6
7. LETRAS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
8. MAX_KEY_LENGTH = 16 # Não tentará chaves mais longas que isso.
9. NUM_MOST_FREQ LETTERS = 4 # Tente esta muitas letras por subchave.
10. SILENT_MODE = False # Se definido como True, o programa não imprime

nada.

11. NONLETTERS_PATTERN = re.compile ('[^ AZ]')

12

13

14. def main ():

15. # Em vez de digitar este texto cifrado, você pode copiá-lo e colá-lo

16. # de <https://www.nostarch.com/crackingcodes/>:

17. ciphertext = """ "Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuo, lgouqdaf, kdmktsvmztsl, izr xoexghzr kkusitaaf. Vz wsa twbhdg ubalmmzhdad qz

- recorte -

azmtmd'g widt ião bwnafz tzm Tcpsw wr Zjrvva ivdcz eaigd yzmbo
Tmzubb a kbmhptgzk dvrwz wa efiohzd. """ "

18. hackedMessage = hackVigenere (texto cifrado)

19

20. se hackedMessage! = Nenhum:

21. print ('Copiando mensagem hackeada para área de transferência:')

22. print (hackedMessage)

23. pyperclip.copy (hackedMessage)

24. else:

25. print ('Falha ao hackar criptografia')

26

27

28. def findRepeatSequencesSpacings (message):

29. # Passa pela mensagem e encontra quaisquer sequências de 3 a 5 letras

30. # que se repetem Retorna um dict com as teclas da sequência e

31. # valores de uma lista de espaçamentos (número de letras entre as
repetições).

32

33. # Use uma expressão regular para remover não letras da mensagem:

34. message = NONLETTERS_PATTERN.sub ("", message.upper ())

35

36. # Compile uma lista de seqüências seqLen-letter encontradas na mensagem:

37. seqSpacings = {} # Chaves são seqüências; valores são listas de
espaçamentos int.

38. para seqLen na faixa (3, 6):

39. para seqStart in range (len (mensagem) - seqLen):

```
40. # Determine qual é a sequência e armazene-a em seq:  
41. seq = mensagem [seqStart: seqStart + seqLen]  
42.  
43. # Procure esta sequência no resto da mensagem:  
44. para i no intervalo (seqStart + seqLen, len (mensagem) - seqLen):  
45. if mensagem [i: i + seqLen] == seq:  
46. # Encontrou uma sequência repetida:  
47. se não seq em seqSpacings:  
48. seqSpacings [seq] = [] # Inicializa a lista em branco.  
49.  
50. # Anexar a distância de espaçamento entre o repetido  
51. # sequência e a sequência original:  
52. seqSpacings [seq] .append (i - seqStart)  
53. return seqEspaços  
54  
55  
56. def getUsefulFactors (num):  
57. # Retorna uma lista de fatores úteis de num. Por "útil" queremos dizer fatores  
58. # menor que MAX_KEY_LENGTH + 1 e não 1. Por exemplo,  
59. # getUsefulFactors (144) retorna [2, 3, 4, 6, 8, 9, 12, 16].  
60  
61. if num <2:  
62. return [] # Números menores que 2 não possuem fatores úteis.  
63  
64. factors = [] # A lista de fatores encontrados.  
65  
66. # Ao encontrar fatores, você só precisa verificar os inteiros até  
67. # MAX_KEY_LENGTH:  
68. para i no intervalo (2, MAX_KEY_LENGTH + 1): # Não teste 1: não é útil.  
69. if num% i == 0:  
70. fatores.append (i)  
71. otherFactor = int (num / i)  
72. if otherFactor <MAX_KEY_LENGTH + 1 e otherFactor! = 1:  
73. factors.append (otherFactor)  
74. lista de retorno (set (fatores)) # Remove fatores duplicados.  
75  
76
```

```
77. def getItemAtIndexOne (x):
78.     retorno x [1]
79
80
81. def getMostCommonFactors (seqFactors):
82.     # Primeiro, conte quantas vezes um fator ocorre em seqFactors:
83.     factorCounts = {} # Key é um fator; valor é com que frequência ocorre.
84
85.     # seqFactores chaves são seqüências; valores são listas de fatores do
86.     # espaçamentos. seqFactors tem um valor como {'GFD': [2, 3, 4, 6, 9, 12,
87.     # 18, 23, 36, 46, 69, 92, 138, 207], 'ALW': [2, 3, 4, 6, ...], ...}.
88.     para seq em seqFactors:
89.         factorList = seqFactors [seq]
90.         para fator em factorList:
91.             se fator não estiver em factorCounts:
92.                 factorCounts [fator] = 0
93.                 factorCounts [fator] += 1
94
95.         # Segundo, coloque o fator e sua contagem em uma tupla e faça uma lista
96.         # dessas tuplas para podermos ordená-las:
97.         factorsByCount = []
98.         para fator em FactorCounts:
99.             # Excluir fatores maiores que MAX_KEY_LENGTH:
100.            if factor <= MAX_KEY_LENGTH:
101.                # factorsByCount é uma lista de tuplas: (factor, factorCount).
102.                # factorsByCount tem um valor como [(3, 497), (2, 487), ...].
103.                factorsByCount.append ((factor, factorCounts [fator]))
104
105.            # Classifique a lista pela contagem de fator:
106.            factorsByCount.sort (key = getItemAtIndexOne, reverso = Verdadeiro)
107.
108.        return factorsByCount
109
110.
111. def kasiskiExamination (texto cifrado):
112.     # Descubra as sequências de 3 a 5 letras que ocorrem várias vezes
113.     # no texto cifrado. repeatSeqSpacings tem um valor como
```

```
114. # {'EXG': [192], 'NAF': [339, 972, 633], ...}:
115. repeatSeqSpacings = findRepeatSequencesSpacings (texto cifrado)
116
117. # (Veja getMostCommonFactors () para uma descrição de seqFactors.)
118. seqFactors = {}
119. para seq em repeatSeqSpacings:
120.     seqFactors [seq] = []
121. para espaçamento em repeatSeqSpacings [seq]:
122.     seqFactors [seq] .extend (getUsefulFactors (espaçamento))
123
124. # (Veja getMostCommonFactors () para obter uma descrição de
factorsByCount.)
125. factorsByCount = getMostCommonFactors (seqFactors)
126
127. # Agora extraímos a contagem de fatores de factorsByCount e
128. # colocá-los em todos osLikelyKeyLengths para que eles sejam mais fáceis
de
129. # use depois:
130. allLikelyKeyLengths = []
131. para twoIntTuple em factorsByCount:
132.     allLikelyKeyLengths.append (twoIntTuple [0])
133
134. return allLikelyKeyLengths
135
136
137. def getNthSubkeysLetters (nth, keyLength, message):
138. # Retorna todas as enésimas letras para cada conjunto de letras de
comprimento no texto.
139. # Por exemplo, getNthSubkeysLetters (1, 3, 'ABCABCABC') retorna
'AAA'
140. # getNthSubkeysLetters (2, 3, 'ABCABCABC') retorna 'BBB'
141. # getNthSubkeysLetters (3, 3, 'ABCABCABC') retorna 'CCC'
142. # getNthSubkeysLetters (1, 5, 'ABCDEFGHI') retorna 'AF'
143
144. # Use uma expressão regular para remover letras que não sejam da
mensagem:
145. message = NONLETTERS_PATTERN.sub (", message.upper ())
```

146
147. i = nth - 1
148. letras = []
149. while i < len (mensagem):
150. letters.append (mensagem [i])
151. i += keyLength
152. return " ".join (letras)
153
154.
155. def attemptHackWithKeyLength (texto cifrado, mostLikelyKeyLength):
156. # Determine as letras mais prováveis para cada letra da chave:
157. ciphertextUp = ciphertext.upper ()
158. # allFreqScores é uma lista do número de listas mostLikelyKeyLength.
159. # Estas listas internas são as listas de freqScores:
160. allFreqScores = []
161. para enésimo intervalo (1, maisLikelyKeyComprimento + 1):
162. nthLetters = getNthSubkeysLetters (nth, mostLikelyKeyLength,
ciphertextUp)
163
164. # freqScores é uma lista de tuplas como
165. # [(<carta>, <Pontuação da partida em Eng. Freq.>], ...]
166. # Lista é classificada por pontuação da partida. Maior pontuação significa
melhor correspondência.
167. # Veja os comentários do englishFreqMatchScore () em freqAnalysis.py.
168. freqScores = []
169. para possívelKey em LETRAS:
170. decryptedText = vigenereCipher.decryptMessage (possívelKey,
nthLetters)
171. keyAndFreqMatchTuple = (possívelKey,
freqAnalysis.englishFreqMatchScore (decryptedText))
172. freqScores.append (keyAndFreqMatchTuple)
173. # Classificar por pontuação do jogo:
174. freqScores.sort (key = getItemAtIndexOne, reverse = True)
175
176. allFreqScores.append (freqScores [: NUM_MOST_FREQ LETTERS])
177
178. se não SILENT_MODE:

```
179. para i no intervalo (len (allFreqScores)):  
180. # Use i + 1 para que a primeira letra não seja chamada de letra "0":  
181. print ('Letras possíveis para a letra% s da chave:'% (i + 1),  
end = "")  
182. para o freqScore em todos os FreqScores [i]:  
183. print ('% s'% freqScore [0], end = "")  
184. print () # Imprime uma nova linha.  
185.  
186. # Tente cada combinação das letras mais prováveis para cada posição  
187. # na chave:  
188. para índices em itertools.product (range  
(NUM_MOST_FREQ LETTERS),  
repeat = mostLikelyKeyLength):  
189. # Crie uma chave possível a partir das letras em todos osFreqScores:  
190. possívelKey = ""  
191. para i no intervalo (mostLikelyKeyLength):  
192. possívelKey += allFreqScores [i] [índices [i]] [0]  
193  
194. se não SILENT_MODE:  
195. print ("Tentativa com chave:% s'% (possívelKey))  
196  
197. decryptedText = vigenereCipher.decryptMessage (possívelKey,  
ciphertextUp)  
198  
199. if detectEnglish.isEnglish (decryptedText):  
200. # Defina o texto cifrado hackeado para o invólucro original:  
201. origCase = []  
202. para i na faixa (len (texto cifrado)):  
203. se texto cifrado [i] .isupper ():  
204. origCase.append (decryptedText [i] .upper ())  
205. else:  
206. origCase.append (decryptedText [i] .lower ())  
207. decryptedText = ".join (origCase)  
208  
209. # Verifique com o usuário para ver se a chave foi encontrada:  
210. print ('Possível corte de criptografia com a chave% s:'% (possibleKey))  
211. print (decryptedText [: 200]) # Mostra apenas os primeiros 200 caracteres.
```

```
212. print ()  
213. print ('Digite D se estiver pronto, qualquer outra coisa para continuar a  
hackear:')  
214. response = input ('>')  
215  
216. if response.strip ().Upper ().Startswith ('D'):  
217.     return decryptedText  
218  
219. # Não foi encontrada descriptografia de aparência inglesa, portanto,  
    retornar Nenhum:  
220. retorno Nenhum  
221  
222  
223. def hackVigenere (texto cifrado):  
224. # Primeiro, precisamos fazer o exame de Kasiski para descobrir o que  
    # comprimento da chave de criptografia do texto cifrado é:  
225. allLikelyKeyLengths = kasiskiExamination (texto cifrado)  
226. se não SILENT_MODE:  
227.     keyLengthStr = "  
228.     para keyLength em allLikelyKeyLengths:  
229.         keyLengthStr += '% s' % (keyLength)  
230.     print ('resultados do exame Kasiski dizem que os comprimentos de chave  
    mais prováveis  
    são: ' + keyLengthStr + '\ n ')  
231.     hackedMessage = Nenhum  
232.     para keyLength em allLikelyKeyLengths:  
233.         se não SILENT_MODE:  
234.             print ('Tentando hackar com comprimento de chave% s (% s chaves  
            possíveis) ...'  
            % (keyLength, NUM_MOST_FREQ LETTERS ** keyLength))  
235.             hackedMessage = attemptHackWithKeyLength (texto cifrado, keyLength)  
236.         se hackedMessage! = Nenhum:  
237.             quebrar  
238.         quebrar  
239  
240. # Se nenhum dos comprimentos de chave encontrados usando o exame  
    Kasiski  
241. # trabalhado, inicie o brute-forcing através de comprimentos de chave:
```

```
242. if hackedMessage == Nenhum:  
243. se não SILENT_MODE:  
244. print ('Não é possível hackear mensagem com comprimentos de chave  
prováveis.  
forçando o tamanho da chave ... ')  
245. para keyLength no intervalo (1, MAX_KEY_LENGTH + 1):  
246. # Não volte a verificar comprimentos de chaves já experimentados no  
Kasiski:  
247. if keyLength não em allLikelyKeyLengths:  
248. se não SILENT_MODE:  
249. print ('Tentando hackear com comprimento de chave% s (% s possivel  
chaves) ... '% (keyLength, NUM_MOST_FREQ LETTERS **  
keyLength))  
250. hackedMessage = attemptHackWithKeyLength (texto cifrado,  
keyLength)  
251. if hackedMessage! = Nenhum:  
252. quebrar  
253. return hackedMessage  
254  
255  
256. # Se vigenereHacker.py for executado (em vez de importado como um  
módulo), chame  
257. # a função main ():  
258. if __name__ == '__main__':  
259. main ()
```

Exemplo de Execução do Programa Vigenere Hacking

Quando você executa o programa *vigenereHacker.py* , a saída deve ficar assim:

Os resultados do exame de Kasiski indicam que os comprimentos de chave mais prováveis são: 3 2 6 4 12

Tentando cortar com comprimento de chave 3 (27 chaves possíveis) ...

Possíveis letras para a letra 1 da chave: ALM

Possíveis letras para a letra 2 da chave: SNO

Possíveis letras para a letra 3 da chave: VIZ

Tentando com chave: ASV

Tentando com chave: ASI

- recorte -

Tentando com chave: MOI

Tentando com chave: MOZ

Tentando cortar com comprimento de chave 2 (9 chaves possíveis) ...

Possíveis letras para a letra 1 da chave: OAE

Possíveis letras para a letra 2 da chave: MSI

Tentando com chave: OM

Tentando com chave: OS

- recorte -

Tentando com chave: ES

Tentando com chave: EI

Tentando hackear com comprimento de chave 6 (729 chaves possíveis) ...

Possíveis letras para a letra 1 da chave: AEO

Possíveis letras para a letra 2 da chave: SDG

Possíveis letras para a letra 3 da chave: IVX

Possíveis letras para a letra 4 da chave: MZQ

Possíveis letras para a letra 5 da chave: OBZ

Possíveis letras para a letra 6 da chave: VIK

Tentando com chave: ASIMOV

Hack de criptografia possível com chave ASIMOV:

ALAN MATHISON TURING ERA UM MATEMÁTICO BRITÂNICO,
LÓGICO, CRYPTANALYST E

CIENTISTA DE COMPUTADOR. ELE FOI ALTAMENTE INFLUENTE NO
DESENVOLVIMENTO DO COMPUTADOR

CIÊNCIA, FORNECENDO A FORMALIZAÇÃO DO CON

Digite D para pronto, ou apenas pressione Enter para continuar o hacking:

> d

Copiando mensagem hackeada para a área de transferência:

Alan Mathison Turing foi um matemático britânico, lógico, criptoanalista e
cientista da computação. Ele foi altamente influente no desenvolvimento de
computador

- recorte -

Importando Módulos e Configurando a Função main ()

Vamos dar uma olhada no código-fonte do programa de hackers Vigenère. O
programa de hackers importa muitos módulos diferentes, incluindo um novo
módulo chamado `itertools` , sobre o qual você aprenderá mais em breve:

1. # Vigenere Cipher Hacker

2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)

3

4. importar ferramentas, re

5. import vigenereCipher, pyperclip, freqAnalise, detectEnglish

Além disso, o programa configura várias constantes nas linhas 7 a 11, que explicarei mais tarde quando forem usadas no programa.

A função main () do programa de hacking é semelhante às funções main () em funções de hackers anteriores:

14. def main ():

15. # Em vez de digitar este texto cifrado, você pode copiá-lo e colá-lo

16. # de <https://www.nostarch.com/crackingcodes/>:

17. ciphertext = """ "Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuo, lgouqdaf, kdmktsvmztsl, izr xoexghrz kkusitaaf. Vz wsa twbhdg ubalmmzhdad qz

- recorte -

azmtmd'g widt ião bwnafz tzm Tcpsw wr Zjrva ivdcz eaigd yzmbo Tmzubb a kbmhptgzk dvrvwz wa efiohzd. """ "

18. hackedMessage = hackVigenere (texto cifrado)

19

20. se hackedMessage! = Nenhum:

21. print ('Copiando mensagem hackeada para área de transferência:')

22. print (hackedMessage)

23. pyperclip.copy (hackedMessage)

24. else:

25. print ('Falha ao hackar criptografia')

O texto cifrado é passado para a função hackVigenere () , que retorna a string descriptografada se o hack for bem-sucedido ou o valor None se o hack falhar. Se for bem sucedido, o programa imprime a mensagem hackeada na tela e a copia para a área de transferência.

Encontrando Seqüências Repetidas

A função findRepeatSequencesSpacings () realiza a primeira etapa do exame Kasiski, localizando todas as seqüências repetidas de letras na sequência de mensagens e contando os espaçamentos entre as seqüências:

28. def findRepeatSequencesSpacings (message):

- recorte -

```
33. # Use uma expressão regular para remover não-letras da mensagem:  
34. message = NONLETTERS_PATTERN.sub ("", message.upper ())  
35  
36. # Compile uma lista de seqüências seqLen-letter encontradas na mensagem:  
37. seqSpacings = {} # Keys são seqüências; valores são listas de espaçamentos  
int.
```

A linha 34 converte a mensagem em maiúscula e remove todos os caracteres que não sejam letras da mensagem usando o método de expressão regular `sub ()`.

O dicionário `seqSpacings` na linha 37 contém seqüências de seqüências repetidas como suas chaves e uma lista com inteiros representando o número de letras entre todas as ocorrências daquela sequência como seus valores. Por exemplo, se passarmos a string de exemplo 'PPQCAXQV ...' como mensagem , a função `findRepeatSequenceSpacings ()` retornará {'VRA': [8, 24, 32], 'AZU': [48], 'YBN': [8]} .

O loop for na linha 38 verifica se cada sequência se repete encontrando as seqüências na mensagem e calculando os espaçamentos entre seqüências repetidas:

```
38. para seqLen no intervalo (3, 6):  
39. para seqInicie no intervalo (len (mensagem) - seqLen):  
40. # Determine qual é a sequência e armazene-a em seq:  
41. seq = message [seqStart: seqStart + seqLen]
```

Indexes	0123456
message	P P Q C A X Q
seqStart = 0	P P Q
seqStart = 1	P Q C
seqStart = 2	Q C A
seqStart = 3	C A X
seqStart = 4	A X Q

Figura 20-2: Valores de seq da mensagem dependendo do valor em seqStart

Na primeira iteração do loop, o código encontra seqüências com exatamente três letras. Na próxima iteração, ele encontra seqüências com exatamente quatro letras e cinco letras. Você pode alterar os comprimentos de seqüência que o código procura modificando o intervalo (3, 6) na linha 38; no entanto, encontrar seqüências repetidas de comprimento três, quatro e cinco parece funcionar para a maioria dos textos cifrados. A razão é que elas são longas o suficiente para que as repetições no texto cifrado provavelmente não sejam coincidências, mas

curtas o suficiente para que sejam encontradas repetições. O comprimento da sequência da por laço é actualmente corrente é armazenado em seqlen .

O loop for na linha 39 divide a mensagem em todas as subseqüências possíveis de length seqLen . Usaremos este para loop para determinar o início da fatia e fatia mensagem em uma substring seqlen caracteres. Por exemplo, se o seqLen é 3 e a mensagem é 'PPQCAXQ' , gostaríamos de começar pelo primeiro índice, que é 0 , e dividir três caracteres para obter a subcadeia 'PPQ' . Então, gostaríamos de ir para o próximo índice, que é 1 , e dividir três caracteres para obter a subseqüência 'PQC'. Precisamos fazer isso para cada índice até os três últimos caracteres, que é o índice equivalente a len (message) - seqLen . Fazendo isso, você obteria as seqüências mostradas na [Figura 20-2](#) .

O loop for na linha 39 percorre cada índice até len (message) - seqLen e atribui o índice atual para iniciar a fatia de substring para a variável seqStart . Depois de termos o índice inicial, a linha 41 define a variável seq para a fatia de substring.

Procuraremos na mensagem por repetições dessa fatia usando o loop for na linha 44.

43. # Procure esta seqüência no resto da mensagem:

44. para i na faixa (seqStart + seqLen, len (mensagem) - seqLen):

45. if mensagem [i: i + seqLen] == seq:

O loop for na linha 44 está dentro do loop for na linha 39 e define i como sendo os índices de cada sequência possível de comprimento seqLen na mensagem . Estes índices iniciam em seqStart + seqLen , ou após a seqüência atualmente em seq , e vão até len (message) - seqLen , que é o último índice onde uma seqüência de length seqLen pode ser encontrada. Por exemplo, se a mensagem era 'PPQCAXQVEKG YBN KMAZU YBN ' , seqStart era 11 e seqLen era3 , a linha 41 definiria seq para 'YBN' . O loop for começaria a olhar para a mensagem a partir do índice 14 .

A mensagem de expressão [i: i + seqLen] na linha 45 avalia uma subseqüência de mensagem , que é comparada a seq para verificar se a subseqüência de caracteres é uma repetição de seq . Se estiver, as linhas 46 a 52 calculam o espaçamento e o adicionam ao dicionário seqSpacings . Na primeira iteração, a linha 45 compara 'KMA' a seq , depois 'MAZ' a seq na próxima iteração, depois 'AZU' a seq na próxima e assim por diante. Quando eu tenho 19 anos , a linha 45 descobre que 'YBN' é igual a seq e a execução executa as linhas 46 a 52:

```
46. # Encontrou uma sequência repetida:  
47. if seq não seqSpacings:  
48. seqSpacings [seq] = [] # Inicializa lista em branco.  
49.  
50. # Anexe a distância de espaçamento entre a  
sequência repetida 51. # e a sequência original:  
52. seqSpacings [seq] .append (i - seqStart)
```

As linhas 47 e 48 verificam se a variável seq existe como uma chave em seqSpacings . Se não, seqSpacings [seq] é definido como uma chave com uma lista em branco como seu valor.

O número de letras entre a seqüência na mensagem [i: i + seqLen] e a seqüência original na mensagem [seqStart: seqStart + seqLen] é simplesmente i - seqStart . Observe que i e seqStart são os índices iniciais antes dos dois pontos. Então, o inteiro que i - seqStart avalia é o número de letras entre as duas seqüências, que acrescentamos à lista armazenada em seqSpacings [seq] .

Quando todos esses loops for finalizados, o dicionário seqSpacings deve conter todas as seqüências repetidas de comprimento 3, 4 e 5, bem como o número de letras entre seqüências repetidas. O dicionário seqSpacings é retornado de findRepeatSequencesSpacings () na linha 53:

```
53. return seqEspaços
```

Agora que você viu como o programa executa a primeira etapa do exame Kasiski encontrando sequências repetidas no texto cifrado e contando o número de letras entre elas, vamos ver como o programa conduz a próxima etapa do exame Kasiski.

Calculando os Fatores dos Espaçamentos

Lembre-se de que o próximo passo do exame de Kasiski envolve encontrar os fatores dos espaçamentos. Estamos procurando por fatores entre 2 e MAX_KEY_LENGTH de comprimento. Para fazer isso, criaremos a função getUsefulFactors () , que recebe um parâmetro num e retorna uma lista apenas dos fatores que atendem a esse critério.

```
56. def getUsefulFactors (num):  
- recorte -  
61. if num <2:  
62. return [] # Números menores que 2 não possuem fatores úteis.
```

63

64. factors = [] # A lista de fatores encontrados.

A linha 61 verifica o caso especial em que num é menor que 2 . Neste caso, a linha 62 retorna a lista vazia porque num não teria nenhum fator útil se fosse menor que 2 .

Se num for maior que 2 , precisaríamos calcular todos os fatores de num e armazená-los em uma lista. Na linha 64, criamos uma lista vazia chamada fatores para armazenar os fatores.

O loop for na linha 68 percorre os inteiros de 2 até MAX_KEY_LENGTH , incluindo o valor em MAX_KEY_LENGTH . Lembre-se de que, como range () faz com que o loop for iterate, mas não inclui o segundo argumento, passamos MAX_KEY_LENGTH + 1 para que MAX_KEY_LENGTH seja incluído. Este laço encontra todos os fatores de num .

68. para i no intervalo (2, MAX_KEY_LENGTH + 1): # Não teste 1: não é útil.

69. if num% i == 0:

70. fatores.append (i)

71. otherFactor = int (num / i)

A linha 69 testa se num% i é igual a 0 ; se for, sabemos que eu divido um número uniformemente sem nenhum resto, o que significa que eu sou um fator de num . Nesse caso, a linha 70 acrescenta i à lista de fatores na variável de fatores . Como sabemos que num / i também deve dividir numérico uniformemente, a linha 71 armazena a forma inteira dele em otherFactor . (Lembre-se de que o operador / sempre avalia um valor flutuante, como 21/7 avaliando o float 3.0 em vez do int 3.) Se o valor resultante for 1 , o programa não o inclui na lista de fatores , portanto, a linha 72 verifica este caso:

72. if otherFactor <MAX_KEY_LENGTH + 1 e otherFactor! = 1:

73. factors.append (otherFactor)

A linha 73 acrescenta o valor, se não for 1 . Nós excluímos 1 porque se a chave de Vigenère tivesse um comprimento de 1, a cifra de Vigenère não seria diferente da cifra de César!

Removendo Duplicados com a Função set ()

Lembre-se de que precisamos conhecer o fator mais comum como parte do exame de Kasiski, porque o fator mais comum será quase certamente o comprimento da chave de Vigenère. No entanto, antes de analisarmos a

frequência de cada fator, precisaremos usar a função set () para remover quaisquer fatores duplicados da lista de fatores . Por exemplo, se getUsefulFactors () foi passada 9 para o num parâmetro, então 9% 3 == 0 seria verdadeiro e ambos i e otherFactor teria sido acrescentada ao factores . Mas tanto eu como int (num / i) são iguais a3 , então 3 seria anexado à lista duas vezes. Para evitar números duplicados, podemos passar a lista para set () , que retorna uma lista como um tipo de dados definido. O tipo de dados *definido* é semelhante ao tipo de dados da lista, exceto que um valor definido pode conter apenas valores exclusivos.

Você pode passar qualquer valor de lista para a função set () para obter um valor definido que não tenha nenhum valor duplicado. Por outro lado, se você passar um valor definido para list () , ele retornará uma versão de valor de lista do conjunto. Para ver exemplos disso, insira o seguinte no shell interativo:

```
>>> set ([1, 2, 3, 3, 4])
set ([1, 2, 3, 4])
>>> spam = list (conjunto ([2, 2, 2, 'gatos', 2 , 2]))
>>> spam
[2, 'gatos']
```

Quaisquer valores de lista repetidos são removidos quando uma lista é convertida em um conjunto. Mesmo quando um conjunto convertido de uma lista é reconvertido em uma lista, ele ainda não terá nenhum valor repetido.

Removendo Fatores Duplicados e Classificando a Lista

A linha 74 passa o valor da lista em fatores para set () para remover quaisquer fatores duplicados:

74. lista de retorno (set (fatores)) # Remove fatores duplicados.

A função getItemAtIndexOne () na linha 77 é quase idêntica a getItemAtIndexZero () no programa freqAnalysis.py que você escreveu no [Capítulo 19](#) (veja “ [Obtendo o primeiro membro de uma tupla](#) ” na [página 268](#)):

77. def getItemAtIndexOne (x):

78. return x [1]

Essa função será passada para sort () mais tarde no programa para classificar com base no item no índice 1 dos itens que estão sendo classificados.

Encontrando os fatores mais comuns

Para encontrar os fatores mais comuns, que são os comprimentos de chave mais prováveis, precisamos escrever a função `getMostCommonFactors()`, que começa na linha 81.

```
81. def getMostCommonFactors (seqFactors):  
82. # Primeiro, conte quantas vezes um fator ocorre em seqFactors:  
83. factorCounts = {} # Key é um fator; valor é com que frequência ocorre.
```

O parâmetro `seqFactors` na linha 81 recebe um valor de dicionário criado usando a função `kasiskiExamination()`, que explicarei em breve. Este dicionário tem seqüências de seqüências como suas chaves e uma lista de fatores inteiros como o valor de cada chave. (Esses são fatores dos inteiros de espaçamento que `findRepeatSequencesSpacings()` retornaram anteriormente.) Por exemplo, `seqFactors` poderia conter um valor de dicionário que se parece com algo como:

```
{'VRA': [8, 2, 4, 2, 3, 4, 6, 8, 12, 16, 8, 2, 4], 'AZU': [2, 3, 4, 6, 8, 12, 16, 24], 'YBN': [8, 2, 4]}
```

A função `getMostCommonFactors()` ordena os fatores mais comuns em `seqFactors`, do mais freqüente ao menos ocorrente, e os retorna como uma lista de tuplas de dois inteiros. O primeiro inteiro na tupla é o fator, e o segundo inteiro é quantas vezes ele apareceu em `seqFactors`.

Por exemplo, `getMostCommonFactors()` pode retornar um valor de lista, como este:

```
[(3, 556), (2, 541), (6, 529), (4, 331), (12, 325), (8, 171), (9, 156), (16, 105), (5, 98), (11, 86), (10, 84), (15, 84), (7, 83), (14, 68), (13, 52)]
```

Esta lista mostra que no dicionário `seqFactors` que foi passado para `getMostCommonFactors()`, o fator 3 ocorreu 556 vezes, o fator 2 ocorreu 541 vezes, o fator 6 ocorreu 529 vezes e assim por diante. Note que 3 aparece em primeiro lugar na lista porque é o fator mais frequente; 13 é o fator menos frequente e, portanto, é o último da lista.

Para a primeira etapa de `getMostCommonFactors()`, configuraremos o dicionário `factorCounts` na linha 83, que usaremos para armazenar as contagens de cada fator. A chave de `factorCounts` será o fator e os valores associados às chaves serão as contagens desses fatores.

Em seguida, o loop `for` na linha 88 faz um loop sobre cada sequência em `seqFactors`, armazenando-a em uma variável denominada `seq` em cada iteração. A lista de fatores em `seqFactors` para `seq` é armazenada em uma variável

denominada factorList na linha 89:

```
88. para seq em seqFactores:  
89.   factorList = seqFactores [seq]  
90. para fator em factorList:  
91.   if factor not in factorContains:  
92.     factorCounts [factor] = 0  
93.     factorCounts [factor] += 1
```

Os fatores nessa lista são colocados em loop com um loop for na linha 90. Se um fator não existir como uma chave em factorCounts , ele será incluído na linha 92 com um valor de 0 . A linha 93 incrementa factorCounts [factor] , que é o valor do fator em factorCounts .

Para a segunda etapa de getMostCommonFactors () , precisamos classificar os valores no dicionário factorCounts por sua contagem. Mas como os valores do dicionário não são ordenados, devemos primeiro converter o dicionário em uma lista de tuplas de dois inteiros. (Fizemos algo semelhante no [Capítulo 19](#) na função getFrequencyOrder () no módulo *freqAnalysis.py*). Armazenamos esse valor de lista em uma variável denominada factorsByCount , que inicia como uma lista vazia na linha 97:

```
97. factorsByCount = []  
98. para fator em factorCounts:  
99.   # Exclui fatores maiores que MAX_KEY_LENGTH:  
100.  if fator <= MAX_KEY_LENGTH:  
101.    # factorsByCount é uma lista de tuplas: (factor, factorCount).  
102.    # factorsByCount tem um valor como [(3, 497), (2, 487), ...].  
103.    factorsByCount.append ((factor, factorCounts [fator]))
```

Em seguida, o loop for na linha 98 passa por cada um dos fatores em factorCounts e anexa essa tupla (factor, factorCounts [factor]) à lista factorsByCount apenas se o fator for menor ou igual a MAX_KEY_LENGTH .

Depois que o loop for conclui a adição de todas as tuplas a factorsByCount , a linha 106 classifica os fatoresByCount como a etapa final da função getMostCommonFactors () .

```
106. factorsByCount.sort (key = getItemAtIndexOne, reverso = Verdadeiro)  
107.  
108. return factorsByCount
```

Porque o getItemAtIndexOne função é passada para a chave argumento palavra-chave e Verdadeiro é passado para o reverso argumento palavra-chave, a lista é ordenada por as contagens fator em ordem decrescente. A linha 108 retorna a lista classificada em factorsByCount , que deve indicar quais fatores aparecem com mais freqüência e, portanto, é mais provável que sejam os comprimentos de chave do Vigenère.

Encontrando os Comprimentos Mais Prováveis da Chave

Antes de podermos descobrir quais são as possíveis subchaves para um texto cifrado, precisamos saber quantas subchaves existem. Ou seja, precisamos saber o tamanho da chave. A função kasiskiExamination () na linha 111 retorna uma lista dos comprimentos de chave mais prováveis para o dado argumento de texto cifrado .

111. def kasiskiExamination (texto cifrado):

- recorte -

115. repeatSeqSpacings = findRepeatSequencesSpacings (texto cifrado)

Os comprimentos de chave são inteiros em uma lista; o primeiro inteiro na lista é o comprimento de chave mais provável, o segundo inteiro o segundo mais provável e assim por diante.

O primeiro passo para encontrar o tamanho da chave é encontrar os espaçamentos entre seqüências repetidas no texto cifrado. Isso é retornado da função findRepeatSequencesSpacings () como um dicionário com as seqüências de caracteres de seqüências suas chaves e uma lista com os espaçamentos como inteiros como seus valores. A função findRepeatSequencesSpacings () foi descrita anteriormente em “ [Encontrando seqüências repetidas](#) ” na [página 294](#) .

Antes de continuar com as próximas linhas de código, você precisará aprender sobre o método da lista extend () .

O método de lista extend ()

Quando você precisa adicionar vários valores ao final de uma lista, há uma maneira mais fácil do que chamar append () dentro de um loop. O método da lista extend () pode adicionar valores ao final de uma lista, semelhante ao método da lista append () . Ao passar por uma lista, o método append () adiciona a lista inteira como um item ao final de outra lista, assim:

```
>>> spam = ['cat', 'dog', 'mouse']
>>> ovos = [1, 2, 3]
```

```
>>> spam.append (ovos)
>>> spam
['gato', 'cachorro', 'rato', [1, 2, 3]]
```

Em contraste, o método extend () adiciona cada item no argumento list ao final de uma lista. Digite o seguinte no shell interativo para ver um exemplo:

```
>>> spam = ['cat', 'dog', 'mouse']
>>> ovos = [1, 2, 3]
>>> spam.extend (ovos)
>>> spam
['gato', 'cachorro', 'rato', 1, 2, 3]
```

Como você pode ver, todos os valores em ovos (1 , 2 e 3) são anexados ao spam como itens distintos.

Estendendo o dicionário repeatSeqSpacings

Embora repeatSeqSpacings seja um dicionário que mapeia strings de sequências para listas de espaçamentos inteiros, na verdade precisamos de um dicionário que mapeie sequências de sequências para listas de fatores desses espaçamentos inteiros. (Veja “ [Obtendo Fatores de Espaçamentos](#) ” na [página 283](#) pelo motivo.) As linhas 118 a 122 fazem isso:

```
118. seqFactors = {}
119. para seq em
repeatSeqSpacings : 120. seqFactors [seq] = []
121. para espaçoamento em
repeatSeqSpacings [seq] : 122. seqFactors [seq] .extend (getUsefulFactors
(espaçamento))
```

A linha 118 começa com um dicionário vazio em seqFactors . O loop for na linha 119 itera sobre cada chave, que é uma sequência de seqüência, no dicionário repeatSeqSpacings . Para cada chave, a linha 120 define uma lista em branco para ser o valor em seqFactors .

O loop for na linha 121 itera sobre todos os inteiros de espaçamentos passando cada um para uma chamada getUsefulFactors () . Cada um dos itens da lista retornados de getUsefulFactors () é adicionado ao seqFactors [seq] usando o método extend () . Quando todo o para loops são acabado, seqFactors deve ser um dicionário que mapeia cadeias de sequência para listas de factores de espaçamentos inteiros. Isso nos permite ter os fatores dos espaçamentos, não

apenas os espaçamentos.

A linha 125 passa o dicionário seqFactors para a função getMostCommonFactors () e retorna uma lista de tuplas de dois inteiros cujo primeiro inteiro representa o fator e cujo segundo inteiro mostra com que frequência esse fator aparece em seqFactors . Em seguida, a tupla é armazenada em factorsByCount .

125. factorsByCount = getMostCommonFactors (seqFactors)

Mas queremos que a função kasiskiExamination () retorne uma lista de fatores inteiros, não uma lista de tuplas com fatores e a contagem de quantas vezes eles apareceram. Como esses fatores são armazenados como o primeiro item da lista de tuplas de dois inteiros em factorsByCount , precisamos extrair esses fatores das tuplas e colocá-los em uma lista separada.

Obtendo os fatores de factorsByCount

As linhas 130 a 134 armazemam a lista separada de fatores em allLikelyKeyLengths .

130. allLikelyKeyLengths = []

131. para twoIntTuple em factorsByCount:

132. allLikelyKeyLengths.append (twoIntTuple [0])

133

134. return allLikelyKeyLengths

O loop for na linha 131 itera sobre cada uma das tuplas em factorsByCount e acrescenta o item 0 do índice da tupla ao final de todos osLikelyKeyLengths . Depois que esse loop for concluído, a variável allLikelyKeyLengths deve conter todos os fatores inteiros em factorsByCount , que são retornados como uma lista de kasiskiExamination () .

Embora agora tenhamos a capacidade de localizar os comprimentos de chave prováveis com os quais a mensagem foi criptografada, precisamos separar as letras da mensagem que foram criptografadas com a mesma subchave. Lembre-se que criptografar 'THEDOGANDTHECAT' com a tecla 'XYZ' acaba usando o 'X' da chave para criptografar as letras da mensagem no índice 0 , 3 , 6 , 9 e 12 . Como essas letras da mensagem original em inglês são criptografadas com a mesma subchave ('X'), o texto descriptografado deve ter uma contagem de frequência de letras semelhante ao inglês. Podemos usar essas informações para descobrir a subchave.

Obtenção de cartas criptografadas com a mesma subchave

Para extrair as letras de um texto cifrado que foram criptografadas com a mesma subchave, precisamos escrever uma função que possa criar uma string usando a 1^a, a 2^a ou a n -ésima letras de uma mensagem. Depois que a função tiver o índice inicial, o comprimento da chave e a mensagem passada a ele, a primeira etapa é remover os caracteres que não são letras da mensagem usando um objeto de expressão regular e seu método sub () na linha 145.

NOTA

Expressões regulares são discutidas em “ [Encontrando caracteres com expressões regulares](#) ” na [página 230](#) .

Esta string somente de letras é então armazenada como o novo valor na mensagem :

137. def getNthSubkeysLetters (nth, keyLength, message):

- recorte -

145. message = NONLETTERS_PATTERN.sub ("", message.upper ())

Em seguida, criamos uma string anexando as sequências de letras a uma lista e, em seguida, usando join () para mesclar a lista em uma única string:

147. i = nth - 1

148. letras = []

149. enquanto i < len (mensagem):

150. letras.append (mensagem [i])

151. i + = keyLength

152. return " .join (letras)

A variável i aponta para o índice da letra na mensagem que você deseja anexar à lista de construção de cadeia, que é armazenada em uma variável chamada letras . A variável i começa com o valor nth - 1 na linha 147, e a variável letters começa com uma lista em branco na linha 148.

O enquanto circuito de linha 149 continua a ser executado, enquanto que é menor do que o comprimento da mensagem . Em cada iteração, a letra na mensagem [i] é anexada à lista em letras . Então eu é atualizado para apontar para a próxima letra na subchave, adicionando keyLength para i na linha 151.

Após este ciclo termina, linha 152 junta-se os valores de cadeia única letra nas cartas lista em uma corda, e esta string é retornado de getNthSubkeysLetters () .

Agora que podemos extrair letras que foram criptografadas com a mesma subchave, podemos usar `getNthSubkeysLetters()` para tentar descriptografar com alguns comprimentos de chave em potencial.

Tentativa de descriptografia com um comprimento de chave provável

Lembre-se de que não é garantido que a função `kasiskiExamination()` retorne o comprimento real da chave Vigenère, mas sim uma lista de vários comprimentos possíveis, ordenados na ordem mais provável para o comprimento de chave menos provável. Se o código determinou o tamanho errado da chave, ele tentará novamente usando um comprimento de chave diferente. A função `attemptHackWithKeyLength()` faz isso quando passa o texto cifrado e o comprimento da chave determinada. Se bem sucedida, esta função retorna uma string da mensagem hackeada. Se o hacking falhar, a função retornará `None`.

O código de hacking funciona somente em letras maiúsculas, mas queremos retornar qualquer string descriptografada com sua caixa original, então precisamos preservar a string original. Para fazer isso, armazenamos a forma maiúscula da string de texto cifrado em uma variável separada chamada `ciphertextUp` na linha 157.

```
155. def attemptHackWithKeyLength(texto_cifrado, mostLikelyKeyLength):
156.     # Determina as letras mais prováveis para cada letra da chave:
157.     ciphertextUp = ciphertext.upper()
```

Se assumirmos que o valor em `mostLikelyKeyLength` é o comprimento de chave correto, o algoritmo de hackeamento chama `getNthSubkeysLetters()` para cada subchave e então força bruta através das 26 letras possíveis para encontrar aquela que produz texto descriptografado cuja freqüência de letras corresponde mais de perto à letra frequência de inglês para essa subchave.

Primeiro, uma lista vazia é armazenada em `allFreqScores` na linha 160, que armazenará as pontuações de correspondência de freqüência retornadas por `freqAnalysis.englishFreqMatchScore()`:

```
160. allFreqScores = []
161. para enésimo intervalo (1, maisLikelyKeyLength + 1):
162.     nthLetters = getNthSubkeysLetters(nth, mostLikelyKeyLength,
163.                                         ciphertextUp)
```

O loop for na linha 161 define a enésima variável para cada inteiro de 1 para o valor `mostLikelyKeyLength`. Lembre-se que quando `range()` é passado dois

argumentos, o intervalo vai até, mas não incluindo, o segundo argumento. O + 1 é colocado no código para que o valor inteiro em mostLikelyKeyLength seja incluído no objeto de intervalo retornado.

As letras da n -ésima subchave são retornadas de getNthSubkeysLetters () na linha 162.

Em seguida, precisamos descriptografar as letras da n -ésima subchave com todas as 26 subchaves possíveis para ver quais produzem freqüências de letras semelhantes a inglês. Uma lista de resultados de correspondência de frequência em inglês é armazenada em uma lista em uma variável denominada freqScores . Esta variável começa como uma lista vazia na linha 168 e então o loop for na linha 169 faz um loop através de cada uma das 26 letras maiúsculas da string LETTERS :

168. freqScores = []

169. para possívelKey em LETTERS:

170. decryptedText = vigenereCipher.decryptMessage (possibleKey, nthLetters)

O possibleKey valor decifra o texto cifrado chamando vigenereCipher.decryptMessage () na linha 170. A subchave no possibleKey é apenas uma letra, mas a corda no nthLetters é composta de apenas as cartas de mensagem que teria sido criptografados com essa subchave se o código determinou o tamanho da chave corretamente.

O texto descriptografado é então passado para freqAnalysis.englishFreqMatchScore () para ver até que ponto a freqüência das letras em decryptedText corresponde à freqüência de letras do inglês regular. Como você aprendeu no [Capítulo 19](#) , o valor de retorno é um número inteiro entre 0 e 12 : lembre-se de que um número maior significa uma correspondência mais próxima.

A linha 171 coloca essa pontuação de correspondência de frequência e a chave usada para descriptografar em uma tupla e a armazena na variável keyAndFreqMatchTuple . Esta tupla é anexada ao final de freqScores na linha 172:

171. keyAndFreqMatchTuple = (possívelKey, freqAnalysis.englishFreqMatchScore (decryptedText))

172. freqScores.append (keyAndFreqMatchTuple)

Após a conclusão do loop for na linha 169, a lista freqScores deve conter 26 tuplas de pontuação de correspondência-e-frequência: uma tupla para cada uma das 26 subchaves. Precisamos classificar essa lista para que as tuplas com as maiores pontuações de correspondência de frequência em inglês estejam em primeiro lugar. Isso significa que queremos ordenar as tuplas no freqScores pelo valor no índice 1 e na ordem inversa (decrescente).

Chamamos o método sort () na lista freqScores , passando o valor da função getItemAtIndexOne para o argumento key keyword. Note que estamos *não* chamar a função, como você pode dizer da falta de parênteses. O valor True é passado para o argumento da palavra-chave reversa para classificar em ordem decrescente.

174. freqScores.sort (key = getItemAtIndexOne, reverse = True)

Inicialmente, a constante NUM_MOST_FREQ LETTERS foi configurada para o valor inteiro 4 na linha 9. Após classificar as tuplas em freqScores na ordem inversa, a linha 176 acrescenta uma lista contendo apenas as três primeiras tuplas, ou as tuplas com as três maiores pontuações de correspondência de frequência em inglês. para allFreqScores . Como resultado, allFreqScores [0] contém as pontuações de frequência para a primeira subchave, allFreqScores [1] contém as pontuações de frequência para a segunda subchave e assim por diante.

176. allFreqScores.append (freqScores [: NUM_MOST_FREQ LETTERS])

Depois que o loop for na linha 161 for concluído, allFreqScores deverá conter um número de valores de lista igual ao valor inteiro em mostLikelyKeyLength . Por exemplo, se mostLikelyKeyLength fosse 3 , allFreqScores seria uma lista de três listas. O primeiro valor da lista contém as tuplas das três principais subchaves correspondentes para a primeira subchave da chave Vigenère completa. O segundo valor da lista contém as tuplas das três principais subchaves correspondentes para a segunda subchave da chave Vigenère completa, e assim por diante.

Originalmente, se quiséssemos forçar a força bruta através da chave Vigenère completa, o número de chaves possíveis seria aumentado para o tamanho da chave. Por exemplo, se a chave fosse ROSEBUD com um comprimento de 7, haveria 26⁷ , ou 8,031,810,176, chaves possíveis.

Mas verificar a correspondência de frequência em inglês ajudou a determinar as quatro letras mais prováveis para cada subchave. Continuando com o exemplo

ROSEBUD, isso significa que precisamos apenas verificar 4 7 ou 16.384 chaves possíveis, o que é uma grande melhoria em mais de 8 bilhões de chaves possíveis!

O argumento final da palavra-chave para impressão ()

Em seguida, queremos imprimir a saída para o usuário. Para fazer isso, usamos `print ()`, mas passamos um argumento para um parâmetro opcional que não usamos antes. Sempre que a função `print ()` é chamada, ela imprime na tela a string passada para ela junto com um caractere de nova linha. Para imprimir outra coisa no final da string em vez de um caractere de nova linha, podemos especificar a string para o `print ()` função do fim argumento palavra-chave. Digite o seguinte no shell interativo para ver como usar o `print ()` da função de final argumento palavra-chave:

```
>>> def printStuff ():  
    ❶ print ('Olá', end = '\ n')  
    ❷ print ('Howdy', end = "")  
    ❸ print ('Saudações', end = 'XYZ')  
    print (' Adeus ')  
>>> printStuff ()  
Olá  
HowdyGreetingsXYZGoodbye
```

Passing `end = '\ n'` imprime a string normalmente ❶ . No entanto, passar `end = ""` ❷ ou `end = 'XYZ'` ❸ substitui o caractere de nova linha usual, portanto as chamadas de `print ()` subsequentes não são exibidas em uma nova linha.

Executando o programa no modo silencioso ou imprimindo informações para o usuário

Neste ponto, queremos saber quais letras são os três principais candidatos para cada subchave. Se a constante `SILENT_MODE` fosse definida como `False` anteriormente no programa, o código nas linhas 178 a 184 imprimaria os valores em `allFreqScores` na tela:

```
178. se não SILENT_MODE:  
179. para i em range (len (allFreqScores)):  
180. # Use i + 1 para que a primeira letra não seja chamada de letra "0th":  
181. print ('Letras possíveis para letra% s da chave: "% (i + 1),  
end =' ')  
182. para o freqScore em allFreqScores [i]:
```

```
183. print ("% s % freqScore [0], end =' ')
184. impressão ( ) # Imprima uma nova linha.
```

Se SILENT_MODE for definido como True , o código no bloco da instrução if será ignorado.

Agora reduzimos o número de subchaves para um número pequeno o suficiente para podermos forçar todos eles. Em seguida, você aprenderá a usar a função `itertools.product()` para gerar todas as combinações possíveis de subchaves para força bruta.

Encontrando Possíveis Combinações de Subchaves

Agora que temos subchaves possíveis, precisamos colocá-los juntos para encontrar a chave inteira. O problema é que, apesar de termos encontrado cartas para cada subchave, a letra mais provável pode não ser a letra correta. Em vez disso, a segunda letra mais provável ou terceira mais provável pode ser a letra da subchave correta. Isso significa que não podemos simplesmente combinar as letras mais prováveis de cada subchave em uma única chave: precisamos tentar combinações diferentes de letras prováveis para encontrar a chave certa.

O programa `vigenereHacker.py` usa a função `itertools.product()` para testar todas as combinações possíveis de subchaves.

A função `itertools.product()`

A função `itertools.product()` produz todas as combinações possíveis de itens em uma lista ou em um valor semelhante a uma lista, como uma string ou uma tupla. Essa combinação de itens é chamada de *produto cartesiano* , que é onde a função recebe seu nome. A função retorna um valor de objeto de produto `itertools`, que também pode ser convertido em uma lista passando-o para `list()` . Digite o seguinte no shell interativo para ver um exemplo:

```
>>> itertools importação
>>> itertools.product ( 'ABC', repetição = 4)
❶ <itertools.product objeto em 0x02C40170>
>>> lista (itertools.product ( 'ABC', repita = 4))
[('A', 'A', 'A', 'A', 'A', 'A', 'A', 'B', 'A', 'A', 'A', 'C'), ('A', 'A', 'A',
'B', 'A'), ('A', 'A', 'B', 'B'), ('A', 'A', 'B', 'C'), ('A', 'A', 'C', 'A'),
('A', 'A', 'C', 'B'), ('A', 'A', 'C', 'C'), ('A', 'B', 'A', 'A'), ('A', 'B',
'A', 'B'), ('A', 'B', 'A', 'C'), ('A', 'B', 'B', 'A'), ('A', 'B', 'B', 'B'),
- recorte -
```

```
(C, B, C, B), (C, B, C, C), (C, C, A), ('C','C',
'A','B'), ('C','C','A','C'), ('C','C','B','A'), ('C','C','B','B'),
('C','C','B','C'), ('C','C','C','A'), ('C','C','C','B'), ('C','C',
'C','C')]
```

Passando 'ABC' e o inteiro 4 para o argumento de palavra-chave repeat para `itertools.product()` retorna um objeto de produto `itertools` ❶ que, quando convertido em uma lista, possui tuplas de quatro valores com todas as combinações possíveis de 'A', 'B' e 'C'. Isso resulta em uma lista com um total de 3⁴ ou 81 tuplas.

Você também pode passar valores de lista para `itertools.product()` e alguns valores semelhantes a listas, como objetos de intervalo retornados de `range()`. Digite o seguinte no shell interativo para ver o que acontece quando listas e objetos como listas são passados para esta função:

```
>>> import itertools
>>> list(itertools.product(range(8), repeat = 5))
[(0, 0, 0, 0, 0), (0, 0, 0, 0, 1), (0, 0, 0, 0, 2), (0, 0, 0, 0, 3), (0, 0, 0,
0, 4), (0, 0, 0, 0, 5), (0, 0, 0, 0, 6), (0, 0, 0, 0, 7), (0, 0, 0, 1, 0), (0,
0, 0, 1, 1), (0, 0, 0, 1, 2), (0, 0, 0, 1, 3), (0, 0, 0, 1, 4),
- recorte -
(7, 7, 7, 6, 6), (7, 7, 7, 6, 7), (7, 7, 7, 7, 0), (7, 7, 7, 7, 1), (7, 7, 7,
7, 2), (7, 7, 7, 7, 3), (7, 7, 7, 7, 4), (7, 7, 7, 7, 5), (7, 7, 7, 7, 6), (7,
7, 7, 7, 7)]
```

Quando o objeto de intervalo retornado do `range(8)` é passado para `itertools.product()`, junto com 5 para o argumento de palavra-chave `repeat`, ele gera uma lista que possui tuplas de cinco valores, inteiros que variam de 0 a 7.

Não podemos simplesmente passar para `itertools.product()` uma lista das possíveis letras de subchave, porque a função cria combinações dos mesmos valores e cada uma das subchaves provavelmente terá letras potenciais diferentes. Em vez disso, como nossas subchaves são armazenadas em tuplas em `allFreqScores`, acessamos essas letras por valores de índice, que vão de 0 ao número de letras que queremos tentar menos 1. Sabemos que o número de letras em cada tupla é igual a `NUM_MOST_FREQ LETTERS` porque só armazenado que número de cartas possíveis em cada tupla mais cedo na linha 176. Assim, a gama de índices vamos precisar acessar é de 0 a

`NUM_MOST_FREQ LETTERS` , que é o que nós vamos passar para `itertools.product()` . Também vamos passar `itertools.product()` um comprimento de chave provável como um segundo argumento, criando tuplas enquanto o tamanho da chave em potencial.

Por exemplo, se quiséssemos tentar apenas as três primeiras letras mais prováveis de cada subchave (que é determinado por `NUM_MOST_FREQ LETTERS`) para uma chave com cinco letras, o primeiro valor que `itertools.product()` produziria seria `(0, 0, 0, 0, 0)` . O próximo valor seria `(0, 0, 0, 0, 1)` , então `(0, 0, 0, 0, 2)` , e os valores seriam gerados até atingir `(2, 2, 2, 2, 2)` . Cada inteiro nas tuplas de cinco valores representa um índice para `allFreqScores` .

Acessando as subchaves em todos os `FreqScores`

O valor em `allFreqScores` é uma lista que contém as letras mais prováveis de cada subchave junto com suas pontuações de correspondência de frequência. Para ver como esta lista funciona, vamos criar um valor `allFreqScores` hipotético em IDLE. Por exemplo, `allFreqScores` pode se parecer com isso com uma chave de seis letras onde encontramos as quatro letras mais prováveis para cada subchave:

```
>>> allFreqScores = [[('A', 9), ('E', 5), ('O', 4), ('P', 4)], [('S', 10),  
('D', 4), ('G', 4), ('H', 4)], [('T', 11), ('V', 4), ('X', 4), ('B', 3)],  
[('M', 10), ('Z', 5), ('Q', 4), ('A', 3)], [('O', 11), ('B', 4), ('Z', 4),  
('A', 3)], [('V', 10), ('T', 5), ('K', 5), ('Z', 4)]]
```

Isso pode parecer complexo, mas podemos detalhar valores específicos das listas e tuplas com indexação. Quando `allFreqScores` é acessado em um índice, ele é avaliado em uma lista de tuplas de possíveis letras para uma única subchave e suas pontuações de correspondência de frequência. Por exemplo, `allFreqScores[0]` tem uma lista de tuplas para a primeira subchave juntamente com as pontuações de correspondência de frequência de cada subchave em potencial, `allFreqScores[1]` tem uma lista de tuplas para a segunda subchave e pontuações de correspondência de frequência e assim por diante:

```
>>> allFreqScores[0]  
[('A', 9), ('E', 5), ('O', 4), ('P', 4)]  
>>> allFreqScores[1]  
[('S', 10), ('D', 4), ('G', 4), ('H', 4)]
```

Você também pode acessar cada tupla das letras possíveis para cada subchave

adicionando uma referência de índice adicional. Por exemplo, obteríamos uma tupla da letra mais provável como a segunda subchave e sua pontuação de correspondência de frequência se acessássemos `allFreqScores [1] [0]`, a segunda letra mais provável de `allFreqScores [1] [1]`, e assim em:

```
>>> allFreqScores [1] [0]
('S', 10)
>>> allFreqScores [1] [1]
('D', 4)
```

Como esses valores são tuplas, precisaríamos acessar o primeiro valor na tupla para obter apenas a letra possível sem seu valor de pontuação de correspondência de frequência. Cada letra é armazenada no primeiro índice das tuplas, então usariamos `allFreqScores [1] [0] [0]` para acessar a letra mais provável da primeira subchave, `allFreqScores [1] [1] [0]` para acessar o carta mais provável da segunda subchave e assim por diante:

```
>>> allFreqScores [1] [0] [0]
'S'
>>> allFreqScores [1] [1] [0]
'D'
```

Uma vez que você consiga acessar subchaves potenciais em todos os `FreqScores`, você precisa combiná-las para encontrar chaves em potencial.

Criando combinações de subchaves com `itertools.product()`

As tuplas produzidas por `itertools.product()` representam uma chave onde a posição na tupla corresponde ao primeiro índice que acessamos em `allFreqScores`, e os inteiros na tupla representam o segundo índice que acessamos em `allFreqScores`.

Como definimos a constante `NUM_MOST_FREQ LETTERS` como 4 anteriormente, `itertools.product (range (NUM_MOST_FREQ LETTERS), repeat = mostLikelyKeyLength)` na linha 188 faz com que o loop for tenha uma tupla de inteiros (de 0 a 3) representando as quatro letras mais prováveis de cada subchave para a variável de índices :

188. para índices em `itertools.product (range (NUM_MOST_FREQ LETTERS), repeat = mostLikelyKeyLength)`:

189. # Crie uma chave possível a partir das letras em `allFreqScores`:

```
190. possibleKey = "
191. para i em range (mostLikelyKeyLength):
192. possívelKey += allFreqScores [i] [indexes [i]] [0]
```

Nós construímos chaves Vigenere completas usando índices , o que leva o valor de uma tupla criada por `itertools.product ()` em cada iteração. A chave inicia como uma string em branco na linha 190, e o loop for na linha 191 percorre os números inteiros de 0 até, mas não inclui, a maioria de `mostLikelyKeyLength` para cada tupla para construir uma chave.

Como a variável `i` muda para cada iteração do loop for , o valor em `índices [i]` é o índice da tupla que queremos usar em `allFreqScores [i]` . É por isso que `allFreqScores [i] [indexes [i]]` é avaliado para a tupla correta desejada. Quando temos a tupla correta, precisamos acessar o índice 0 nessa tupla para obter a letra da subchave.

Se `SILENT_MODE` for `False` , a linha 195 imprimirá a chave que foi criada pelo loop for na linha 191:

```
194. se não SILENT_MODE:
195. print ("Tentativa com chave:% s'% (possívelKey))
```

Agora que temos uma chave Vigenère completa, as linhas 197 a 208 descriptografam o texto cifrado e verificam se o texto descriptografado é legível em inglês. Se estiver, o programa imprime na tela para que o usuário confirme se é realmente inglês para verificar falsos positivos.

Imprimir o texto descriptografado com a caixa correta

Como o `decryptedText` está em letras maiúsculas, as linhas 201 a 207 criam uma nova string anexando uma forma maiúscula ou minúscula das letras em `decryptedText` à lista `origCase` :

```
197. decryptedText = vigenereCipher.decryptMessage (possívelKey,
ciphertextUp)
198.
199. if detectEnglish.isEnglish (decryptedText):
200. # Defina o texto cifrado hackeado para a caixa original:
201. origCase = []
202. para i na faixa (len (texto cifrado)):
203. se texto cifrado [i] .isupper ():
204. origCase.append (decryptedText [i] .upper ())
```

205. mais:

206. origCase.append (decryptedText [i] .lower ())

207. decryptedText = " .join (origCase)

O loop for na linha 202 percorre cada um dos índices na cadeia de texto cifrado , que, ao contrário de ciphertextUp , tem a caixa original do texto cifrado . Se o texto cifrado [i] for maiúsculo, a forma maiúscula de decryptedText [i] é anexada ao origCase . Caso contrário, a forma minúscula de decryptedText [i] é anexada. A lista no origCase é então unida na linha 207 para se tornar o novo valor de decryptedText .

As próximas linhas de código imprimem a saída de descriptografia ao usuário para verificar se a chave foi encontrada:

210. print ('Possível corte de criptografia com chave% s:'% (possívelKey))

211. print (decryptedText [: 200]) # Mostra somente os primeiros 200 caracteres.

212. print ()

213. print ('Digite D se estiver pronto, qualquer outra coisa para continuar o hacking:')

214. response = input ('>')

215.

216. if response.strip (). Upper (). Startswith ('D'):

217. return decryptedText

O texto descriptografado corretamente encapsulado é impresso na tela para o usuário confirmar que é o inglês. Se o usuário inserir 'D' , a função retornará a string decryptedText .

Caso contrário, se nenhuma das decifrações parecer com o inglês, o hacking falhou e o valor None é retornado:

220. retorno Nenhum

Retornando a mensagem hackeada

Finalmente, todas as funções que definimos serão usadas pela função hackVigenere () , que aceita uma string de texto cifrado como um argumento e retorna a mensagem hackeada (se hackear foi bem sucedida) ou None (se não foi). Ele começa obtendo os comprimentos de chave prováveis com kasiskiExamination () :

223. def hackVigenere (texto cifrado):

```
224. # Primeiro, precisamos fazer o exame de Kasiski para descobrir qual é o  
225. # comprimento da chave de criptografia do texto cifrado:  
226. allLikelyKeyLengths = kasiskiExamination (ciphertext)
```

A saída da função hackVignere () depende se o programa está em SILENT_MODE :

```
227. se não SILENT_MODE:  
228. keyLengthStr = "  
229. para keyLength em allLikelyKeyLengths:  
230. keyLengthStr += '% s' % (keyLength)  
231. print ('Os resultados do exame da Kasiski indicam que os comprimentos de  
chave mais prováveis  
são:' + keyLengthStr + '\ n')
```

Os comprimentos de chave prováveis são impressos na tela se SILENT_MODE for False .

Em seguida, precisamos encontrar letras de subchaves prováveis para cada comprimento de chave. Nós vamos fazer isso com outro loop que tenta hackar a cifra com cada tamanho de chave que encontramos.

Rompendo o loop quando uma chave em potencial é encontrada

Queremos que o código continue dando um loop e verificando comprimentos de chave até encontrar um comprimento de chave potencialmente correto. Quando encontrar um tamanho de chave que pareça correto, interromperemos o loop com uma instrução break .

Semelhante a como a instrução continue é usada dentro de um loop para retornar ao início do loop, a instrução break é usada dentro de um loop para sair imediatamente do loop. Quando a execução do programa sai de um loop, move imediatamente para a primeira linha de código depois que o loop termina. Vamos sair do loop sempre que o programa encontrar uma chave potencialmente correta e precisarmos pedir ao usuário para confirmar se a chave está correta.

```
232. hackedMessage = Nenhum  
233. para keyLength em todosLikelyKeyLengths:  
234. se não SILENT_MODE:  
235. print ('Tentando hackar com comprimento de chave% s (% s chaves  
possíveis) ...'  
% (keyLength, NUM_MOST_FREQ LETTERS ** keyLength))
```

```
236. hackedMessage = attemptHackWithKeyLength (texto cifrado, keyLength)
237. if hackedMessage! = Nenhum:
238. break
```

Para cada comprimento de chave possível, o código chama o método tryHackWithKeyLength () na linha 236. Se o método attemptHackWithKeyLength () não retornar None , o hack é bem-sucedido e a execução do programa deve sair do loop for na linha 238.

Forçando Todos os Outros Comprimentos Principais

Se o hack falhar todos os comprimentos de chave possíveis que kasiskiExamination () retornou, hackedMessage será definido como Nenhum quando a instrução if na linha 242 for executada. Nesse caso, todos os *outros* comprimentos de chave até MAX_KEY_LENGTH são tentados. Se o exame de Kasiski não conseguir calcular o tamanho correto da chave, podemos apenas forçar a força bruta através dos comprimentos de chave com o loop for na linha 245:

```
242. se hackedMessage == Nenhum:
243. se não SILENT_MODE:
244. print ('Não é possível hackear a mensagem com tamanho (s) de chave provável (s).
Forçando o tamanho da chave ...')
245. for keyLength in range (1, MAX_KEY_LENGTH + 1):
246. # Não repique os comprimentos de chaves já experimentados no Kasiski:
247. if keyLength não em allLikelyKeyLengths:
248. if not SILENT_MODE:
249. print ('Tentando hackar com comprimento de chave% s (% s
chaves possíveis) ) ... % (keyLength, NUM_MOST_FREQ LETTERS **keyLength))
250. hackedMessage = attemptHackWithKeyLength (texto cifrado,
keyLength)
251. if hackedMessage! = Nenhum:
252. quebrar
```

A linha 245 inicia um loop for que chama o método tryHackWithKeyLength () para cada valor de keyLength (que varia de 1 a MAX_KEY_LENGTH), desde que não esteja em allLikelyKeyLengths . A razão é que os comprimentos de chave em allLikelyKeyLengths já foram tentados no código nas linhas 233 a

238.

Finalmente, o valor em hackedMessage é retornado na linha 253:

253. return hackedMessage

Chamando a função main ()

As linhas 258 e 259 chamam a função main () se este programa foi executado por si só, em vez de ser importado por outro programa:

256. # Se vigenereHacker.py for executado (em vez de importado como um módulo), chame

257. # a função main ():

258. if __name__ == '__main__':

259. main ()

Esse é o programa completo de hackers da Vigenère. Se é bem sucedido depende das características do texto cifrado. Quanto mais próxima a frequência de letras do texto original for a frequência de letras do inglês regular e quanto mais longo for o texto simples, maior a probabilidade de o programa de hacking funcionar.

Modificando as constantes do programa Hacking

Podemos modificar alguns detalhes se o programa de hackers não funcionar. Três constantes que definimos nas linhas 8 a 10 afetam o funcionamento do programa de hackers:

8. MAX_KEY_LENGTH = 16 # Não tentará chaves mais longas que isso.

9. NUM_MOST_FREQ LETTERS = 4 # Tente esta muitas letras por subchave.

10. SILENT_MODE = False # Se definido como True, o programa não imprime nada.

Se a chave Vigenère for maior que o inteiro em MAX_KEY_LENGTH na linha 8, não há como o programa hacker encontrar a chave correta. Se o programa de invasão não conseguir cortar o texto cifrado, tente aumentar esse valor e executar o programa novamente.

Lembre-se de que tentar hackear um comprimento incorreto de chave leva pouco tempo. Mas se MAX_KEY_LENGTH for definido muito alto e a função kasisikiExamination () erroneamente achar que o tamanho da chave pode ser um inteiro enorme, o programa pode passar horas, ou mesmo meses, tentando hackear o texto cifrado usando os comprimentos de chave incorretos.

To prevent this, NUM_MOST_FREQ LETTERS on line 9 limits the number of possible letters tried for each subkey. By increasing this value, the hacking program tries many more keys, which you might need to do if the freqAnalysis.englishFreqMatchScore() was inaccurate for the original plaintext message, but this also causes the program to slow down. And setting NUM_MOST_FREQ LETTERS to 26 would cause the program to skip narrowing down the number of possible letters for each subkey entirely!

For both MAX_KEY_LENGTH and NUM_MOST_FREQ LETTERS , a smaller value is faster to execute but less likely to succeed in hacking the cipher, and a larger value is slower to execute but more likely to succeed.

Finalmente, para aumentar a velocidade do seu programa, você pode definir SILENT_MODE para True na linha 10, para que o programa não perca tempo imprimindo informações na tela. Embora o seu computador possa realizar cálculos rapidamente, a exibição de caracteres na tela pode ser relativamente lenta. A desvantagem de não imprimir informações é que você não saberá como o programa está sendo executado até que esteja completamente concluído.

Resumo

A invasão da criptografia Vigenère exige que você siga várias etapas detalhadas. Além disso, muitas partes do programa de hacking podem falhar: por exemplo, talvez a chave Vigenère usada para criptografia seja maior que MAX_KEY_LENGTH , ou talvez a função de correspondência de frequência em inglês tenha recebido resultados imprecisos porque o texto não segue a freqüência normal das letras ou talvez o texto simples tem muitas palavras que não estão no arquivo do dicionário e o isEnglish () não o reconhece como inglês.

Ao identificar diferentes maneiras pelas quais o programa de hacking pode falhar, você pode alterar o código para lidar com esses casos. Mas o programa de hacking neste livro faz um bom trabalho de reduzir bilhões ou trilhões de chaves possíveis para meros milhares.

No entanto, há um truque que torna a cifra Vigenère matematicamente impossível de quebrar, não importa o quanto poderoso seja o seu computador ou quanto inteligente seja seu programa de hackers. Você aprenderá sobre esse truque, chamado de um bloco único, no [Capítulo 21](#) .

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em

<https://www.nostarch.com/crackingcodes/> .

1. O que é um ataque de dicionário?
2. O que Kasiski examina sobre um texto cifrado?
3. Quais são as duas alterações ao converter um valor de lista em um valor definido com a função set () ?
4. Se a variável spam contiver ['cat', 'dog', 'mouse', 'dog'] , esta lista contém quatro itens. Quantos itens a lista retornou da lista (set (spam)) ?
5. O que o código a seguir imprime?

```
print ('Hello', end = "")  
print ('Mundo')
```

21

O CCD DE ALMOFADA

“Eu já superei isso mil vezes”, diz Waterhouse, “e a única explicação que posso pensar é que eles estão convertendo suas mensagens em grandes números binários e então combinando-os com outros grandes números binários - blocos únicos, provavelmente.”

“Nesse caso, seu projeto está condenado”, diz Alan, “porque você não pode quebrar um bloco único”.

-Neal Stephenson, Cryptonomicon



Neste capítulo, você aprenderá sobre uma cifra impossível de decifrar, não importa o quanto seu computador seja poderoso, quanto tempo você gasta tentando decifrá-lo ou quão inteligente você é um hacker. Chama -se *cifra de pad one-time* , e a boa notícia é que não precisamos escrever um novo programa para usá-lo! O programa de codificação Vigenère que você escreveu no [Capítulo 18](#) pode implementar essa codificação sem quaisquer alterações. Mas a codificação de pad de uma só vez é tão inconveniente de usar em uma base

regular que muitas vezes é reservada para o mais secreto de mensagens.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A codificação de bloco único inquebrável
- O bloco de duas vezes é a cifra Vigenère

A criptografia de bloco único inquebrável

A codificação de pad de uso único é uma codificação Vigenère que se torna inquebrável quando a chave atende aos seguintes critérios:

1. É exatamente o tempo que a mensagem criptografada.
2. É composto de símbolos verdadeiramente aleatórios.
3. É usado apenas uma vez e nunca mais para qualquer outra mensagem.

Seguindo estas três regras, você pode tornar sua mensagem criptografada invulnerável ao ataque de qualquer criptoanalista. Mesmo com poder computacional infinito, a cifra não pode ser quebrada.

A chave para a cifra do pad de uso único é chamada de *pad* porque as chaves costumavam ser impressas em blocos de papel. Depois que a folha de papel superior foi usada, ela seria arrancada do bloco para revelar a próxima chave a ser usada. Geralmente, uma grande lista de chaves de acesso único é gerada e compartilhada pessoalmente, e as chaves são marcadas para datas específicas. Por exemplo, se recebêssemos uma mensagem de nosso colaborador em 31 de outubro, examinariamos a lista de blocos únicos para encontrar a chave correspondente naquele dia.

Comprimento da chave igual comprimento da mensagem

Para entender por que a codificação de pad de uso único é inquebrável, vamos pensar sobre o que torna a cifra Vigenère comum vulnerável a ataques. Lembre-se de que o programa de hackers de código Vigenère funciona usando a análise de frequência. Mas, se a chave tiver o mesmo comprimento da mensagem, a subchave de cada letra de texto simples será única, o que significa que cada letra de texto simples pode ser criptografada em qualquer letra de texto cifrada com a mesma probabilidade.

Por exemplo, para criptografar a mensagem SE VOCÊ QUISER SOBREVIVER AQUI, VOCÊ TEM QUE SABER ONDE SEU TOALHO, removemos os espaços e a pontuação para receber uma mensagem com 55 letras. Para

criptografar esta mensagem usando um bloco único, precisamos de uma chave com 55 letras. Usar a chave de exemplo

KCQYZHEPXAUTIQUEKXEJMORETZHZTRWWQDYLBTTEJMEDBSANY para criptografar a sequência resultaria no texto cifrado

SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUER conforme mostrado na [Figura 21-1](#) .

Plaintext IFYOUWANTTOSURVIVEOUTHEREYOUVEGOTTOKNOWWHEREYOURTOWELIS

Key KCQYZHEPXAUTIQUEKXEJMORETZHZTRWWQDYLBTTEJMEDBSANYBPXQIK

Ciphertext SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

Figura 21-1: Criptografando uma mensagem de exemplo usando um bloco único

Agora imagine que um criptoanalista se apodera do texto cifrado (SHOM TDEC ...). Como eles poderiam atacar a cifra? Impulsionar brute através das teclas não funcionaria, porque são muitas, mesmo para um computador. O número de chaves seria igual a 26 elevado à potência do número total de letras na mensagem. Portanto, se a mensagem tiver 55 letras, como no nosso exemplo, haveria um total de 26^{55} , ou 666,091,878,431,395,624,153,823,182, 526,730,590,376,250,379,528,249,805,353,030,484,209,594,192,101,376 chaves possíveis.

Mesmo que o criptoanalista tivesse um computador poderoso o bastante para testar todas as teclas, ele ainda não seria capaz de quebrar a codificação do pad de uso único *porque, para qualquer texto cifrado, todas as mensagens de texto simples possíveis são igualmente prováveis* .

Por exemplo, o texto cifrado SHOMTDEC ... poderia facilmente ter resultado de um texto plano completamente diferente com o mesmo número de letras, como a chave criptografada usando a chave

ZAKAVKXOLFQDLZHWSQJBZMTWMMNAWWWDCDCYWKSGORGH como mostrado na [Figura 21. -2](#) .

Plaintext THEMYTHOFOSIRISWASOFIMPORTANCEINANCIENTEGYPTIANRELIGION

Key ZAKAVKXOLFQDLZHWSQJBZMTWMMNAKWURWEXDCUYWKSGORGHNEDVTCP

Ciphertext SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

Figura 21-2: Criptografando uma mensagem de exemplo diferente usando uma chave diferente, mas produzindo o mesmo texto cifrado de antes

A razão pela qual podemos hackear qualquer criptografia é que sabemos que geralmente há apenas uma chave que descriptografa a mensagem para um inglês sensato. Mas acabamos de ver no exemplo anterior que o *mesmo* texto cifrado

poderia ter sido feito usando duas mensagens de texto simples muito *diferentes*. Quando usamos o pad de uso único, o criptoanalista não tem como dizer qual é a mensagem original. Na verdade, *qualquer* mensagem de texto simples em inglês legível que tenha exatamente 55 letras de tamanho é *a mesma que provavelmente* será o texto original. Só porque uma determinada chave pode descriptografar o texto cifrado para legível em inglês não significa que é a chave de criptografia original.

Como qualquer texto simples em inglês poderia ter sido usado para criar um texto cifrado com igual probabilidade, é impossível hackear uma mensagem criptografada usando um bloco único.

Fazendo a chave verdadeiramente aleatória

Como você aprendeu no [Capítulo 9](#), o módulo aleatório embutido no Python não gera números verdadeiramente aleatórios. Eles são calculados usando um algoritmo que cria números que parecem apenas aleatórios, o que é bom o suficiente na maioria dos casos. No entanto, para o one-time pad funcionar, o pad deve ser gerado a partir de uma fonte verdadeiramente aleatória; caso contrário, perde o seu segredo matematicamente perfeito.

O Python 3.6 e posteriores têm o módulo secrets, que usa a origem do sistema operacional de números verdadeiramente aleatórios (geralmente coletados de eventos aleatórios, como o tempo entre as teclas digitadas pelo usuário). Os segredos da função `.randbelow()` pode retornar números verdadeiramente aleatórios entre 0 e até, mas não incluindo o argumento passado para ele, como neste exemplo:

```
>>> segredos de importação
>>> secrets.randbelow(10)
2
>>> secrets.randbelow(10)
0
>>> secrets.randbelow(10)
6
```

As funções nos segredos são mais lentas que as funções aleatórias, então as funções aleatórias são preferidas quando a aleatoriedade verdadeira não é necessária. Você também pode usar a função `secrets.choice()`, que retorna um valor escolhido aleatoriamente da string ou da lista passada para ele, como neste exemplo:

```
>>> segredos de importação
>>> secrets.choice ('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
'R'
>>> secrets.choice (['cat', 'dog', 'mouse'])
'cão'
```

Para criar um bloco único verdadeiramente aleatório com 55 caracteres, por exemplo, use o seguinte código:

```
>>> segredos de importação
>>> otp = "
>>> para i na faixa (55):
    otp + = secrets.choice ('ABCDEFGHIJKLMNOPQRSTUVWXYZ')

>>> otp
'MVOVAAYDPELIRNRUZNNQHDNSOUWWNPJUPIUAIMKFKNHQANI'
```

Há mais um detalhe que devemos ter em mente ao usar o bloco único. Vamos examinar por que precisamos evitar usar a mesma chave de acesso único mais de uma vez.

Evitando o Two-Time Pad

Uma *cifra de dois tempos* refere-se ao uso da mesma chave de um só passo para criptografar duas mensagens diferentes. Isso cria uma fraqueza em uma criptografia.

Como mencionado anteriormente, só porque uma chave descriptografa o texto cifrado de um teclado para ler em inglês não significa que é a chave correta. No entanto, quando você usa a mesma chave para duas mensagens diferentes, você está dando informações cruciais para o hacker. Se você criptografar duas mensagens usando a mesma chave e um hacker encontrar uma chave que descriptografa o primeiro texto cifrado para o inglês legível, mas descriptografar a segunda mensagem em texto aleatório, o hacker saberá que a chave encontrada não deve ser a chave original. De fato, é altamente provável que exista apenas uma chave que descriptografe *ambas as* mensagens para o inglês, como você verá na próxima seção.

Se o hacker tiver apenas uma das duas mensagens, essa mensagem ainda estará perfeitamente criptografada. Mas devemos sempre presumir que *todas as* nossas mensagens criptografadas estão sendo interceptadas por hackers e governos.

Caso contrário, não nos incomodaríamos em criptografar mensagens em primeiro lugar. A máxima de Shannon é importante ter em mente: o inimigo conhece o sistema! Isso inclui todo o seu texto cifrado.

Por que o bloco de duas vezes é a criptografia Vigenère

Você já aprendeu como quebrar as cifras de Vigenère. Se pudermos mostrar que uma cifra de dois tempos é igual a uma cifra de Vigenère, podemos provar que ela é quebrável usando as mesmas técnicas usadas para quebrar a cifra de Vigenère.

Para explicar por que o bloco de duas vezes é hackeável como a cifra de Vigenère, vamos rever como a cifra de Vigenère funciona quando criptografa uma mensagem que é mais longa que a chave. Quando ficamos sem letras na chave para criptografar, voltamos para a primeira letra da chave e continuamos a criptografar. Por exemplo, para criptografar uma mensagem de 20 letras como BLUE IODINE INBOUND CAT com uma chave de 10 letras como YZNMPZXYXY, as 10 primeiras letras (IODINE AZUL) são criptografadas com YZNMPZXYXY e as próximas 10 letras (INBOUND CAT) também são criptografado com YZNMPZXYXY. [Figura 21-3](#) mostra esse efeito envolvente.

Plaintext	BLUEIODINEINBOUNDCAT
Vigenère key	YZNMPZXYXYZNMPZXYXY
Vigenère ciphertext	ZKHQXNAGKCGMOAJMAAXR

Figura 21-3: O efeito envolvente da cifra de Vigenère

Usando a codificação de pad de uso único, digamos que a mensagem de 10 letras BLUE IODINE é criptografada usando a tecla YZNMPZXYXY de one-time pad. Então, o criptógrafo comete o erro de criptografar uma segunda mensagem de 10 letras, INBOUND CAT, com a mesma tecla de um tempo, YZNMPZXYXY, conforme mostrado na [Figura 21-4](#).

	Message 1	Message 2
Plaintext	BLUEIODINE	INBOUNDCAT
One-time pad key	YZNMPZXYXY	YZNMPZXYXY
One-time pad ciphertext	ZKHQXNAGKC	GMOAJMAAXR

Figura 21-4: Criptografar texto simples usando um bloco único produz o mesmo texto cifrado que a cifra Vigenère.

Quando comparamos o texto cifrado da cifra Vigenère mostrada na [Figura 21-3](#) (ZKHQXNAGKCGMOAJMAAXR) aos textos cifrados da cifra de almofada

única mostrada na [Figura 21-4](#) (ZKHQXNAGKC GMOAJMAAXR), podemos ver que eles são exatamente os mesmos. Isso significa que, como a criptografia do bloco de duas vezes tem as mesmas propriedades que a codificação Vigenère, podemos usar as mesmas técnicas para cortá-la!

Resumo

Em suma, um pad de uso único é uma maneira de tornar as criptografias de criptografia Vigenère invulneráveis a hackers usando uma chave que tenha o mesmo tamanho da mensagem, seja verdadeiramente aleatória e seja usada apenas uma vez. Quando essas três condições são atendidas, é impossível quebrar o bloco único. No entanto, porque é tão inconveniente de usar, não é usado para criptografia todos os dias. Normalmente, o one-time pad é distribuído pessoalmente e contém uma lista de chaves. Mas tenha certeza que esta lista não caia nas mãos erradas!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Por que não é apresentado um programa de preenchimento único neste capítulo?
2. Qual cifra é equivalente ao bloco de duas vezes?
3. O uso de uma tecla duas vezes mais longa que a mensagem em texto simples tornaria o pad onetime duas vezes mais seguro?

22

ENCONTRANDO E GERANDO NÚMEROS PRIMEIROS

"Matemáticos tentaram em vão até hoje descobrir alguma ordem na sequência dos números primos, e temos razão para acreditar que é um mistério no qual a mente humana nunca penetrará".

- Leonhard Euler, matemático do século XVIII



Todas as cifras descritas neste livro até agora existem há centenas de anos. Essas cifras funcionavam bem quando os hackers dependiam de lápis e papel, mas são mais vulneráveis agora que os computadores podem manipular dados trilhões de vezes mais rapidamente do que uma pessoa. Outro problema com essas cifras clássicas é que elas usam a mesma chave para criptografia e descriptografia. Usar uma tecla causa problemas quando você está tentando enviar uma mensagem criptografada: por exemplo, como você pode enviar a chave para descriptografá-la com segurança?

No [Capítulo 23](#), você aprenderá como a cifra de chave pública melhora as cifras antigas usando números primos muito grandes para criar duas chaves: uma chave pública para criptografia e uma chave privada para descriptografia. Para gerar números primos para as chaves da cifra da chave pública, você precisará aprender sobre algumas propriedades dos números primos (e a dificuldade de fatorar grandes números) que tornam a cifra possível. Neste capítulo, você explorará esses recursos de números primos para criar o módulo `primeNum.py`, que pode gerar chaves determinando rapidamente se um número é primo ou não.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- números primos e compostos
- O teste de primalidade da divisão experimental
- A peneira de Eratóstenes
- O teste de primalidade de Rabin-Miller

O que é um número primo?

Um *número primo* é um número inteiro maior que 1 e possui apenas dois fatores: 1 e ele mesmo. Lembre-se de que os fatores de um número são aqueles números que podem ser multiplicados para igualar o número original. Por exemplo, os números 3 e 7 são fatores de 21. O número 12 tem fatores 2 e 6, bem como 3 e 4.

Cada número tem fatores de 1 e ele próprio porque 1 multiplicado por qualquer número sempre será igual a esse número. Por exemplo, 1 e 21 são fatores de 21 e

os números 1 e 12 são fatores de 12. Se não houver outros fatores para um número, o número é primo. Por exemplo, 2 é um número primo porque tem apenas 1 e 2 como seus fatores.

Aqui está uma pequena lista de números primos (note que 1 não é considerado um número primo): 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281 e assim por diante.

Existe um número infinito de números primos, o que significa que não existe o maior número primo. Eles continuam a ficar maiores e maiores, assim como os números regulares. A cifra de chave pública usa grandes números primos para tornar a chave muito grande para a força bruta.

Números primos podem ser difíceis de encontrar e grandes números primos, como aqueles usados para chaves públicas, são ainda mais difíceis de encontrar. Para gerar números primos grandes como chaves públicas, vamos encontrar um grande número aleatório e, em seguida, verificar se o número é primo usando um *teste de primalidade*. Se o número for primo de acordo com o teste de primalidade, nós o usaremos; caso contrário, continuaremos criando e testando números grandes até encontrarmos um que seja primo.

Vamos ver alguns números muito grandes para ilustrar o tamanho dos números primos usados na codificação da chave pública.

Um *googol* é 10 elevado à potência de 100 e é escrito como 1 seguido de 100 zeros:

10.000.000.000.000.000.000.000.000.000.000.000.000,
000.000.000.000.000.000.000.000.000.000.000.000,
000.000.000.000

Um bilhão de bilhões de bilhões de googols tem mais 27 zeros que um googol:

10.000.000.000.000.000.000.000.000.000.000.000.000.000,
000.000.000.000.000.000.000.000.000.000.000.000.000,
000.000.000.000.000.000.000.000.000.000.000.000.000

Mas esses são números pequenos em comparação com os números primos usados pela codificação de chave pública. Por exemplo, um número primo típico usado no programa de chave pública tem centenas de dígitos e pode se parecer com algo assim:

112,829,754,900,439,506,175,719,191,782,841,802,172,556,768,
253.593.054.977.186.2355.84.979.780.304.652.423.405.148.425,
447.063.090.165.759.070.742.102.132.335.103.295.947.000.718,
386.333.756.395.799.633.478.227.612.244.071.875.721.006.813,
307.628.061.280.861.610.153.485.352.017.238.548.269.452.852,
733.818.231.045.171.038.838.387.845.888.589.411.762.622.041,
204.120.706.150.518.465.720.862.068.595.814.264.819

Esse número é tão grande que eu aposto que você nem notou o erro.

Algumas outras características interessantes dos números primos também são úteis para se conhecer. Como todos os números pares são múltiplos de dois, 2 é o único número primo possível. Além disso, a multiplicação de dois números primos deve resultar em um número cujos únicos fatores são 1, ele próprio e os dois números primos que foram multiplicados. (Por exemplo, multiplicando números primos 3 e 7 resulta em 21, cujos únicos fatores são 1, 21, 3 e 7.)

Inteiros que não são primos são chamados de *números compostos* porque são compostos de pelo menos dois fatores além de 1 e o número. Todo número composto tem uma *fatoração primária*, que é uma fatoração composta apenas de números primos. Por exemplo, o número composto 1386 é composto dos números primos 2, 3, 7 e 11, porque $2 \times 3 \times 3 \times 7 \times 11 = 1386$. A fatoração primária de cada número composto é exclusiva desse número composto.

Usaremos essas informações sobre o que faz um número primo escrever um módulo que possa determinar se um número pequeno é primo e gerar números primos. O módulo, *primeNum.py*, definirá as seguintes funções:

`isPrimeTrialDiv()` usa o algoritmo de divisão de avaliação para retornar True se o número passado a ele for primo ou False se o número passado a ele não for primo.

`primeSieve()` usa a peneira do algoritmo de Eratóstenes para gerar números primos.

`rabinMiller()` usa o algoritmo de Rabin-Miller para verificar se o número passado para ele é primo. Este algoritmo, ao contrário do algoritmo de divisão experimental, pode trabalhar rapidamente em números muito grandes. Essa função é chamada não diretamente, mas sim por `isPrime()`.

`isPrime()` é chamado quando o usuário deve determinar se um inteiro grande é primo ou não.

`generateLargePrime()` retorna um número primo grande com centenas de dígitos. Esta função será usada no programa `makePublicPrivateKeys.py` no [Capítulo 23](#).

Código-fonte para o módulo Prime Numbers

Como o `cryptomath.py`, introduzido no [Capítulo 13](#), o programa `primeNum.py` deve ser importado como um módulo por outros programas e não faz nada quando executado por conta própria. O módulo `primeNum.py` importa os módulos matemáticos e aleatórios do Python para usar ao gerar números primos.

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**.
Digite o seguinte código no editor de arquivos e salve-o como `primeNum.py`.

`primeNum.py`

```
1. # peneira número principal
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importar matemática, aleatória
5
6
7. def isPrimeTrialDiv (num):
8. # Retorna True se num for um número primo, caso contrário, False.
9
10. # Usa o algoritmo da divisão experimental para testar a primalidade.
11
12. # Todos os números menores que 2 não são primos:
13. se num <2:
14. retornar False
15
16. # Veja se num é divisível por qualquer número até a raiz quadrada de num:
17. para i no intervalo (2, int (math.sqrt (num)) + 1):
18. if num% i == 0:
19. retornar falso
20. retornar Verdadeiro
21
22
23. def primeSieve (peneira):
24. # Retorna uma lista de números primos calculados usando
```

25. # o algoritmo de Crivo de Eratóstenes.

26

27. peneira = [Verdadeiro] * peneira

28. peneira [0] = Falso # Zero e um não são números primos.

29. peneira [1] = Falso

30

31. # Crie a peneira:

32. para i no intervalo (2, int (math.sqrt (sieveSize)) + 1):

33. ponteiro = i * 2

34. while ponteiro <sieveSize:

35. peneira [ponteiro] = Falso

36. ponteiro += i

37

38. # Compile a lista de primos:

39. primes = []

40. para i na faixa (sieveSize):

41. if peneira [i] == Verdadeiro:

42. primes.append (i)

43

44. primos de retorno

45

46. def rabinMiller (num):

47. # Retorna True se num for um número primo.

48. if num% 2 == 0 ou num <2:

49. return False # Rabin-Miller não trabalha com números inteiros iguais.

50. if num == 3:

51. return Verdadeiro

52. s = num - 1

53. t = 0

54. enquanto s% 2 == 0:

55. # Manter a metade até que seja estranho (e use t

56. # para contar quantas vezes nós dividimos pela metade):

57. s = s // 2

58. t += 1

59. para ensaios no intervalo (5): # Tente falsificar a primalidade num 5 vezes.

60. a = random.randrange (2, num - 1)

61. v = pow (a, s, num)

```
62. if v! = 1: # Este teste não se aplica se v for 1.
63. i = 0
64. enquanto v! = (Num - 1):
65. se eu == t - 1:
66. return False
67. else:
68. i = i + 1
69. v = (v ** 2)% num
70. return Verdadeiro
71
72. # Na maior parte do tempo podemos determinar rapidamente se num não é
primo
73. # dividindo pelas primeiras dúzias de números primos. Isso é mais rápido
74. # que rabinMiller (), mas não detecta todos os compostos.
75. LOW_PRIMES = primeSieve (100)
76
77
78. def isPrime (num):
79. # Retorna True se num for um número primo. Esta função é mais rápida
80. # número primo verifique antes de chamar rabinMiller ().
81. if (num <2):
82. return False # 0, 1, e os números negativos não são primos.
83. # Veja se algum dos números primos baixos pode dividir num:
84. para prime em LOW_PRIMES:
85. if (num == prime):
86. return Verdadeiro
87. if (num% primo == 0):
88. retornar falso
89. # Se tudo mais falhar, chame rabinMiller () para determinar se num é primo:
90. retornar rabinMiller (num)
91
92
93. def generateLargePrime (tamanho da chave = 1024):
94. # Retorna um número primo aleatório que tem tamanho de tamanho de bits:
95. enquanto Verdadeiro:
96. num = random.randrange (2 ** (tamanho da chave-1), 2 ** (tamanho da
chave))
```

```
97. if isPrime (num):  
98.     retornar num
```

Execução de Amostra do Módulo de Números Principais

Para ver a saída de amostra do módulo *primeNum.py* , insira o seguinte no shell interativo:

```
>>> import primeNum  
>>> primeNum.generateLargePrime ()  
12288116834221104103052368351544323900748429060070155536948827174  
46375131251147129101194573241337844666680914050203700367321105215  
99056307685956683501638255651896712492153821239703634581598364114  
63721834845554443590842840019256584962050960031246875795389955344  
>>> primeNum.isPrime (45943208739848451)  
Falso  
>>> primeNum.isPrime (13)  
Verdade
```

A importação do módulo *primeNum.py* nos permite gerar um número primo muito grande usando a função *generateLargePrime ()* . Ele também nos permite passar qualquer número, grande ou pequeno, para a função *isPrime ()* para determinar se é um número primo.

Como o Algoritmo da Divisão de Julgamento funciona

Para descobrir se um dado número é ou não primo, usamos o *algoritmo de divisão de teste* . O algoritmo continua a dividir um número por números inteiros (começando com 2, 3 e assim por diante) para ver se algum deles divide o número com 0 como o restante. Por exemplo, para testar se 49 é primo, podemos tentar dividi-lo por inteiros começando com 2:

$$49 \div 2 = 24 \text{ resto } 1$$

$$49 \div 3 = 16 \text{ resto } 1$$

$$49 \div 4 = 12 \text{ resto } 1$$

$$49 \div 5 = 9 \text{ restante } 4$$

$$49 \div 6 = 8 \text{ restantes } 1$$

$$49 \div 7 = 7 \text{ resto } 0$$

Como 7 divide uniformemente 49 com um resto de 0, sabemos que 7 é um fator

de 49. Isso significa que 49 não pode ser um número primo, porque tem pelo menos um fator diferente de 1 e ele próprio.

Podemos agilizar esse processo dividindo apenas números primos, não números compostos. Como mencionado anteriormente, os números compostos são nada mais que *compostos* de números primos. Isto significa que se 2 não pode dividir 49 uniformemente, então um número composto como 6, cujos fatores incluem 2, não ser capaz de dividir 49 igualmente. Em outras palavras, *qualquer* número que 6 divide uniformemente também pode ser dividido por 2 uniformemente, porque 2 é um fator de 6. A [Figura 22-1](#) ilustra esse conceito.

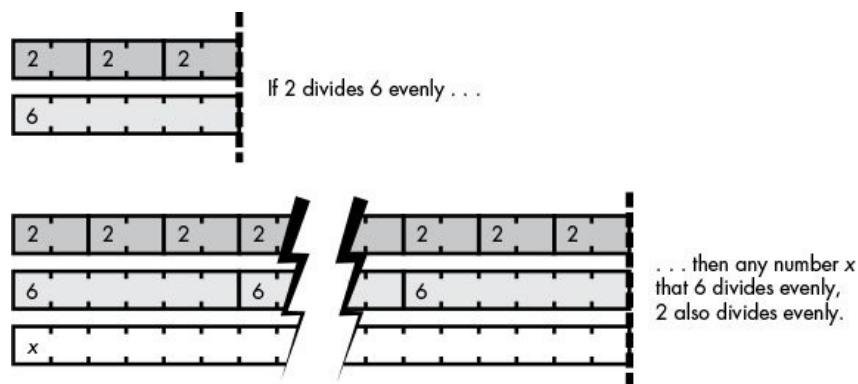


Figura 22-1: Qualquer número que divida uniformemente por 6 também divide uniformemente por 2.

Como outro exemplo, vamos testar se 13 é primo:

$$13 \div 2 = 6 \text{ restante } 1$$

$$13 \div 3 = 4 \text{ resto } 1$$

Nós só temos que testar inteiros até (e incluindo) a raiz quadrada do número que estamos testando para primality. A *raiz quadrada* de um número refere-se ao número que, quando multiplicado por si, resulta no número original. Por exemplo, a raiz quadrada de 25 é 5 porque $5 \times 5 = 25$. Como um número não pode ter dois fatores maiores que sua raiz quadrada, podemos limitar o teste do algoritmo de divisão de teste a inteiros menores que a raiz quadrada do número. A raiz quadrada de 13 é de cerca de 3,6, então só precisamos dividir por 2 e 3 para determinar que 13 é primo.

Como outro exemplo, o número 16 tem uma raiz quadrada de 4. Multiplicar dois números maiores que 4 sempre resultará em um número maior que 16, e qualquer fator de 16 maior que 4 sempre será emparelhado com fatores menores que 4, como 8×2 . Portanto, você encontrará todos os fatores maiores que a raiz

quadrada encontrando quaisquer fatores menores que a raiz quadrada.

Para encontrar a raiz quadrada de um número no Python, você pode usar a função `math.sqrt()`. Digite o seguinte no shell interativo para ver alguns exemplos de como essa função funciona:

```
>>> importar matemática  
>>> 5 * 5  
25  
>>> math.sqrt(25)  
5,0  
>>> math.sqrt(10)  
3,1622776601683795
```

Observe que `math.sqrt()` sempre retorna um valor de ponto flutuante.

Implementando o Teste de Algoritmo da Divisão de Avaliação

A função `isPrimeTrialDiv()` na linha 7 em `primeNum.py` usa um número como o parâmetro `num` e usa o teste do algoritmo de divisão de teste para verificar se o número é primo. A função retorna `False` se `num` for um número composto e `True` se `num` for um número primo.

```
7. def isPrimeTrialDiv (num):  
8. # Retorna True se num for um número primo, caso contrário, False.  
9  
10. # Usa o algoritmo da divisão experimental para testar a primalidade.  
11  
12. # Todos os números menores que 2 não são primos:  
13. se num <2:  
14. retornar False
```

A linha 13 verifica se `num` é menor que 2 e, se estiver, a função retorna `False`, porque um número menor que 2 não pode ser primo.

A linha 17 inicia o loop `for` que implementa o algoritmo de divisão de avaliação. Ele também usa a raiz quadrada de `num` usando `math.sqrt()` e usa o valor de ponto flutuante retornado para definir o limite superior do intervalo de inteiros que vamos testar.

```
16. # Veja se num é divisível por qualquer número até a raiz quadrada de num:  
17. para i no intervalo (2, int (math.sqrt (num)) + 1):
```

18. if num% i == 0:
19. retornar falso
20. retornar Verdadeiro

A linha 18 verifica se o restante é 0 usando o operador mod (%). Se o restante for 0, num é divisível por i e, portanto, não é um número primo e o loop retornará False . Se o loop for na linha 17 nunca retornar False , a função retornará True na linha 20 para indicar que num é provavelmente um número primo.

O algoritmo de divisão de avaliação na função isPrimeTrialDiv () é útil, mas não é a única maneira de testar a primalidade. Você também pode encontrar números primos usando a peneira de Eratóstenes.

A peneira de Eratóstenes

A *peneira de Eratóstenes* (pronuncia-se "era-taws-thuh-joelhos") é um algoritmo que encontra todos os números primos dentro de um intervalo de números. Para ver como esse algoritmo funciona, imagine um grupo de caixas. Cada caixa contém um número inteiro de 1 a 50, todos marcados como primos, conforme mostrado na [Figura 22-2](#) .

Para implementar a peneira de Eratóstenes, eliminamos os números não-primos de nosso intervalo até que apenas os números primos permaneçam. Como 1 nunca é primo, vamos começar marcando o número 1 como "não primo". Então vamos marcar todos os múltiplos de dois (exceto para 2) como "não primos". Isso significa que marcaremos os inteiros 4 (2×2), 6 (2×3), 8 (2×4), 10, 12, e assim por diante até 50 como “não prime”, como mostrado na [Figura 22-3](#) .

Prime 1	Prime 2	Prime 3	Prime 4	Prime 5	Prime 6	Prime 7	Prime 8	Prime 9	Prime 10
Prime 11	Prime 12	Prime 13	Prime 14	Prime 15	Prime 16	Prime 17	Prime 18	Prime 19	Prime 20
Prime 21	Prime 22	Prime 23	Prime 24	Prime 25	Prime 26	Prime 27	Prime 28	Prime 29	Prime 30
Prime 31	Prime 32	Prime 33	Prime 34	Prime 35	Prime 36	Prime 37	Prime 38	Prime 39	Prime 40
Prime 41	Prime 42	Prime 43	Prime 44	Prime 45	Prime 46	Prime 47	Prime 48	Prime 49	Prime 50

Figura 22-2: Configurando a peneira de Eratóstenes para números de 1 a 50

Not Prime 1	Prime 2	Prime 3	Not Prime 4	Prime 5	Not Prime 6	Prime 7	Not Prime 8	Prime 9	Not Prime 10
Prime 11	Not Prime 12	Prime 13	Not Prime 14	Prime 15	Not Prime 16	Prime 17	Not Prime 18	Prime 19	Not Prime 20
Prime 21	Not Prime 22	Prime 23	Not Prime 24	Prime 25	Not Prime 26	Prime 27	Not Prime 28	Prime 29	Not Prime 30
Prime 31	Not Prime 32	Prime 33	Not Prime 34	Prime 35	Not Prime 36	Prime 37	Not Prime 38	Prime 39	Not Prime 40
Prime 41	Not Prime 42	Prime 43	Not Prime 44	Prime 45	Not Prime 46	Prime 47	Not Prime 48	Prime 49	Not Prime 50

Figura 22-3: Eliminando o número 1 e todos os números pares

Em seguida, repetimos o processo usando múltiplos de três: excluímos 3 e marcamos 6, 9, 12, 15, 18, 21 e assim por diante como "não primos". Repetimos esse processo para os múltiplos de quatro, excluindo 4, os múltiplos de cinco, exceto para 5, e assim por diante até chegarmos a múltiplos de oito. Paramos em 8 porque é maior que 7.071, que é a raiz quadrada de 50. Todos os múltiplos de 9, 10, 11 e assim por diante já terão sido marcados, porque qualquer número que é um fator maior que a raiz quadrada será emparelhado com um fator menor que a raiz quadrada, que já marcamos.

A peneira preenchida deve se parecer com a [Figura 22-4](#), com números primos mostrados em caixas brancas.

Not Prime 1	Prime 2	Prime 3	Not Prime 4	Prime 5	Not Prime 6	Prime 7	Not Prime 8	Not Prime 9	Not Prime 10
Prime 11	Not Prime 12	Prime 13	Not Prime 14	Not Prime 15	Not Prime 16	Prime 17	Not Prime 18	Prime 19	Not Prime 20
Not Prime 21	Not Prime 22	Prime 23	Not Prime 24	Not Prime 25	Not Prime 26	Not Prime 27	Not Prime 28	Prime 29	Not Prime 30
Prime 31	Not Prime 32	Not Prime 33	Not Prime 34	Not Prime 35	Not Prime 36	Prime 37	Not Prime 38	Not Prime 39	Not Prime 40
Prime 41	Not Prime 42	Prime 43	Not Prime 44	Not Prime 45	Not Prime 46	Prime 47	Not Prime 48	Not Prime 49	Not Prime 50

Figura 22-4: Números primos encontrados usando a peneira de Eratóstenes

Usando a peneira de Eratóstenes, descobrimos que os números primos menores que 50 são 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 e 47. Este algoritmo de peneira é melhor usado quando você deseja encontrar rapidamente todos os

números primos *em um determinado intervalo* de números. É muito mais rápido do que usar o algoritmo de divisão de avaliação anterior para verificar cada número individualmente.

Gerando números primos com a peneira de Eratóstenes

A função primeSieve () na linha 23 do módulo *primeNum.py* usa a peneira do algoritmo de Eratóstenes para retornar uma lista de todos os números primos entre 1 e sieveSize :

```
23. def primeSieve (peneira):  
24. # Retorna uma lista de números primos calculados usando  
25. # o algoritmo de Crivo de Eratóstenes.  
26  
27. peneira = [Verdadeiro] * peneira  
28. peneira [0] = Falso # Zero e um não são números primos.  
29. peneira [1] = Falso
```

A linha 27 cria uma lista de valores True booleanos que representam o tamanho de sieveSize . Os índices 0 e 1 são marcados como False porque 0 e 1 não são números primos.

O loop for na linha 32 passa por cada inteiro de 2 até a raiz quadrada de sieveSize :

```
31. # Crie a peneira:  
32. para i no intervalo (2, int (math.sqrt (sieveSize)) + 1):  
33. ponteiro = i * 2  
34. while ponteiro <sieveSize:  
35. peneira [ponteiro] = Falso  
36. ponteiro += i
```

O ponteiro variável começa no primeiro múltiplo de i depois de i , que é $i * 2$ na linha 33. Em seguida, o loop while define o índice do ponteiro na lista de peneiras como False , e a linha 36 altera o ponteiro para apontar para o próximo múltiplo de i .

Depois que o loop for na linha 32 terminar, a lista de peneiras deve agora conter True para cada índice que é um número primo. Criamos uma nova lista, que começa como uma lista vazia em primos , faz um loop em toda a lista de peneiras e acrescenta números quando peneira [i] é True ou quando i é primo:

38. # Compile a lista de primos:

39. primes = []

40. para i na faixa (sieveSize):

41. if peneira [i] == Verdadeiro:

42. primes.append (i)

A linha 44 retorna a lista de números primos:

44. primos de retorno

A função primeSieve () pode localizar todos os números primos dentro de um intervalo pequeno, e a função isPrimeTrialDiv () pode determinar rapidamente se um número pequeno é primo. Mas e quanto a um inteiro grande com centenas de dígitos?

Se passarmos um inteiro grande para isPrimeTrialDiv () , levaria vários segundos para determinar se é ou não primo. E se o número tiver centenas de dígitos, como os números primos que usaremos no programa de codificação de chave pública no [Capítulo 23](#) , levaria mais de um trilhão de anos só para descobrir se esse número é primo.

Na próxima seção, você aprenderá como determinar se um número muito grande é primo usando o teste de primalidade de Rabin-Miller.

O Algoritmo de Primalidade de Rabin-Miller

O principal benefício do algoritmo de Rabin-Miller é que ele é um teste de primalidade relativamente simples e leva apenas alguns segundos para ser executado em um computador normal. Mesmo que o código Python do algoritmo tenha apenas algumas linhas, a explicação da prova matemática de por que ele funciona seria muito longa para este livro. O algoritmo de Rabin-Miller não é um teste infalível de primalidade. Em vez disso, ele encontra números que são muito provavelmente primos, mas não são garantidos como primos. Mas a chance de um falso positivo é pequena o suficiente para que essa abordagem seja boa o suficiente para os objetivos deste livro. Para aprender mais sobre como o algoritmo de Rabin-Miller funciona, você pode ler sobre isso em https://en.wikipedia.org/wiki/Miller-Rabin_primality_test .

A função rabinMiller () implementa esse algoritmo para encontrar números primos:

46. def rabinMiller (num):

```
47. # Retorna True se num for um número primo.  
48. if num% 2 == 0 ou num <2:  
49. return False # Rabin-Miller não trabalha com números inteiros iguais.  
50. if num == 3:  
51. return Verdadeiro  
52. s = num - 1  
53. t = 0  
54. enquanto s% 2 == 0:  
55. # Manter a metade até que seja estranho (e use t  
56. # para contar quantas vezes nós dividimos pela metade):  
57. s = s // 2  
58. t += 1  
59. para ensaios no intervalo (5): # Tente falsificar a primalidade num 5 vezes.  
60. a = random.randrange (2, num - 1)  
61. v = pow (a, s, num)  
62. if v! = 1: # Este teste não se aplica se v for 1.  
63. i = 0  
64. enquanto v! = (Num - 1):  
65. se eu == t - 1:  
66. return False  
67. else:  
68. i = i + 1  
69. v = (v ** 2)% num  
70. return Verdadeiro
```

Não se preocupe sobre como esse código funciona. O conceito importante a ter em mente é que, se a função rabinMiller () retornar True , o argumento num é muito provável que seja primo. Se rabinMiller () retorna False , num é definitivamente composto.

Encontrar grandes números primos

Vamos criar outra função chamada isPrime () que irá chamar rabinMiller () . O algoritmo de Rabin-Miller nem sempre é a maneira mais eficiente de verificar se um número é primo; portanto, no início da função isPrime () , faremos algumas verificações simples como atalhos para descobrir se o número armazenado no parâmetro num é primo. Vamos armazenar uma lista de todos os primos menores que 100 na variável constante LOW_PRIMES . Podemos usar a função primeSieve () para calcular essa lista:

72. # Na maior parte do tempo podemos determinar rapidamente se num não é primo

73. # dividindo pelas primeiras dúzias de números primos. Isso é mais rápido

74. # que rabinMiller (), mas não detecta todos os compostos.

75. LOW_PRIMES = primeSieve (100)

Usaremos essa lista exatamente como fizemos em isPrimeTrialDiv () e descontaremos qualquer número menor que 2 (linhas 81 e 82):

78. def isPrime (num):

79. # Retorna True se num for um número primo. Esta função é mais rápida

80. # número primo verifique antes de chamar rabinMiller ().

81. if (num <2):

82. return False # 0, 1, e os números negativos não são primos.

Quando num não é menor que 2 , podemos usar a lista LOW_PRIMES como um atalho para testar num , também. Verificar se num é divisível por todos os primos menores que 100 não nos dirá definitivamente se o número é primo, mas pode nos ajudar a encontrar números compostos. Cerca de 90 por cento dos grandes números inteiros passados para isPrime () podem ser detectados como compostos dividindo pelos números primos menos de 100. A razão é que, se o número puder ser dividido por um número primo, como 3, você não tem que verificar se o número pode ser dividido igualmente por números compostos 6, 9, 12, 15 ou qualquer outro múltiplo de 3. Dividir o número por primos menores é muito mais rápido do que executar o algoritmo Rabin-Miller mais lento no número, então este atalho ajuda o programa a executar mais rapidamente cerca de 90% do tempo que o isPrime () é chamado.

A linha 85 faz um loop através de cada um dos números primos na lista LOW_PRIMES :

83. # Veja se algum dos números primos baixos pode dividir num:

84. para prime em LOW_PRIMES:

85. if (num == prime):

86. return Verdadeiro 87. if (num% primo == 0):

88. retornar falso

Se o inteiro em num é o mesmo que primo , então obviamente, num deve ser um número primo e a linha 86 retorna Verdadeiro . O número inteiro em num é modificado por cada número primo usando o operador mod na linha 87, e se o resultado for avaliado como 0 , sabemos que o primo divide num, então num não

é primo. Nesse caso, a linha 88 retorna False .

Esses são os três testes rápidos que vamos realizar para determinar se um número é primo. Se a execução continuar após a linha 87, a função rabinMiller () verifica a primalidade do num .

A linha 90 chama a função rabinMiller () para determinar se o número é primo; então a função rabinMiller () pega seu valor de retorno e a retorna da função isPrime () :

```
89. # Se tudo mais falhar, chame rabinMiller () para determinar se num é primo:  
90. retornar rabinMiller (num)
```

Agora que você sabe como determinar se um número é primo, usaremos esses testes de primalidade para gerar números primos. Estes serão usados pelo programa de chave pública no [Capítulo 23](#) .

Gerando grandes números primos

Usando um loop infinito, a função generateLargePrime () na linha 93 retorna um inteiro que é primo. Ele faz isso gerando um grande número aleatório, armazenando-o em num e, em seguida, passando num para isPrime () . A função isPrime () então testa se num é primo.

```
93. def generateLargePrime (tamanho da chave = 1024):  
94. # Retorna um número primo aleatório que tem tamanho de tamanho de bits:  
95. enquanto Verdadeiro:  
96.     num = random.randrange (2 ** (tamanho da chave-1), 2 ** (tamanho da  
chave))  
97.     if isPrime (num):  
98.         retornar num
```

Se num for prime, a linha 98 retornará num . Caso contrário, o loop infinito volta para a linha 96 para tentar um novo número aleatório. Esse loop continua até encontrar um número que a função isPrime () determine como primo.

O parâmetro keysize da função generateLargePrime () tem um valor padrão de 1024 . Quanto maior o tamanho da chave , mais chaves possíveis existem e mais difícil a cifra é a força bruta. Os tamanhos de chave pública são geralmente calculados em termos de números chamados *bits* , sobre os quais você aprenderá mais nos [Capítulos 23](#) e [24](#) . Por enquanto, apenas saiba que um número de 1024 bits é muito grande: são cerca de 300 dígitos!

Resumo

Números primos têm propriedades fascinantes em matemática. Como você aprenderá no [Capítulo 23](#), eles também são a espinha dorsal das cifras usadas no software de criptografia profissional. A definição de um número primo é bastante simples: é um número que tem apenas 1 e ele próprio como fatores. Mas determinar quais números são primos requer algum código inteligente.

Neste capítulo, escrevemos a função `isPrimeTrialDiv()` para determinar se um número é primo modificando um número por todos os números entre 2 e a raiz quadrada do número. Este é o algoritmo da divisão experimental. Um número primo nunca deve ter um resto de 0 quando modificado por qualquer número que não seja seus fatores, 1 e ele próprio. Então, sabemos que um número que tem 0 restante não é primo.

Você aprendeu que a peneira de Eratóstenes pode encontrar rapidamente todos os números primos em uma série de números, embora use muita memória para encontrar primos grandes.

Como a peneira de Eratóstenes e o algoritmo de divisão de testes em `primeNum.py` não são rápidos o suficiente para encontrar grandes números primos, precisávamos de outro algoritmo para a codificação de chave pública, que usa números primos extremamente grandes com centenas de dígitos. Como solução alternativa, você aprendeu a usar o algoritmo de Rabin-Miller, que usa um raciocínio matemático complexo para determinar se um número muito grande é primo.

No [Capítulo 23](#), você usará o módulo `primeNum.py` para escrever o programa de criptografia de chave pública. Por fim, você criará uma cifra que é mais fácil de usar do que a cifra de uma só vez, mas não pode ser hackeada pelas técnicas simples de hackers apresentadas neste livro!

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Quantos números primos estão lá?
2. O que são inteiros que não são primos chamados?
3. Quais são os dois algoritmos para encontrar números primos?

23

GERANDO CHAVES PARA A CIPRA CHAVE PÚBLICA

“Use criptografia deliberadamente comprometida, que tenha uma porta dos fundos para a qual apenas os ‘mocinhos’ devam ter as chaves, e você efetivamente não tem segurança. Você pode também skywrite como criptografar com criptografia pré-quebrada e sabotada ”.

—Cory Doctorow, autor de ficção científica, 2015



Todas as cifras que você aprendeu neste livro até agora têm um recurso em comum: a chave de criptografia é a mesma que a chave de descriptografia. Isso leva a um problema complicado: como você compartilha mensagens criptografadas com alguém com quem nunca falou antes? Qualquer bisbilhoteiro poderia interceptar uma chave de criptografia que você envia com a mesma facilidade com que poderia interceptar as mensagens criptografadas.

Neste capítulo, você aprenderá sobre criptografia de chave pública, que permite que estranhos compartilhem mensagens criptografadas usando uma chave pública e uma chave privada. Você aprenderá sobre a codificação da chave pública, que neste livro é baseada na cifra RSA. Como a cifra RSA é complexa e envolve várias etapas, você vai escrever dois programas. Neste capítulo, você escreverá o programa de geração de chave pública para gerar suas chaves públicas e privadas. Então, em [Capítulo 24](#), você vai escrever um segundo programa para criptografar e descriptografar mensagens usando a cifra de chave pública e aplicando as chaves geradas aqui. Antes de mergulharmos no programa, vamos explorar como a criptografia de chave pública funciona.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- criptografia de chave pública
- Autenticação

- Assinaturas digitais
- ataques MITM
- Gerando chaves públicas e privadas
- Sistemas criptográficos híbridos

Criptografia de chave pública

Imagine que alguém do outro lado do mundo queira se comunicar com você. Vocês dois sabem que as agências de espionagem estão monitorando todos os e-mails, cartas, textos e telefonemas. Para enviar mensagens criptografadas para essa pessoa, ambos devem concordar com uma chave secreta a ser usada. Mas se um de vocês enviar a chave secreta para o outro, a agência de espionagem interceptará essa chave e depois descriptografará quaisquer mensagens futuras criptografadas usando essa chave. Secretamente reunião pessoalmente para trocar a chave é impossível. Você pode tentar criptografar a chave, mas isso requer o envio *dessa chave secreta* para essa mensagem para a outra pessoa, que também será interceptada.

A *criptografia de chave pública* resolve esse problema de criptografia usando duas chaves, uma para criptografia e outra para descriptografia, e é um exemplo de uma *cifra assimétrica*. As cifras que usam a mesma chave para criptografia e descriptografia, como muitas das cifras anteriores deste livro, são *cifras simétricas*.

É importante saber que *uma mensagem criptografada usando a chave de criptografia (chave pública) só pode ser descriptografada usando a chave de descriptografia (chave privada)*. Portanto, mesmo que alguém obtenha a chave de criptografia, ela não poderá ler a mensagem original porque a chave de criptografia não pode descriptografar a mensagem.

A chave de criptografia é chamada *de chave pública* porque pode ser compartilhada com o mundo inteiro. Por outro lado, a *chave privada*, ou a chave de decodificação, deve ser mantida em segredo.

Por exemplo, se Alice quiser enviar uma mensagem para Bob, Alice descobrirá a chave pública de Bob ou Bob poderá dar a ela. Então Alice criptografa sua mensagem para Bob usando a chave pública de Bob. Como a chave pública não pode descriptografar mensagens, não importa que todos tenham acesso à chave pública de Bob.

Quando Bob recebe a mensagem criptografada de Alice, ele usa sua chave privada para descriptografá-la. Apenas Bob tem a chave privada que pode descriptografar mensagens criptografadas usando sua chave pública. Se Bob quiser responder a Alice, ele encontrará a chave pública dela e a usará para criptografar sua resposta. Porque só Alice sabe sua chave privada, Alice é a única pessoa que pode descriptografar a resposta criptografada de Bob. Mesmo que Alice e Bob estejam em lados opostos do mundo, eles podem trocar mensagens sem medo de interceptação.

A particular codificação de chave pública que implementaremos neste capítulo é baseada na cifra RSA, que foi inventada em 1977 e é nomeada usando as iniciais dos sobrenomes de seus inventores: Ron Rivest, Adi Shamir e Leonard Adleman.

A cifra RSA usa grandes números primos com centenas de dígitos em seu algoritmo. É a razão pela qual discutimos a matemática dos números primos no [Capítulo 22](#). O algoritmo de chave pública cria dois números primos aleatórios e, em seguida, usa matemática complicada (incluindo encontrar um inverso modular, que você aprendeu como fazer no [Capítulo 13](#)) para criar as chaves públicas e privadas.

OS PERIGOS DE USAR O TEXTBOOK RSA

Embora não escrevamos um programa neste livro para hackear o programa de codificação de chave pública, tenha em mente que o programa `publicKeyCipher.py` que você vai escrever no [Capítulo 24](#) não é seguro. Acertar a criptografia é muito difícil, e muita experiência é necessária para saber se uma cifra (e um programa que a implementa) é realmente segura.

O programa baseado em RSA neste capítulo é conhecido como *livro didático RSA*, porque, embora ele tecnicamente implemente o algoritmo RSA corretamente usando grandes números primos, ele é vulnerável a ataques de hackers. Por exemplo, usar funções pseudo-aleatórias em vez de gerar números verdadeiramente aleatórios torna a cifra vulnerável e, como você aprendeu no [Capítulo 22](#), o teste de primalidade de Rabin-Miller não tem a garantia de estar sempre correto.

Então, embora você possa não ser capaz de hackear o texto cifrado criado por `publicKeyCipher.py`, outros podem conseguir. Parafraseando o criptógrafo altamente talentoso Bruce Schneier, qualquer um pode criar um algoritmo criptográfico que não pode se quebrar. O que é desafiador é criar um algoritmo

que ninguém mais pode quebrar, mesmo depois de anos de análise. O programa deste livro é apenas um exemplo divertido que pretende ensinar os fundamentos da codificação RSA. Mas não se esqueça de usar um software de criptografia profissional para proteger seus arquivos. Você pode encontrar uma lista de software de criptografia (geralmente gratuito) em <https://www.nostarch.com/crackingcodes/>.

O problema com autenticação

Por mais engenhoso que possa parecer a cifra de chave pública, há um pequeno problema. Por exemplo, imagine que você recebeu este e-mail: “Olá. Eu sou Emmanuel Goldstein, líder da resistência. Eu gostaria de me comunicar secretamente com você sobre assuntos importantes. Anexada é minha chave pública.

Usando essa chave pública, você pode ter certeza de que as mensagens enviadas não podem ser lidas por ninguém além do remetente da chave pública. Mas como você sabe que o remetente é na verdade Emmanuel Goldstein? Você não sabe se está enviando mensagens criptografadas para Emmanuel Goldstein ou para uma agência de espionagem fingindo ser Emmanuel Goldstein para atraí-lo para uma armadilha!

Embora os códigos de chave pública e, de fato, todas as cifras neste livro possam fornecer *confidencialidade* (mantendo a mensagem em segredo), nem sempre fornecem *autenticação* (prova de que com quem você está se comunicando é quem ele diz que é).

Normalmente, isso não é um problema com cifras simétricas, porque quando você troca chaves com uma pessoa, você pode ver quem é essa pessoa. No entanto, quando você usa a criptografia de chave pública, não precisa ver uma pessoa para obter sua chave pública e começar a enviar mensagens criptografadas. A autenticação é fundamental para manter em mente quando você está usando criptografia de chave pública.

Um campo inteiro chamado *PKI* (*public key infrastructure, infraestrutura de chave pública*) gerencia a autenticação para que você possa corresponder chaves públicas a pessoas com algum nível de segurança; no entanto, esse tópico está além do escopo deste livro.

Assinaturas digitais

As *assinaturas digitais* permitem assinar documentos eletronicamente usando

criptografia. Para entender por que as assinaturas digitais são necessárias, vamos analisar o seguinte exemplo de email de Alice para Bob:

Caro Bob

Eu prometo comprar seu velho laptop quebrado por um milhão de dólares.

Atenciosamente,

Alice

Esta é uma ótima notícia para Bob, porque ele quer se livrar de seu laptop inútil por qualquer preço. Mas e se Alice depois alegar que ela não fez essa promessa e que o e-mail que Bob recebeu é uma farsa que ela não enviou. Afinal, Bob poderia ter criado esse e-mail com facilidade.

Se eles se encontrassem pessoalmente, Alice e Bob poderiam simplesmente assinar um contrato concordando com a venda. Uma assinatura manuscrita não é tão fácil de falsificar e fornece alguma prova de que Alice fez essa promessa. Mas, mesmo que Alice tenha assinado tal contrato, tirou uma foto dele com sua câmera digital e enviou a ele o arquivo de imagem, ainda é concebível que a imagem possa ter sido alterada.

A cifra RSA (como outras cifras de chave pública) não apenas criptografa mensagens, mas também permite *assinar digitalmente* um arquivo ou string. Por exemplo, Alice pode criptografar uma mensagem usando sua chave privada, produzindo um texto cifrado que somente a chave pública de Alice pode descriptografar. Este texto cifrado se torna a assinatura digital do arquivo. Na verdade, não é um segredo porque todos no mundo têm acesso à chave pública de Alice para descriptografá-la. *Mas, ao criptografar uma mensagem usando sua chave privada, Alice pode assinar digitalmente a mensagem de uma forma que não pode ser forjada.* Como apenas Alice tem acesso a sua chave privada, apenas Alice poderia ter produzido esse texto cifrado, e ela não poderia dizer que Bob forjou ou alterou!

A garantia de que alguém que criou uma mensagem não poderá negar a criação dessa mensagem posteriormente como *não-repúdio* .

As pessoas usam assinaturas digitais para muitas atividades importantes, incluindo criptomoeda, autenticação de chaves públicas e navegação na web anônima. Para saber mais, acesse <https://www.nostarch.com/crackingcodes/>. Continue lendo para entender por que a autenticação é tão importante quanto uma criptografia segura.

Cuidado com o ataque MITM

Ainda mais insidioso do que alguém que hackeia nossas mensagens criptografadas é um *ataque do tipo intermediário* ou *intermediário (MITM, man-in-the-middle ou machine-in-the-middle)*. Nesse tipo de ataque, alguém intercepta sua mensagem e a transmite sem o seu conhecimento. Por exemplo, digamos que Emmanuel Goldstein realmente queira se comunicar com você e enviar uma mensagem não criptografada com sua chave pública, mas os roteadores da agência de espionagem a interceptam. A agência pode substituir a chave pública Emmanuel anexada ao email com sua própria chave pública e, em seguida, enviar a mensagem para você. Você não teria como saber se a chave que você recebeu é a chave de Emmanuel ou a chave da agência de espionagem!

Então, quando você criptografa uma resposta para Emmanuel, você está realmente criptografando usando a chave pública da agência de espionagem, não a chave pública de Emmanuel. A agência de espionagem seria capaz de interceptar essa mensagem, descriptografá-la, lê-la e depois reencryptá-la usando a chave pública real de Emmanuel antes de enviar sua mensagem para Emmanuel. A agência poderia fazer o mesmo com qualquer resposta que Emmanuel lhe enviasse. [A Figura 23-1](#) mostra como o ataque MITM funciona.

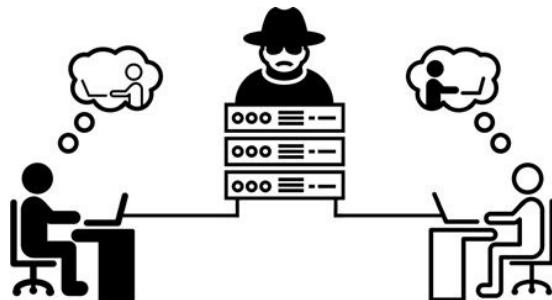


Figura 23-1: Um ataque MITM

Para você e Emmanuel, parece que você está se comunicando em segredo. Mas a agência de espionagem está lendo todas as suas mensagens porque você e Emmanuel estão criptografando suas mensagens usando chaves públicas geradas pela agência de espionagem! Novamente, esse problema existe porque uma cifra de chave pública fornece apenas confidencialidade, não autenticação. Uma discussão aprofundada sobre autenticação e infraestrutura de chave pública está além do escopo deste livro. Mas agora que você sabe como a criptografia de chave pública fornece confidencialidade, vamos ver como gerar chaves para a codificação de chave pública.

Etapas para gerar chaves públicas e privadas

Cada chave no esquema de chave pública é composta de dois números. A chave pública será os dois números n e e . A chave privada será os dois números n e d .

As três etapas para criar esses números são as seguintes:

1. Crie dois números primos aleatórios, distintos e muito grandes: p e q . Multiplique esses dois números para obter um número chamado n .
2. Crie um número aleatório, chamado e , que é relativamente primo para $(p - 1) \times (q - 1)$.
3. Calcule o inverso modular de e , que vamos chamar d .

Observe que n é usado em ambas as chaves. O número d deve ser mantido em segredo porque pode descriptografar mensagens. Agora você está pronto para escrever um programa que gera essas chaves.

Código Fonte do Programa de Geração de Chave Pública

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Certifique-se de que os módulos *primeNum.py* e *cryptomath.py* estejam na mesma pasta que o arquivo de programa. Digite o seguinte código no editor de arquivos e salve-o como *makePublicPrivateKeys.py*.

tornar público

PrivateKeys.py

```
1. # gerador de chave pública
2. # https://www.nostarch.com/crackingcodes/ (Licenciado pelo BSD)
3
4. importação aleatória, sys, os, primeNum, cryptomath
5
6
7. def main():
8. # Crie um par de chaves público / privado com chaves de 1024 bits:
9. print ('Criando arquivos de chave ...')
10. makeKeyFiles ('al_sweigart', 1024)
11. print ('Arquivos de chave feitos.')
12
13. def generateKey (keySize):
14. # Cria chaves públicas / privadas keySize bits em tamanho.
```

```
15. p = 0
16. q = 0
17. # Passo 1: Crie dois números primos, pe q. Calcule n = p * q:
18. print ('Gerando p prime ...')
19. enquanto p == q:
20.     p = primeNum.generateLargePrime (keySize)
21.     q = primeNum.generateLargePrime (keySize)
22.     n = p * q
23
24. # Passo 2: Crie um número e que seja relativamente primo para (p-1) * (q-1):
25. print ('Gerando e que é relativamente primo para (p-1) * (q-1) ...')
26. enquanto True:
27.     # Continue tentando números aleatórios para e até que um seja válido:
28.     e = random.randrange (2 ** (keySize - 1), 2 ** (keySize))
29.     se cryptomath.gcd (e, (p - 1) * (q - 1)) == 1:
30.         quebrar
31
32. # Etapa 3: Calcule d, o mod inverso de e:
33. print ('Calculando d que é mod inverso de e ...')
34. d = cryptomath.findModInverse (e, (p - 1) * (q - 1))
35
36. publicKey = (n, e)
37. privateKey = (n, d)
38
39. print ('Chave pública:', publicKey)
40. print ('Chave privada:', chave privada)
41.
42. return (publicKey, privateKey)
43
44
45. def makeKeyFiles (nome, tamanho da chave):
46.     # Cria dois arquivos 'x_pubkey.txt' e 'x_privkey.txt' (onde x
47.     # é o valor no nome) com os inteiros n, e ed, escritos em
48.     # eles, delimitados por uma vírgula.
49.
50. # Nossa verificação de segurança nos impedirá de sobrescrever nossos
arquivos de chaves antigos:
```

```
51. if os.path.exists ('% s_pubkey.txt' % (name)) ou
os.path.exists ('% s_privkey.txt' % (name)):
52.     sys.exit ('AVISO: O arquivo % s_pubkey.txt ou % s_privkey.txt
já existe! Use um nome diferente ou exclua esses arquivos e
execute novamente este programa. % (nome nome))
53
54. publicKey, privateKey = generateKey (keySize)
55
56. print ()
57. print ('A chave pública é um número de dígitos % s.' %
(len (str (publicKey [0])), len (str (publicKey [1]))))
58. print ('Gravando chave pública no arquivo % s_pubkey.txt ... % (name)')
59. fo = open ('% s_pubkey.txt' % (name), 'w')
60. fo.write ('% s,% s,% s' % (keySize, publicKey [0], publicKey [1]))
61. fo.close ()
62
63. print ()
64. print ('A chave privada é um número de % s e um número de % s.' %
(len (str (publicKey [0])), len (str (publicKey [1]))))
65. print ('Escrevendo chave privada para o arquivo % s_privkey.txt ... % (name)')
66. fo = open ('% s_privkey.txt' % (name), 'w')
67. fo.write ('% s,% s,% s' % (keySize, chave privada [0], chave privada [1]))
68. fo.close ()
69
70
71. # Se makePublicPrivateKeys.py for executado (em vez de importado como
um módulo),
72. # chama a função main():
73. if __name__ == '__main__':
74.     main ()
```

Execução de amostra do programa de geração de chave pública

Quando executamos o programa *makePublicPrivateKeys.py*, a saída deve ser semelhante à seguinte (é claro, os números das chaves serão diferentes porque são gerados aleatoriamente):

Fazendo arquivos chave ...

Gerando p prime ...

Gerando q prime ...

Gerando e que é relativamente primo para $(p-1) * (q-1)$...

Calculando d que é mod inverso de e ...

Chave pública:

(210902406316700502401968491406579417405090396754616926135810621
21611619133808656784074598753554688979280723862705107204438273246
85839374968506241167761472418211520269463228768694043944839222024
24247892081318269900084735267117442965485638667684542514049519608
49897523048895590808649185211634877784953627068508544697095291564
2042218037444940658810103314864683053174496070278847877031572995
26531132766377616771007701834003666830661266575941720784582347990
06812521100232929833871861585954209372109725826359561748245019920
04468791300114315056117093,
17460230769175161021731845459236833553832403910869
12905495420037367858093524760662226576438823575217665473780584902
30868551366950991745119582261139809895130667660095888918956459958
39369327768340435481157568160599065914531707412708455723353750410
0216777273298110097435989)

Chave privada:

(21090240631670050240196849140657941740509039675461692613581062
12161161913380865678407459875355468897928072386270510720443827324
48583937496850624116776147241821152026946322876869404394483922202
42424789208131826990008473526711744296548563866768454251404951960
54989752304889559080864918521163487778495362706850854469709529156
2042218037444940658810103314864683053174496070278847877031572995
26531132766377616771007701834003666830661266575941720784582347990
06812521100232929833871861585954209372109725826359561748245019920
04468791300114315056117093,
47676735798137710412166884916983765043173120289416
90434129597155228687099187466609993337100807594854900855122476069
96816899540499393450839901428305371067676083594890231288863993840
36077306236416266427614496565255854533116668173598098138449334931
68372702963348445191139635826000818122373486213256488077192893119
25681884603640028673273135292831170178614206817165802812291528319
55726168047084560706359601833919317974375031636011432177696164717
26990539739057474642785416933878499897014777481407371328053001838
845219087249544663398589)

A chave pública é um número 617 e um número de 309 dígitos.

Escrevendo chave pública para o arquivo al_sweigart_pubkey.txt ...

A chave privada é um número 617 e um número de 309 dígitos.

Escrevendo a chave privada para o arquivo al_sweigart_privkey.txt ...

Como as duas chaves são muito grandes, elas são gravadas em seus próprios arquivos, *al_sweigart_pubkey.txt* e *al_sweigart_privkey.txt*. Usaremos ambos os arquivos no programa de criptografia de chave pública no [Capítulo 24](#) para criptografar e descriptografar mensagens. Esses nomes de arquivos vêm da string '*al_sweigart*' , que é passada para a função `makeKeyFiles()` na linha 10 do programa. Você pode especificar diferentes nomes de arquivos passando uma string diferente.

Se rodarmos *makePublicPrivateKeys.py* novamente e passarmos a mesma string para `makeKeyFiles()` , a saída do programa deve ficar assim:

Fazendo arquivos chave ...

AVISO: O arquivo *al_sweigart_pubkey.txt* ou *al_sweigart_privkey.txt* já existe! Use um nome diferente ou exclua esses arquivos e execute novamente este programa.

Esse aviso é fornecido para que não sobrescrevamos acidentalmente nossos arquivos de chaves, o que impossibilitaria a recuperação de arquivos criptografados. Certifique-se de manter esses arquivos importantes seguros!

Criando a função main ()

Quando executamos *makePublicPrivateKeys.py* , a função `main()` é chamada, o que cria os arquivos de chave pública e privada usando a função `makeKeyFiles()` que definiremos em breve.

```
7. def main():
8. # Crie um par de chaves público / privado com chaves de 1024 bits:
9. print('Criando arquivos de chave ...')
10. makeKeyFiles('al_sweigart', 1024)
11. print('Arquivos de chave feitos.')
```

Como o computador pode demorar um pouco para gerar as chaves, imprimimos uma mensagem na linha 9 antes da chamada `makeKeyFiles()` para informar ao usuário o que o programa está fazendo.

A chamada `makeKeyFiles()` na linha 10 passa a string '*al_sweigart*' e o inteiro

1024 , que gera chaves que são 1024 bits e as armazena nos arquivos *al_sweigart_pubkey.txt* e *al_sweigart_privkey.txt* . Quanto maior o tamanho da chave, mais chaves possíveis existem e mais forte a segurança da cifra. No entanto, tamanhos de chave grandes significam que leva mais tempo para criptografar ou descriptografar mensagens. Eu escolhi o tamanho de 1024 bits como um equilíbrio entre velocidade e segurança para os exemplos deste livro; mas, na realidade, um tamanho de chave de 2048 bits ou mesmo 3072 bits é necessário para a criptografia de chave pública segura.

Gerando chaves com a função generateKey ()

O primeiro passo na criação de chaves está chegando com os dois números primos aleatórios p e q . Esses números devem ser dois números primos grandes e distintos.

13. def generateKey (keySize):
14. # Cria chaves públicas / privadas keySize bits em tamanho.
15. p = 0
16. q = 0
17. # Passo 1: Crie dois números primos, pe q. Calcule $n = p * q$:
18. print ('Gerando p prime ...')
19. enquanto $p == q$:
20. $p = primeNum.generateLargePrime (keySize)$
21. $q = primeNum.generateLargePrime (keySize)$
22. $n = p * q$

A função *generateLargePrime ()* que você escreveu no programa *primeNum.py* no [Capítulo 22](#) retorna dois números primos como valores inteiros nas linhas 20 e 21, que armazenamos nas variáveis p e q . Isso é feito dentro de um loop while que continua a fazer loop enquanto p e q forem iguais. Se *generateLargePrime ()* retornar o mesmo inteiro para peq , o programa tentará novamente encontrar primos únicos para peq . O valor em *keySize* determina os tamanhos de peq . Na linha 22, multiplicamos p e q e armazenamos o produto em n .

Em seguida, na etapa 2, calculamos a outra parte da chave pública: e .

Calculando um valor e

O valor para e é calculado encontrando um número que é relativamente primo para $(p - 1) \times (q - 1)$. Não entraremos em detalhes sobre por que e é calculado dessa maneira, mas queremos que e seja relativamente primo para $(p - 1) \times (q - 1)$.

1) porque isso garante que e sempre resultará em um texto cifrado único.

Usando um loop while infinito, a linha 26 calcula um número e , que é relativamente primo para o produto de $p - 1$ e $q - 1$.

24. # Passo 2: Crie um número e que seja relativamente primo para $(p-1) * (q-1)$:

25. print ('Gerando e que é relativamente primo para $(p-1) * (q-1) ...'$)

26. enquanto True:

27. # Continue tentando números aleatórios para e até que um seja válido:

28. $e = \text{random.randrange}(2^{**(\text{keySize} - 1)}, 2^{**(\text{keySize})})$

29. se cryptomath.gcd ($e, (p - 1) * (q - 1)$) == 1:

30. quebrar

A chamada random.randrange () na linha 28 retorna um inteiro aleatório e o armazena na variável e . Então a linha 29 testa se e é relativamente primo para $(p - 1) * (q - 1)$ usando a função gcd () do módulo cryptomath . Se e é relativamente primo, a instrução de quebra na linha 30 irrompe do loop infinito. Caso contrário, a execução do programa volta para a linha 26 e continua tentando diferentes números aleatórios até encontrar um que seja relativamente primo para $(p - 1) * (q - 1)$.

Em seguida, calculamos a chave privada.

Cálculo do valor do anúncio

O terceiro passo é encontrar a outra metade da chave privada usada para descriptografar, que é d . O d é apenas o inverso modular de e , e já temos a função findModInverse () no módulo cryptomath , que usamos no [Capítulo 13](#) , para calcular isso.

A linha 34 chama o método findModInverse () e armazena o resultado na variável d .

32. # Etapa 3: Calcule d , o mod inverso de e :

33. print ('Calculando d que é mod inverso de e ...')

34. $d = \text{cryptomath.findModInverse}(e, (p - 1) * (q - 1))$

Nós agora definimos todos os números que precisamos para gerar as chaves públicas e privadas.

Retornando as Chaves

Lembre-se de que, na criptografia de chave pública, as chaves pública e privada consistem em dois números cada. Os inteiros armazenados em n e e representam

a chave pública, e os inteiros armazenados em n e d representam a chave privada. As linhas 36 e 37 armazenam esses pares inteiros como valores de tupla em publicKey e privateKey .

```
36. publicKey = (n, e)
37. privateKey = (n, d)
```

As linhas restantes na função generateKey () imprimem as teclas na tela usando as chamadas print () nas linhas 39 e 40.

```
39. print ('Chave pública:', publicKey)
40. print ('Chave privada:', chave privada)
41.
42. return (publicKey, privateKey)
```

Então, quando generateKey () é chamado, a linha 42 retorna uma tupla contendo publicKey e privateKey .

Depois que geramos as chaves públicas e privadas, também queremos armazená-las em arquivos para que nosso programa de criptografia de chave pública possa usá-las posteriormente para criptografar e descriptografar. Além disso, armazenar as chaves em arquivos é muito útil, pois os dois inteiros que compõem cada chave têm centenas de dígitos, o que dificulta a memorização ou a anotação conveniente.

Criando arquivos de chave com a função makeKeyFiles ()

A função makeKeyFiles () armazena as chaves nos arquivos, usando um nome de arquivo e um tamanho de chave como parâmetros.

```
45. def makeKeyFiles (nome, tamanho da chave):
46. # Cria dois arquivos 'x_pubkey.txt' e 'x_privkey.txt' (onde x
47. # é o valor no nome) com os inteiros n, e ed, escritos em
48. # eles, delimitados por uma vírgula.
```

A função cria dois arquivos no formato $<name>_pubkey.txt$ e $<name>_privkey.txt$ que armazenam as chaves públicas e privadas, respectivamente. O parâmetro name passado para a função determina a parte $<name>$ dos arquivos.

Para evitar a exclusão acidental de arquivos-chave, executando o programa novamente, a linha 51 verifica se os arquivos de chave pública ou privada com o nome fornecido já existem. Se o fizerem, o programa sai com uma mensagem de aviso.

50. # Nossa verificação de segurança nos impedirá de sobrescrever nossos arquivos de chaves antigos:

51. if os.path.exists ('% s_pubkey.txt' % (name)) ou
os.path.exists ('% s_privkey.txt' % (name)):

52. sys.exit ('AVISO: O arquivo% s_pubkey.txt ou% s_privkey.txt já existe! Use um nome diferente ou exclua esses arquivos e execute novamente este programa. ' % (nome nome))

Após essa verificação, a linha 54 chama generateKey () e passa o keySize para criar as chaves públicas e privadas do tamanho especificado.

54. publicKey, privateKey = generateKey (keySize)

A função generateKey () retorna uma tupla de duas tuplas que ela atribui às variáveis publicKey e privateKey usando várias atribuições. A primeira tupla tem dois inteiros para a chave pública, e a segunda tupla tem dois inteiros para a chave privada.

Agora que terminamos a configuração para criar os arquivos de chave, podemos criar os arquivos de chave reais. Vamos armazenar os dois números que compõem cada chave em arquivos de texto.

A linha 56 imprime uma linha vazia e, em seguida, a linha 57 imprime informações sobre a chave pública do usuário.

56. print ()

57. print ('A chave pública é um número de dígitos% se% s.'%
(len (str (publicKey [0])), len (str (publicKey [1]))))

58. print ('Gravando chave pública no arquivo% s_pubkey.txt ...' % (name))

A linha 57 indica quantos dígitos estão no inteiro em publicKey [0] e publicKey [1] convertendo esses valores em strings usando a função str () e, em seguida, localizando o comprimento da string usando a função len () . Em seguida, a linha 58 relata que a chave pública está sendo gravada no arquivo.

O conteúdo do arquivo-chave inclui o tamanho da chave, uma vírgula, o inteiro n , outra vírgula e o inteiro e (ou d). Por exemplo, o conteúdo do arquivo deve ter esta aparência: tamanho da chave inteiro , n inteiro , e ou d inteiro , como no seguinte arquivo *al_sweigart_pubkey.txt* :

```
1024, 141189561571082936553468080511334338940916460395383120069233997353624936052632037024975858
93776717003286326229134304078204210728995962809448233282087726441833718356477474042405336332872
07520733469653530410225698180493180588850258751531087325796653837774040742213790777243761337634
29403748158391548973157601450752430714012338584282327252143912951516980441475584541848071057874
19519119343953276836694146614061330872356766933442169358208953710231872729486994792595105820069
35116306633036219116343447342195108296634686096567178928088702044098327996749848014723273440168
2910892741619433374703999689201536556462802829353073, 100718103971294791099836725874012546370680
92601218580576540105227626258238571515977536644616265994855975364767266381161481376979016411453
12931752030296204272437195994689585517456366655589415261645234299654897035299400304656468484497
15020479155556561228677211251598560502855023412904336022230634725973056990069
```

As linhas 59 a 61 abrem o arquivo `<name>_pubkey.txt` no modo de gravação, gravam as chaves no arquivo e o fecham.

```
59. fo = open ('% s_pubkey.txt'% (name), 'w')
60. fo.write ('% s,% s,% s'%(keySize, publicKey [0], publicKey [1]))
61. fo.close ()
```

As linhas 63 a 68 fazem a mesma coisa que as linhas 56 a 61, exceto que elas gravam a chave privada no arquivo `<nome>_privkey.txt`.

```
63. print ()
64. print ('A chave privada é um número de% s e um número de% s.'%
(len (str (publicKey [0])), len (str (publicKey [1]))))
65. print ('Escrevendo chave privada para o arquivo% s_privkey.txt ...'% (name))
66. fo = open ('% s_privkey.txt'% (name), 'w')
67. fo.write ('% s,% s,% s'%(keySize, chave privada [0], chave privada [1]))
68. fo.close ()
```

Certifique-se de que ninguém hackeia seu computador e copia esses arquivos-chave. Se eles obtiverem seu arquivo de chave privada, eles poderão descriptografar todas as suas mensagens!

Chamando a função main ()

No final do programa, as linhas 73 e 74 chamam a função `main ()` se `makePublicPrivateKeys.py` estiver sendo executado como um programa em vez de ser importado como um módulo por outro programa.

```
71. # Se makePublicPrivateKeys.py for executado (em vez de importado como
um módulo),
72. # chama a função main () :
73. if __name__ == '__main__':
74. main ()
```

Agora você tem um programa que pode gerar chaves públicas e privadas. Embora a codificação de chave pública seja útil para iniciar comunicações entre duas pessoas que não têm uma maneira segura de trocar chaves, ela geralmente não é usada para toda a comunicação criptografada devido às desvantagens da criptografia de chave pública. Muitas vezes, as cifras de chave pública são usadas em criptosistemas híbridos.

Sistemas Criptográficos Híbridos

O RSA e a criptografia de chave pública levam muito tempo para serem computados. Isso é especialmente verdadeiro para servidores que precisam fazer milhares de conexões criptografadas com outros computadores por segundo. Como solução alternativa, as pessoas podem usar a criptografia de chave pública para criptografar e distribuir a chave para uma *codificação de chave simétrica* muito mais rápida, que é qualquer tipo de criptografia em que as chaves de criptografia e descriptografia são as mesmas. Depois de enviar com segurança a chave da cifra simétrica para o receptor usando uma mensagem criptografada pela chave pública, o remetente pode usar a cifra simétrica para mensagens futuras. O uso de uma codificação de chave simétrica e uma codificação de chave assimétrica para comunicação é chamado de *criptosistema híbrido*. Você pode ler mais sobre criptosistemas híbridos em https://en.wikipedia.org/wiki/Hybrid_cryptosystem.

Resumo

Neste capítulo, você aprendeu como funciona a criptografia de chave pública e como escrever um programa que gera as chaves públicas e privadas. No [Capítulo 24](#), você usará essas chaves para executar a criptografia e a descriptografia usando a codificação de chave pública.

QUESTÕES PRÁTICAS

As respostas para as questões práticas podem ser encontradas no site do livro em <https://www.nostarch.com/crackingcodes/>.

1. Qual é a diferença entre uma cifra simétrica e uma cifra assimétrica?
2. Alice gera uma chave pública e uma chave privada. Infelizmente, mais tarde ela perde sua chave privada.
 1. Outras pessoas poderão enviar mensagens criptografadas?
 2. Ela poderá descriptografar mensagens enviadas anteriormente

para ela?

3. Ela será capaz de assinar documentos digitalmente?
4. Outras pessoas poderão verificar seus documentos previamente assinados?
3. O que são autenticação e confidencialidade? Como eles são diferentes?
4. O que é não-repúdio?

24

PROGRAMANDO A CIPRA CHAVE PÚBLICA

"Um colega me disse uma vez que o mundo estava cheio de sistemas de segurança ruins projetados por pessoas que liam a Criptografia Aplicada".
—Bruce Schneier, autor de Criptografia Aplicada



No [Capítulo 23](#), você aprendeu como funciona a criptografia de chave pública e como gerar arquivos de chave pública e privada usando o programa de geração de chave pública. Agora você está pronto para enviar seu arquivo de chave pública para outras pessoas (ou postá-lo on-line) para que elas possam usar-o para criptografar suas mensagens antes de enviá-las para você. Neste capítulo, você escreverá o programa de codificação de chave pública que criptografa e descriptografa mensagens usando as chaves públicas e privadas geradas.

AVISO

A implementação da cifra de chave pública neste livro é baseada na cifra RSA. No entanto, muitos detalhes adicionais que não serão abordados aqui tornam essa codificação imprópria para uso no mundo real. Embora a cifra da chave pública seja impermeável às técnicas da criptoanálise neste livro, ela é vulnerável às técnicas avançadas que os criptógrafos profissionais usam

atualmente.

TÓPICOS ABORDADOS NESTE CAPÍTULO

- A função pow ()
- As funções min () e max ()
- O método da lista insert ()

Como funciona a criptografia de chave pública

Semelhante às cifras anteriores que programamos, a cifra de chave pública converte caracteres em números e, em seguida, realiza cálculos nesses números para criptografá-los. O que diferencia a cifra de chave pública de outras cifras que você aprendeu é que ela converte vários caracteres em um inteiro chamado *bloco* e criptografa um bloco de cada vez.

A razão pela qual a codificação de chave pública precisa funcionar em um bloco que representa vários caracteres é que, se usássemos o algoritmo de criptografia de chave pública em um único caractere, os mesmos caracteres de texto simples sempre criptografariam para os mesmos caracteres de texto cifrado. Portanto, a cifra da chave pública não seria diferente de uma simples cifra de substituição com matemática chique.

Criando Blocos

Na criptografia, um *bloco* é um inteiro grande que representa um número fixo de caracteres de texto. O programa *publicKeyCipher.py* que criaremos neste capítulo converte o valor da string de mensagem em blocos e cada bloco é um inteiro que representa 169 caracteres de texto. O tamanho máximo do bloco depende do tamanho do conjunto de símbolos e do tamanho da chave. Em nosso programa, o conjunto de símbolos, que contém os únicos caracteres que uma mensagem pode ter, será a cadeia de 66 caracteres

'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456

.

O *tamanho do bloco de tamanho de conjunto de símbolo > tamanho da chave da equação 2* deve ser verdadeiro. Por exemplo, quando você usa uma chave de 1024 bits e um conjunto de símbolos de 66 caracteres, o tamanho máximo do bloco é um número inteiro de até 169 caracteres, pois 2^{1024} é maior que 66^{169} . No entanto, 2^{1024} não é maior que 66^{170} . Se você usar blocos muito grandes, a matemática da chave pública não funcionará e você não poderá

descriptografar o texto cifrado que o programa produziu.

Vamos explorar como converter uma string de mensagem em um bloco inteiro grande.

Convertendo uma String em um Bloco

Como nos nossos programas de codificação anteriores, podemos usar o índice da string do conjunto de símbolos para converter caracteres de texto em números inteiros e vice-versa. Vamos armazenar o conjunto de símbolos em uma constante denominada SYMBOLS, onde o caractere no índice 0 é 'A' , o caractere no índice 1 é 'B' e assim por diante. Digite o seguinte no shell interativo para ver como essa conversão funciona:

```
>>> SÍMBOLOS =  
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456  
7890!  
>>> len (SÍMBOLOS)  
66  
>>> SÍMBOLOS [0]  
'UMA'  
>>> SÍMBOLOS [30]  
'e'
```

Podemos representar caracteres de texto por seus índices inteiros no conjunto de símbolos. No entanto, também precisamos de uma maneira de combinar esses pequenos inteiros em um inteiro grande que represente um bloco.

Para criar um bloco, multiplicamos o índice do conjunto de símbolos de um caractere pelo tamanho do conjunto de símbolos aumentado para poderes crescentes. O bloco é a soma de todos esses números. Vejamos um exemplo de combinação de inteiros pequenos em um bloco grande para a string 'Howdy' usando as etapas a seguir.

O inteiro do bloco começa em 0 e o conjunto de símbolos tem 66 caracteres. Digite o seguinte no shell interativo usando esses números como um exemplo:

```
>>> blockInteger = 0  
>>> len (SÍMBOLOS)  
66
```

O primeiro caractere na mensagem 'Howdy' é 'H' , então encontramos o índice do conjunto de símbolos para esse caractere, assim:

```
>>> SYMBOLS.index ('H')
```

```
7
```

Como esse é o primeiro caractere da mensagem, multiplicamos seu índice de conjunto de símbolos por 66⁰ (em Python, o operador de expoente é `**`), que é avaliado como 7 . Adicionamos esse valor ao bloco:

```
>>> 7 * (66 ** 0)
```

```
7
```

```
>>> blockInteger = blockInteger + 7
```

Em seguida, encontramos o índice do conjunto de símbolos para 'o' , o segundo caractere em 'Howdy' . Como esse é o segundo caractere da mensagem, multiplicamos o índice do conjunto de símbolos para 'o' por 66¹ em vez de 66⁰ e, em seguida, adicione-o ao bloco:

```
>>> SYMBOLS.index ('o')
```

```
40
```

```
>>> blockInteger += 40 * (66 ** 1)
```

```
>>> blockInteger
```

```
2647
```

O bloco é agora 2647 . Podemos encurtar o processo de encontrar o índice do conjunto de símbolos para cada personagem usando uma única linha de código:

```
>>> blockInteger += SYMBOLS.index ('w') * (len (SYMBOLS) ** 2)
```

```
>>> blockInteger += SYMBOLS.index ('d') * (len (SYMBOLS) ** 3)
```

```
>>> blockInteger += SYMBOLS.index ('y') * (len (SYMBOLS) ** 4)
```

```
>>> blockInteger
```

```
957285919
```

A codificação 'Howdy' em um único bloco inteiro grande produz o inteiro 957,285,919, que se refere exclusivamente à string. Ao continuar usando potências maiores e maiores de 66, podemos usar um inteiro grande para representar uma string de qualquer tamanho até o tamanho do bloco. Por exemplo, 277,981 é um bloco que representa a string '42! ' e 10.627.106.169.278.065.987.481.042.235.655.809.080.528 representa a cadeia "Eu nomeei meu gato Zophie" .

Como nosso tamanho de bloco é 169, só podemos criptografar até 169 caracteres em um único bloco. Se a mensagem que queremos codificar tem mais de 169 caracteres, podemos usar mais blocos. No programa `publicKeyCipher.py` ,

usaremos vírgulas para separar os blocos, para que o usuário possa identificar quando um bloco termina e o próximo começa.

A Tabela 24-1 contém uma mensagem de exemplo dividida em blocos e mostra o número inteiro que representa cada bloco. Cada bloco pode armazenar no máximo 169 caracteres da mensagem.

Tabela 24-1: uma mensagem dividida em blocos

	mensagem	Bloquear inteiro
1º bloco (169 caracteres)	Alan Mathison Turing foi um criptoanalista britânico e cientista da computação. Ele foi altamente influente no desenvolvimento da ciência da computação e forneceu uma formalização de os conceitos de algoritmo e computação com a máquina de Turing. Turing é amplamente considerado o pai da ciência da computação e da inteligência artificial. Durante W	3013810338120027658120611166332270159047176083261525954313915757971407078374850898659286061395648657712401264848061468979998711065254489615586402779944568481071584216206595263324642598595698762771963146093256595688769305982915401292341459466451130935260873543216661377362346098640381109985392482698
2º bloco (169 caracteres)	Segunda Guerra Mundial, ele	1106890780922147455215935080195634373132610270819271365148408547540267775279195807872272026708702634070281109709555761008581376819190225258032442691476944762174257390214806410726987166909365500457701428029424452471175143504911739898604483879159730789371948601125747980165875644527924515615863348631
Terceiro		

bloco trabalhou para o 15836797549616019144289524472175836978758
(82 Código do 63597486412804750943905655902273209591807
caracteres) Governo e 29054194485980905328691576422832688749509
 Escola Cypher 27709935741799076979034
 em Bletchley
 Park.

Neste exemplo, a mensagem de 420 caracteres consiste em dois blocos de 169 caracteres e precisa de um terceiro bloco para os 82 caracteres restantes.

A Matemática da Criptografia e Descriptografia de Cifra de Chave Pública

Agora que você sabe como converter caracteres em inteiros de blocos, vamos explorar como a criptografia de chave pública usa a matemática para criptografar cada bloco.

Aqui estão as equações gerais para a cifra da chave pública:

$$C = M^e \text{ mod } n$$

$$M = C^d \text{ mod } n$$

Usamos a primeira equação para criptografar cada bloco inteiro e a segunda para descriptografar. M representa um inteiro do bloco de mensagens e C é um inteiro do bloco de texto cifrado. Os números e e n formam a chave pública para criptografia, e os números d e n formam a chave privada. Lembre-se de que todos, incluindo o criptoanalista, têm acesso à chave pública (e, n).

Normalmente, criamos a mensagem criptografada, aumentando cada inteiro do bloco, como aqueles que calculamos na seção anterior, para o poder e modificando o resultado por n . Este cálculo resulta em um inteiro que representa o bloco criptografado C . A combinação de todos os blocos resulta na mensagem criptografada completa.

Por exemplo, vamos criptografar a cadeia de cinco caracteres 'Howdy' e enviá-la para Alice. Quando convertida em um bloco inteiro, a mensagem é [957285919] (a mensagem completa cabe em um bloco, portanto, há apenas um inteiro no valor da lista). A chave pública de Alice é 64 bits, o que é muito pequeno para ser seguro, mas vamos usá-lo para simplificar nossa saída neste exemplo. Seu n é 116.284.564.958.604.315.258.674.918.142.848.831.759 e e é 13.805.220.545.651.593.223. (Esses números seriam muito maiores para chaves de 1024 bits.)

Para criptografar, calculamos $(957.285.919 \cdot 13.805.220.545.651.593.223) \% 116.284.564.958.604.315.258.674.918.142.848.831.759$ passando esses números para a função `pow()` do Python, assim:

```
>>> pow(957285919, 13805220545651593223,  
116284564958604315258674918142848831759)  
43924807641574602969334176505118775186
```

A função `pow()` do Python usa um truque matemático chamado exponenciação modular para calcular rapidamente um expoente tão grande. De fato, avaliar a expressão $(957285919^{13805220545651593223}) \% 116.284.564.958.604.315.258.674.918.142.848.831.759$ produziria a mesma resposta, mas levaria horas para ser concluído. O inteiro que `pow()` retorna é um bloco que representa a mensagem criptografada.

Para descriptografar, o destinatário da mensagem criptografada precisa ter a chave privada (d, n), elevar cada inteiro do bloco criptografado para a energia d e, em seguida, modificar por n . Quando todos os blocos descriptografados são decodificados em caracteres e combinados, o destinatário receberia a mensagem de texto original.

Por exemplo, Alice tenta decifrar o bloco inteiro 43.924.807.641.574.602.969.334.176.505.118.775.186. Sua chave privada é o mesmo que ela n chave pública, mas sua chave privada tem um d de 72.424.475.949.690.145.396.970.707.764.378.340.583. Para decifrar, ela executa o seguinte:

```
>>> pow(43924807641574602969334176505118775186,  
72424475949690145396970707764378340583,  
116284564958604315258674918142848831759)  
957285919
```

Quando convertemos o bloco inteiro 957285919 em uma string, obtemos 'Howdy', que é a mensagem original. Em seguida, você aprenderá a converter um bloco em uma string.

Convertendo um bloco em uma string

Para descriptografar um bloco para o inteiro original do bloco, o primeiro passo é convertê-lo nos inteiros pequenos para cada caractere de texto. Esse processo começa com o último caractere que foi adicionado ao bloco. Usamos operadores de divisão de piso e mod para calcular os inteiros pequenos para cada caractere

de texto.

Lembre-se de que o inteiro do bloco no exemplo anterior 'Howdy' era 957285919 ; a mensagem original tinha cinco caracteres, fazendo o último índice do personagem 4; e o conjunto de símbolos usado para a mensagem tinha 66 caracteres. Para determinar o índice do conjunto de símbolos do último caractere, calculamos $957.285.919 / 66^4$ e arredondamos para baixo, o que resulta em 50. Podemos usar o operador de divisão inteira (//) para dividir e arredondar para baixo. O caractere no índice 50 no conjunto de símbolos (SYMBOLS [50]) é 'y' , que é de fato o último caractere da mensagem 'Howdy' .

No shell interativo, calculamos este bloco inteiro usando o seguinte código:

```
>>> blockInteger = 957285919
>>> SÍMBOLOS =
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456
7890!
>>> blockInteger // (66 ** 4)
50
>>> SÍMBOLOS [50]
'y'
```

O próximo passo é modificar o bloco inteiro por 66^4 para obter o próximo inteiro do bloco. O cálculo de $957.285.919 \% (66^4)$ resulta em 8.549.119, que é o valor inteiro do bloco para a string 'Howd' . Podemos determinar o último caractere deste bloco usando a divisão de piso de (66^3) . Digite o seguinte no shell interativo para fazer isso:

```
>>> blockInteger = 8549119
>>> SÍMBOLOS [blockInteger // (len (SÍMBOLOS) ** 3)]
'd'
```

O último caractere deste bloco é 'd' , fazendo a string convertida até agora 'dy' . Podemos remover esse caractere do inteiro do bloco como fizemos antes:

```
>>> blockInteger = blockInteger% (len (SYMBOLS) ** 3)
>>> blockInteger
211735
```

O inteiro 211735 é o bloco da string 'How' . Ao continuar o processo, podemos recuperar a string completa do bloco, da seguinte forma:

```
>>> SÍMBOLOS [blockInteger // (len (SÍMBOLOS) ** 2)]
```

```
'W'
>>> blockInteger = blockInteger% (len (SYMBOLS) ** 2)
>>> SÍMBOLOS [blockInteger // (len (SÍMBOLOS) ** 1)]
'o'
>>> blockInteger = blockInteger% (len (SYMBOLS) ** 1)
>>> SÍMBOLOS [blockInteger // (len (SÍMBOLOS) ** 0)]
'H'
```

Agora você sabe como os caracteres da string 'Howdy' são recuperados do valor inteiro do bloco original 957285919 .

Por que não podemos invadir a criptografia de chave pública

Todos os tipos diferentes de ataques criptográficos que usamos neste livro são inúteis contra a codificação de chave pública quando ela é implementada corretamente. Aqui estão algumas razões do porquê:

1. O ataque de força bruta não funcionará porque há muitas chaves possíveis para checar.
2. Um ataque de dicionário não funcionará porque as chaves são baseadas em números, não em palavras.
3. Um ataque de padrão de palavra não funcionará porque a mesma palavra de texto simples pode ser criptografada de forma diferente, dependendo de onde aparece o bloco.
4. A análise de frequência não funciona porque um único bloco criptografado representa vários caracteres; não podemos obter uma contagem de frequência dos caracteres individuais.

Como a chave pública (e, n) é conhecida por todos, se um criptoanalista puder interceptar o texto cifrado, eles saberiam e, n e C . Mas sem saber d , é matematicamente impossível resolver para M , a mensagem original.

Lembre-se do [Capítulo 23](#) que e é relativamente primo com o número $(p - 1) \times (q - 1)$ e que d é o inverso modular de e e $(p - 1) \times (q - 1)$. No [Capítulo 13](#), você aprendeu que o inverso modular de dois números é calculado encontrando i para a equação $(ai) \% m = 1$, onde a e m são dois números no problema modular $a \text{ mod } m$. Isso significa que o criptoanalista sabe que d é o inverso de $e \text{ mod } (p - 1) \times (q - 1)$, então podemos encontrar d para obter toda a chave de decriptação resolvendo a equação $(ed) \text{ mod } (p - 1) \times (q - 1) = 1$; no entanto, não há como saber o que $(p - 1) \times (q - 1)$ é.

Nós sabemos os tamanhos das chaves do arquivo de chave pública, então o criptoanalista sabe que p e q são menores que 2 1024 e que e é relativamente primo com $(p - 1) \times (q - 1)$. Mas e é relativamente primo com *muitos* números, e encontrar $(p - 1) \times (q - 1)$ de um intervalo de 0 a 2 1024 números possíveis é um problema muito grande para a força bruta.

Embora não seja suficiente para decifrar o código, o criptoanalista pode obter outra sugestão da chave pública. A chave pública é composta dos dois números (e, n), e nós sabemos $n = p \times q$ porque foi assim que calculamos n quando criamos as chaves pública e privada no [Capítulo 23](#). E porque p, q são números primos, para um dado número n , pode haver exatamente dois números que podem ser p, q .

Lembre-se de que um número primo não tem fatores além de 1 e de si mesmo. Portanto, se você multiplicar dois números primos, o produto terá 1 e ele e os dois números primos com os quais você começou como seus únicos fatores.

Portanto, para hackar a cifra da chave pública, tudo o que precisamos fazer é descobrir os fatores de n . Como sabemos que dois e apenas dois números podem ser multiplicados para obter n , não teremos muitos números diferentes para escolher. Depois de descobrirmos quais dois números primos (p e q) quando multiplicamos para n , podemos calcular $(p - 1) \times (q - 1)$ e depois usar esse resultado para calcular d . Este cálculo parece bastante fácil de fazer. Vamos usar a função `isPrime()` que escrevemos no programa `primeNum.py` no [Capítulo 22](#) para fazer o cálculo.

Podemos modificar o `isPrime()` para retornar os primeiros fatores que encontrar, porque sabemos que só pode haver dois fatores de n além de 1 e n :

```
def isPrime (num):
    # Retorna (p, q) onde p e q são fatores de num.
    # Veja se num é divisível por qualquer número até a raiz quadrada de num:
    para i no intervalo (2, int (math.sqrt (num)) + 1):
        if num% i == 0:
            retorno (i, num / i)
    return Nenhum # Nenhum fator existe para num; num deve ser primo.
```

Se escrevêssemos um programa hacker de cifra de chave pública, poderíamos apenas chamar essa função, passá-la n (que obteríamos do arquivo de chave pública) e esperar que ela encontrasse os fatores p e q . Então podemos encontrar o que $(p - 1) \times (q - 1)$ é, o que significa que podemos calcular o mod inverso de

$e \text{ mod } (p - 1) \times (q - 1)$ para obter d , a chave de decriptografia. Então seria fácil calcular M , a mensagem de texto simples.

Mas tem um problema. Lembre-se de que n é um número com aproximadamente 600 dígitos. A função `math.sqrt()` do Python não pode manipular um número tão grande, então ele fornece uma mensagem de erro. Mas mesmo que pudesse processar esse número, o Python executaria esse loop por muito tempo. Por exemplo, mesmo que seu computador continuasse a rodar 5 bilhões de anos a partir de agora, ainda não há quase nenhuma chance de encontrar os fatores de n . É assim que esses números são grandes.

E esta é exatamente a força da cifra da chave pública: *matematicamente, não há nenhum atalho para encontrar os fatores de um número*. É fácil chegar a dois números primos p e multiplicá-los juntos para obter n . Mas é quase impossível pegar um número n e descobrir o que p seria. Por exemplo, quando você olha para um número pequeno como 15, pode facilmente dizer que 5 e 3 são dois números que, quando multiplicados por 15. Mas é outra coisa totalmente diferente tentar descobrir os fatores de um número como 178.565.887.643.607.245.654.502.737. Esse fato faz com que a codificação da chave pública seja praticamente impossível de ser quebrada.

Código-fonte para o programa de codificação de chave pública

Abra uma nova janela do editor de **arquivos** selecionando **File ▶ New File**. Digite o seguinte código no editor de arquivos e salve-o como `publicKeyCipher.py`.

chave pública

Cipher.py

1. # Cifra de chave pública
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. import sys, matemática
- 5
6. # As chaves pública e privada para este programa são criadas por
7. # o programa makePublicPrivateKeys.py.
8. # Este programa deve ser executado na mesma pasta que os arquivos de chaves.
- 9
10. SÍMBOLOS =

```
'ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345  
67890!  
11  
12. def main ():  
13. # Executa um teste que criptografa uma mensagem em um arquivo ou  
descriptografa uma mensagem  
14. # de um arquivo.  
15. filename = 'encrypted_file.txt' # O arquivo para gravar / ler.  
16. mode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.  
17  
18. if mode == 'encriptar':  
19. message = 'Jornalistas pertencem à sarjeta porque é onde  
as classes dominantes lançam seus segredos culpados. Gerald Priestland.  
Os fundadores deram a imprensa livre a proteção que deve  
tem que descobrir os segredos do governo e informar as pessoas.  
Hugo Black.  
20. pubKeyFilename = 'al_sweigart_pubkey.txt'  
21. print ('Criptografando e gravando em% s ...% (filename))  
22. encryptedText = encryptAndWriteToFile (nome do arquivo,  
pubKeyFilename,  
mensagem)  
23  
24. print ("Texto criptografado:")  
25. print (encryptedText)  
26  
27. modo elif == 'decifrar':  
28. privKeyFilename = 'al_sweigart_privkey.txt'  
29. print ('Lendo de% s e descriptografando ...% (filename))  
30. decryptedText = readFromFileAndDecrypt (nome do arquivo,  
privKeyFilename)  
31  
32. print ("Texto descriptografado:")  
33. print (decryptedText)  
34  
35  
36. def getBlocksFromText (mensagem, tamanho do bloco):  
37. # Converte uma mensagem de string em uma lista de inteiros de bloco.
```

```
38. para o personagem na mensagem:  
39. se o personagem não estiver em SÍMBOLOS:  
40. print ('ERRO: O conjunto de símbolos não possui o caractere% s'%  
(personagem))  
41. sys.exit ()  
42. blockInts = []  
43. para blockStart no intervalo (0, len (mensagem), blockSize):  
44. # Calcular o inteiro do bloco para este bloco de texto:  
45. blockInt = 0  
46. para i na faixa (blockStart, min (blockStart + blockSize,  
len (mensagem))):  
47. blockInt += (SYMBOLS.index (mensagem [i])) * (len (SYMBOLS) **  
(eu% blockSize))  
48. blockInts.append (blockInt)  
49. return blockInts  
50  
51  
52. def getTextFromBlocks (blockInts, messageLength, blockSize):  
53. # Converte uma lista de inteiros de bloco na string de mensagem original.  
54. # O tamanho original da mensagem é necessário para converter corretamente  
o último  
55. # bloco inteiro.  
56. message = []  
57. para blockInt em blockInts:  
58. blockMessage = []  
59. para i no intervalo (blockSize - 1, -1, -1):  
60. se len (mensagem) + i <messageLength:  
61. # Decodifica a string de mensagem para o 128 (ou qualquer  
62. # blockSize está definido para) caracteres deste inteiro do bloco:  
63. charIndex = blockInt // (len (SÍMBOLOS) ** i)  
64. blockInt = blockInt% (len (SÍMBOLOS) ** i)  
65. blockMessage.insert (0, SYMBOLS [charIndex])  
66. message.extend (blockMessage)  
67. return " .join (mensagem)  
68  
69  
70. def encryptMessage (message, key, blockSize):
```

```
71. # Converte a string de mensagem em uma lista de inteiros de bloco, e depois
72. # criptografa cada inteiro do bloco. Passe a chave PUBLIC para criptografar.
73. encryptedBlocks = []
74. n, e = chave
75
76. para o bloco em getBlocksFromText (message, blockSize):
77. # ciphertext = plaintext ^ e mod n
78. encryptedBlocks.append (pow (bloco, e, n))
79. return encryptedBlocks
80
81
82. def decryptMessage (encryptedBlocks, messageLength, key, blockSize):
83. # Descriptografa uma lista de ints de bloqueio criptografados na mensagem
original
84. # string. O comprimento da mensagem original é necessário para
descriptografar corretamente
85. # o último bloco. Certifique-se de passar a chave PRIVATE para
descriptografar.
86. decryptedBlocks = []
87. n, d = chave
88. para bloquear em bloqueios criptografados:
89. # plaintext = texto cifrado ^ d mod n
90. decryptedBlocks.append (pow (bloco, d, n))
91. return getTextFromBlocks (decryptedBlocks, messageLength, blockSize)
92
93
94. def readKeyFile (keyFilename):
95. # Dado o nome do arquivo de um arquivo que contém uma chave pública ou
privada,
96. # retorna a chave como um valor de tupla (n, e) ou (n, d).
97. fo = open (keyFilename)
98. content = fo.read ()
99. fo.close ()
100. keySize, n, EorD = content.split(',')
101. return (int (tamanho da chave), int (n), int (EorD))
102
103
```

```
104. def encryptAndWriteToFile (messageFilename, keyFilename, mensagem,  
blockSize = nenhum):  
105. # Usando uma chave de um arquivo de chave, criptografe a mensagem e  
salve-a em um arquivo.  
106. # arquivo. Retorna a string de mensagem criptografada.  
107. keySize, n, e = readKeyFile (keyFilename)  
108. if blockSize == Nenhum:  
109. # Se blockSize não for dado, configure-o para o maior tamanho permitido  
por  
o tamanho da chave e o tamanho do conjunto de símbolos.  
110. blockSize = int (math.log (2 ** keySize, len (SÍMBOLOS)))  
111. # Verifique se o tamanho da chave é grande o suficiente para o tamanho do  
bloco:  
112. se não (math.log (2 ** keySize, len (SYMBOLS))> = blockSize):  
113. sys.exit ('ERRO: O tamanho do bloco é muito grande para a chave e o  
símbolo  
tamanho do conjunto. Você especificou o arquivo de chave correto e  
criptografou  
Arquivo?')  
114. # Criptografar a mensagem:  
115. encryptedBlocks = encryptMessage (mensagem, (n, e), blockSize)  
116  
117. # Converta os grandes valores int em um valor de string:  
118. para i no intervalo (len (encryptedBlocks)):  
119. encryptedBlocks [i] = str (encryptedBlocks [i])  
120. encryptedContent = ','. Join (encryptedBlocks)  
121  
122. # Escreva a string criptografada no arquivo de saída:  
123. encryptedContent = '% s_% s_% s%' (len (mensagem), blockSize,  
encryptedContent)  
124. fo = open (messageFilename, 'w')  
125. fo.write (encryptedContent)  
126. fo.close ()  
127. # Também retorna a string criptografada:  
128. return encryptedContent  
129  
130
```

```
131. def readFromFileAndDecrypt (messageFilename, keyFilename):
132. # Usando uma chave de um arquivo de chave, leia uma mensagem
criptografada de um arquivo
133. # e depois descriptografar. Retorna a string de mensagem descriptografada.
134. keySize, n, d = readKeyFile (keyFilename)
135
136
137. # Leia o tamanho da mensagem e a mensagem criptografada do arquivo:
138. fo = open (messageFilename)
139. content = fo.read ()
140. messageLength, blockSize, encryptedMessage = content.split ('_')
141. messageLength = int (messageLength)
142. blockSize = int (tamanho do bloco)
143
144. # Verifique se o tamanho da chave é grande o suficiente para o tamanho do
bloco:
145. se não (math.log (2 ** keySize, len (SYMBOLS))> = blockSize):
146. sys.exit ('ERRO: O tamanho do bloco é muito grande para a chave e o
símbolo
tamanho do conjunto. Você especificou o arquivo de chave correto e
criptografou
Arquivo?')
147
148. # Converta a mensagem criptografada em grandes valores int:
149. encryptedBlocks = []
150. para bloco em encryptedMessage.split (','):
151. encryptedBlocks.append (int (bloco))
152
153. # Descriptografar os grandes valores int:
154. return decryptMessage (encryptedBlocks, messageLength, (n, d),
tamanho do bloco)
155.
156
157. # Se publicKeyCipher.py for executado (em vez de importado como um
módulo), chame
158. # a função main () .
159. if __name__ == '__main__':
```

160. main ()

Execução de exemplo do programa de codificação de chave pública

Vamos tentar executar o programa *publicKeyCipher.py* para criptografar uma mensagem secreta. Para enviar uma mensagem secreta para alguém usando este programa, obtenha o arquivo de chave pública dessa pessoa e coloque-o no mesmo diretório que o arquivo de programa.

Para criptografar uma mensagem, verifique se a variável de modo na linha 16 está configurada para a string 'encrypt'. Atualize a variável de mensagem na linha 19 para a cadeia de mensagens que você deseja criptografar. Em seguida, defina a variável *pubKeyFilename* na linha 20 para o nome do arquivo do arquivo de chave pública. A variável *filename* na linha 21 contém um nome de arquivo no qual o texto cifrado é gravado. As variáveis *filename*, *pubKeyFilename* e *message* são passadas para *encryptAndWriteToFile ()* para criptografar a mensagem e salvá-la em um arquivo.

Quando você executa o programa, a saída deve ficar assim:

Criptografando e escrevendo para *encrypted_file.txt* ...

Texto criptografado:

```
258_169_451084515249071382368598160394837212194759075902379039182  
48566603013231572537249780228617020983244277382842255301862133801  
83392298908904649695569377970724343149165228396922770345794635947  
93072346500886898507443612627071299717824076104502080479271296878  
70182779184902972157857592572908558122210889070169049830255421744  
60153100891558762342778833813452473536806245856296729397095570161  
51241925684094837372334973040879696240435161582216894541480960207  
74772465708958607695479122809498585662785064751254235489968738346  
, 1253384  
33369751155397613322504026998688351506230175824381168400492360835  
37194564531336584762711760352485970219723164545265450694528387663  
40668777211355113134542525897339719622190160666149783903786111759  
94295456059017143390825427250151405309856851172325987781765456381  
38592446600919103910996210281921774151961564699729773052126762937  
46682406932300321410973125564006299615186357994786521960723164249  
63394249489758046609236166827672429482963016783120418289344737868  
133539825048880814063389057192492939651199537310635280371
```

O programa grava essa saída em um arquivo denominado *encrypted_file.txt*.

Esta é a criptografia da string na variável message na linha 19. Como a chave pública que você está usando é provavelmente diferente da minha, a saída que você obtém pode ser diferente, mas o formato da saída deve ser o mesmo. Como você pode ver neste exemplo, a criptografia é dividida em dois blocos, ou dois inteiros grandes, separados por uma vírgula.

O número 258 no início da criptografia representa o tamanho da mensagem original e é seguido por um sublinhado e outro número 169 , que representa o tamanho do bloco. Para descriptografar essa mensagem, altere a variável de modo para 'decifrar' e execute o programa novamente. Como na criptografia, certifique-se de que privKeyFilename na linha 28 esteja configurado para o nome da chave privada e que este arquivo esteja na mesma pasta que *publicKeyCipher.py* . Além disso, verifique se o arquivo *criptografado* , *encrypted_file.txt* , está na mesma pasta que *publicKeyCipher.py* . Quando você executa o programa, a mensagem criptografada em *encrypted_file.txt* é descriptografada e a saída deve ficar assim:

Lendo de encrypted_file.txt e descriptografando ...

Texto descriptografado:

Os jornalistas pertencem à sarjeta porque é aí que as classes dominantes lançam seus segredos culpados. Gerald Priestland. Os fundadores deram a liberdade pressione a proteção que deve ter para descobrir os segredos do governo e informar as pessoas. Hugo Black.

Observe que o programa *publicKeyCipher.py* só pode criptografar e descriptografar arquivos de texto simples (simples).

Vamos dar uma olhada mais de perto no código-fonte do programa *publicKeyCipher.py* .

Configurando o Programa

A cifra da chave pública trabalha com números, então vamos converter nossa mensagem de string em um inteiro. Este inteiro é calculado com base nos índices no conjunto de símbolos, que é armazenado na variável SYMBOLS na linha 10.

1. # Cifra de chave pública
2. # <https://www.nostarch.com/crackingcodes/> (Licenciado pelo BSD)
- 3
4. import sys, matemática

5

6. # As chaves pública e privada para este programa são criadas por

7. # o programa makePublicPrivateKeys.py.

8. # Este programa deve ser executado na mesma pasta que os arquivos de chaves.

9

10. SÍMBOLOS =

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
67890!'

Como o programa determina se deve criptografar ou descriptografar

O programa *publicKeyCipher.py* determina se deve criptografar ou descriptografar um arquivo e qual arquivo de chave deve ser usado, armazenando valores em variáveis. Enquanto observamos como essas variáveis funcionam, também veremos como o programa imprime a saída de criptografia e descriptografia.

Dizemos ao programa se ele deve criptografar ou descriptografar dentro de main () :

12. def main ():

13. # Executa um teste que criptografa uma mensagem em um arquivo ou descriptografa uma mensagem

14. # de um arquivo.

15. filename = 'encrypted_file.txt' # O arquivo para gravar / ler.

16. mode = 'encrypt' # Defina como 'encrypt' ou 'decrypt'.

Se o modo na linha 16 estiver configurado para 'criptografar' , o programa criptografará uma mensagem gravando-a no arquivo especificado em filename . Se o modo estiver definido como 'descriptografar' , o programa lerá o conteúdo de um arquivo criptografado (especificado por nome de arquivo) para descriptografá-lo.

As linhas 18 a 25 especificam o que o programa deve fazer se confirmar que o usuário deseja criptografar um arquivo.

18. if mode == 'encriptar':

19. message = 'Jornalistas pertencem à sarjeta porque é onde as classes dominantes lançam seus segredos culpados. Gerald Priestland.

Os fundadores deram a imprensa livre a proteção que deve

tem que descobrir os segredos do governo e informar as pessoas.
Hugo Black.

```
20. pubKeyFilename = 'al_sweigart_pubkey.txt'  
21. print ('Criptografando e gravando em% s ...% (filename))  
22. encryptedText = encryptAndWriteToFile (nome do arquivo,  
pubKeyFilename,  
mensagem)  
23  
24. print ('Texto criptografado:')  
25. print (encryptedText)
```

A variável de mensagem na linha 19 contém o texto a ser criptografado, e pubKeyFilename na linha 20 contém o nome do arquivo da chave pública, que é *al_sweigart_pubkey.txt* neste exemplo. Lembre-se de que a mensagem só pode conter caracteres na variável SYMBOLS , o símbolo definido para essa cifra.A linha 22 chama a função encryptAndWriteToFile () , que criptografa a mensagem usando a chave pública e grava a mensagem criptografada no arquivo especificado por filename .

As linhas 27 a 28 informam ao programa o que fazer se o modo estiver definido como 'decifrar' . Em vez de criptografar, o programa lê a partir do arquivo de chave privada em privKeyFilename na linha 28.

```
27. modo elif == 'decifrar':  
28. privKeyFilename = 'al_sweigart_privkey.txt'  
29. print ('Lendo de% s e descriptografando ...% (filename))  
30. decryptedText = readFromFileAndDecrypt (nome_do_arquivo,  
privKeyFilename)  
31  
32. print ('Decrypted text:')  
33. print (decryptedText)
```

Em seguida, passamos as variáveis filename e privKeyFilename para a função readFromFileAndDecrypt () (definida posteriormente no programa), que retorna a mensagem descriptografada. A linha 30 armazena o valor de retorno de readFromFileAndDecrypt () em decryptedText e a linha 33 o imprime na tela. Este é o fim da função main () .

Agora vamos ver como executar as outras etapas da codificação de chave pública, como converter a mensagem em blocos.

Convertendo Strings em Blocos com getBlocksFromText ()

Vamos ver como o programa converte uma string de mensagem em blocos de 128 bytes. A função `getBlocksFromText ()` na linha 36 recebe uma mensagem e um tamanho de bloco como parâmetros para retornar uma lista de blocos, ou uma lista de valores inteiros grandes, que representa a mensagem.

```
36. def getBlocksFromText (message, blockSize):
37. # Converte uma mensagem de string em uma lista de inteiros de bloco.
38. para caracteres na mensagem:
39. se caractere não estiver em SYMBOLS:
40. print ('ERRO: O conjunto de símbolos não possui o caractere% s'% 
(caractere))
41. sys.exit ()
```

As linhas 38 a 41 asseguram que o parâmetro `message` contenha apenas caracteres de texto que estão no conjunto de símbolos na variável `SYMBOLS`. O parâmetro `blockSize` é opcional e pode receber qualquer tamanho de bloco. Para criar blocos, primeiro convertemos a string em bytes.

Para fazer um bloco, combinamos todos os índices do conjunto de símbolos em um inteiro grande, como fizemos em “[Convertendo uma String em um Bloco](#)” na [página 357](#). Usaremos a lista vazia `blockInts` na linha 42 para armazenar os blocos quando os criarmos.

```
42. blockInts = []
```

Queremos que os blocos sejam bytes `blockSize`, mas quando uma mensagem não é divisível por `blockSize`, o último bloco será menor que os caracteres `blockSize`. Para lidar com essa situação, usamos a função `min ()`.

As funções `min ()` e `max ()`

A função `min ()` retorna o menor valor de seus argumentos. Digite o seguinte no shell interativo para ver como a função `min ()` funciona:

```
>>> min (13, 32, 13, 15, 17, 39)
13
```

Você também pode passar uma única lista ou valor de tupla como um argumento para `min ()`. Digite o seguinte no shell interativo para ver um exemplo:

```
>>> min ([31, 26, 20, 13, 12, 36])
12
```

```
>>> spam = (10, 37, 37, 43, 3)
>>> min (spam)
3
```

Nesse caso, `min (spam)` retorna o menor valor na lista ou tupla. O oposto de `min ()` é `max ()`, que retorna o maior valor de seus argumentos, assim:

```
>>> max (18, 15, 22, 30, 31, 34)
34
```

Vamos voltar ao nosso código para ver como o programa `publicKeyCipher.py` usa `min ()` para garantir que o último bloco de mensagens seja truncado para o tamanho apropriado.

Armazenando Blocos no blockInt

O código dentro do loop `for` na linha 43 cria os inteiros para cada bloco, definindo o valor em `blockStart` para o índice do bloco que está sendo criado.

43. para `blockStart` no intervalo (`0, len (mensagem), blockSize`):

44. # Calcula o inteiro do bloco para este bloco de texto:

45. `blockInt = 0`

46. para `i` no intervalo (`blockStart, min (blockStart + blockSize, len (mensagem))`):

Vamos armazenar o bloco que criamos no `blockInt`, que inicialmente definimos como 0 na linha 45. O loop `for` na linha 46 define `i` como sendo os índices de todos os caracteres que estarão no bloco da mensagem. Os índices devem começar em `blockStart` e ir até `blockStart + blockSize` ou `len (message)`, o que for menor. A chamada `min ()` na linha 46 retorna a menor dessas duas expressões.

O segundo argumento para `range ()` na linha 46 deve ser o menor de `blockStart + blockSize` e `len (message)` porque cada bloco é sempre composto por 128 caracteres (ou qualquer valor que esteja em `blockSize`), *exceto* pelo último bloco. O último bloco pode ter exatamente 128 caracteres, mas é mais provável que seja menor que os 128 caracteres completos. Nesse caso, queremos que eu pare em `len (mensagem)` porque esse é o último índice na mensagem.

Depois de termos os caracteres que compõem o bloco, usamos matemática para transformar os caracteres em um inteiro grande. Lembre-se de “ [Convertendo uma Cadeia de Caracteres em um Bloco](#) ”, na [página 358](#) , criamos um inteiro grande multiplicando o valor inteiro do índice do conjunto de símbolos de cada

caractere por 66 de *índice de caracteres* (66 é o comprimento da sequência SYMBOLS). Para fazer isso no código, calculamos SYMBOLS.index (mensagem [i]) (o valor inteiro do índice do conjunto de símbolos do caractere) multiplicado por (len (SYMBOLS) ** (i% blockSize)) para cada caractere e adicionamos cada resultado para bloquear .

47. `blockInt += (SYMBOLS.index (mensagem [i])) * (len (SYMBOLS) ** (i% blockSize))`

Queremos que o expoente seja o índice relativo ao *bloco da iteração atual* , que é sempre de 0 a blockSize . Não podemos usar a variável i diretamente como a parte do índice de caracteres da equação, porque ela se refere ao índice em toda a cadeia de mensagens , que possui índices de 0 até len (mensagem) . Usar i resultaria em um inteiro muito maior que 66. Modificando i por blockSize , podemos obter o índice relativo ao bloco, e é por isso que a linha 47 é len (SYMBOLS) ** (i% blockSize) em vez de simplesmente len (SÍMBOLOS) ** i

Depois que o loop for na linha 46 for concluído, o inteiro para o bloco foi calculado. Usamos o código na linha 48 para acrescentar este bloco inteiro à lista blockInts . A próxima iteração do loop for na linha 43 calcula o inteiro do bloco para o próximo bloco da mensagem.

48. `blockInts.append (blockInt)`

49. `return blockInts`

Depois que o loop for na linha 43 terminar, todos os inteiros de bloco devem ter sido calculados e armazenados na lista blockInts . A linha 49 retorna blockInts de getBlocksFromText () .

Neste ponto, convertemos toda a string da mensagem em números inteiros de blocos, mas também precisamos transformar os números inteiros em blocos na mensagem de texto original para o processo de descriptografia, que é o que faremos a seguir.

Usando getTextFromBlocks () para descriptografar

A função getTextFromBlocks () na linha 52 faz o oposto de getBlocksFromText () . Essa função recebe uma lista de inteiros de bloco como o parâmetro blockInts , o tamanho da mensagem e o tamanho do bloco para retornar o valor da string que esses blocos representam. Precisamos do tamanho da mensagem codificada em messageLength , porque a função getTextFromBlocks () usa essa

informação para obter a string do último inteiro do bloco quando não é tamanho blockSize . Esse processo foi descrito em “ [Convertendo um bloco em uma string](#) ” na [página 354](#) .

```
52. def getTextFromBlocks(blockInts, messageLength, blockSize):
53. # Converte uma lista de inteiros de bloco na string de mensagem original.
54. # O tamanho original da mensagem é necessário para converter corretamente
os últimos
55. # blocos inteiros.
56. message = []
```

A lista de mensagens , que é criada como uma lista em branco na linha 56, armazena um valor de string para cada caractere, que calcularemos a partir dos inteiros do bloco em blockInts .

O loop for na linha 57 itera sobre cada inteiro do bloco na lista blockInts . Dentro do loop for , o código nas linhas 58 a 65 calcula as letras que estão no bloco da iteração atual.

```
57. para blockInt em blockInts:
58.     blockMessage = []
59.     para i em intervalo (blockSize - 1, -1, -1):
```

O código em getTextFromBlocks () divide cada inteiro do bloco em inteiros blockSize , onde cada um representa o índice do conjunto de símbolos de um caractere. Devemos trabalhar para trás para extrair os índices do conjunto de símbolos de blockInt porque quando criptografamos a mensagem, começamos com os expoentes menores (66 0 , 66 1 , 66 2 e assim por diante), mas ao descriptografar, devemos dividir e modificar usando os maiores expoentes primeiro. É por isso que o loop for na linha 59 começa em blockSize - 1 e depois subtrai 1 em cada iteração para baixo, mas não inclui -1 . Isso significa que o valor de ina última iteração é 0 .

Antes de convertermos o índice do conjunto de símbolos em um caractere, precisamos nos certificar de que não estamos decodificando blocos além do tamanho da mensagem . Para fazer isso, verificamos que o número de caracteres que foram traduzidos dos blocos até agora, len (mensagem) + i , ainda é menor que messageLength na linha 60.

```
60. if len (mensagem) + i < messageLength:
61.     # Decodifica a sequência de caracteres de mensagem para os caracteres 128
```

(ou qualquer

62. # blockSize está definido como) desse inteiro do bloco.

63. charIndex = blockInt // (len (SÍMBOLOS) ** i)

64. blockInt = blockInt% (len (SÍMBOLOS) ** i)

Para obter os caracteres do bloco, seguimos o processo descrito em “

[Convertendo um bloco em uma string](#) ” na [página 354](#) . Nós colocamos cada personagem ema lista de mensagens . Codificar cada bloco realmente inverte os caracteres, o que você viu anteriormente, então não podemos simplesmente acrescentar o caractere decodificado à mensagem . Em vez disso, inserimos o caractere na frente da mensagem , o que precisaremos fazer com o método de lista insert () .

Usando o método de lista insert ()

O método de lista append () só adiciona valores ao final de uma lista, mas o método de lista insert () pode adicionar um valor em *qualquer lugar* da lista. O método insert () usa um índice inteiro de onde na lista inserir o valor e o valor a ser inserido como seus argumentos. Digite o seguinte no shell interativo para ver como o método insert () funciona:

```
>>> spam = [2, 4, 6, 8]
>>> spam.insert(0, 'olá')
>>> spam
['hello', 2, 4, 6, 8]
>>> spam.insert(2, 'mundo')
>>> spam
['olá', 2, 'mundo', 4, 6, 8]
```

Neste exemplo, criamos uma lista de spam e, em seguida, inserimos a string 'hello' no índice 0 . Como você pode ver, podemos inserir valores em qualquer índice existente na lista, como no índice 2 .

Mesclando a lista de mensagens em uma string

Podemos usar a string SYMBOLS para converter o índice do conjunto de símbolos em charIndex em seu caractere correspondente e inserir esse caractere no início da lista no índice 0 .

65. blockMessage.insert (0, SYMBOLS [charIndex])

66. message.extend (blockMessage)

67. return " .join (mensagem)

Essa string é então retornada de getTextFromBlocks () .

Escrevendo a função encryptMessage ()

A função encryptMessage () criptografa cada bloco usando a string de texto simples na mensagem junto com a tupla de dois inteiros da chave pública armazenada na chave , que é criada com a função readKeyFile () que escreveremos mais adiante neste capítulo. A função encryptMessage () retorna uma lista de blocos criptografados.

70. def encryptMessage (message, key, blockSize):
71. # Converte a string de mensagem em uma lista de inteiros de bloco e, em seguida,
72. # criptografa cada inteiro do bloco. Passe a chave PUBLIC para criptografar.
73. encryptedBlocks = []
74. n, e = chave

A linha 73 cria a variável encryptedBlocks , que começa como uma lista vazia que conterá os blocos inteiros. Então a linha 74 atribui os dois inteiros na chave às variáveis n e e . Agora que temos as variáveis de chave pública configuradas, podemos executar a matemática em cada bloco de mensagem para criptografar.

Para criptografar cada bloco, realizamos algumas operações matemáticas que resultam em um novo inteiro, que é o bloco criptografado. Nós elevar o bloco para a e potência e, em seguida, mod-o por n usando POW (bloco, e, n) na linha 78.

76. para bloco em getBlocksFromText (message, blockSize):
77. # texto cifrado = texto sem formatação ^ e mod n
78. encryptedBlocks.append (pow (bloco, e, n))
79. return encryptedBlocks

O número inteiro do bloco criptografado é então anexado ao encryptedBlocks .

Escrevendo a função decryptMessage ()

A função decryptMessage () na linha 82 descriptografa os blocos e retorna a string de mensagem descriptografada. Ele pega a lista de blocos criptografados, o comprimento da mensagem, a chave privada e o tamanho do bloco como parâmetros.

A variável decryptedBlocks que configuramos na linha 86 armazena uma lista dos blocos descriptografados, e usando o truque de atribuição múltipla, os dois

inteiros da tupla chave são colocados em n e d , respectivamente.

82. def decryptMessage (encryptedBlocks, messageLength, key, blockSize):

83. # Descriptografa uma lista de ints de bloco criptografados na mensagem original

84. # string. O tamanho original da mensagem é necessário para descriptografar corretamente

85. # o último bloco. Certifique-se de passar a chave PRIVATE para descriptografar.

86. decryptedBlocks = []

87. n, d = chave

A matemática para descriptografar é a mesma que a matemática da criptografia, exceto que o bloco inteiro está sendo aumentado para d ao invés de e , como você pode ver na linha 90.

88. para bloquear em encryptedBlocks:

89. # plaintext = texto cifrado \wedge d mod n

90. decryptedBlocks.append (pow (bloco, d, n))

Os blocos descriptografados, juntamente com os MessageLength e BLOCKSIZE parâmetros são passados para getTextFromBlocks () de modo que DecryptMessage () retorna o texto simples descriptografado como uma cadeia em linha 91.

91. return getTextFromBlocks (decryptedBlocks, messageLength, blockSize)

Agora que você aprendeu sobre a matemática que faz a criptografia e descriptografia possível, vamos olhar para a forma como o readKeyFile () função lê nos arquivos de chaves públicas e privadas para criar valores de tupla que passados para EncryptMessage () e DecryptMessage () .

Leitura nas chaves públicas e privadas de seus arquivos-chave

A função readKeyFile () é chamada para ler valores de arquivos de chaves criados com o programa *makePublicPrivateKeys.py* , que criamos no [Capítulo 23](#) . O nome do arquivo a ser aberto é passado para keyFilename e o arquivo deve estar na mesma pasta que o programa *publicKeyCipher.py* .

As linhas 97 a 99 abrem esse arquivo e leem o conteúdo como uma string na variável de conteúdo .

94. def readKeyFile (keyFilename):

```
95. # Dado o nome do arquivo de um arquivo que contém uma chave pública ou  
privada,  
96. # retorna a chave como um valor de tupla (n, e) ou (n, d).  
97. fo = open (keyFilename)  
98. conteúdo = fo.read ()  
99. fo.close ()  
100. keySize, n, EORD = content.split(',')  
101. retorno (int (tamanho da chave), int (n), int (EorD))
```

O arquivo de chaves armazena o tamanho da chave em bytes como *n* e *e* ou *d* , dependendo se o arquivo de chave é uma chave de criptografia ou uma chave de descriptografia. Como você aprendeu no capítulo anterior, esses valores foram armazenados como texto e separados por vírgulas, então usamos o método de string split () para dividir a string em conteúdo nas vírgulas. A lista que split () retorna possui três itens, e a atribuição múltipla na linha 100 coloca cada um desses itens nas variáveis keySize , n e EorD , respectivamente.

Lembre-se de que o conteúdo era uma string quando foi lido do arquivo e os itens da lista que split () retorna também serão valores de string. Para alterar esses valores de string em inteiros, passamos os valores de keySize , n e EorD para int () . A função readKeyFile () retorna três inteiros, int (keySize) , int (n) e int (EorD) , que você usará para criptografia ou descriptografia.

Escrevendo a criptografia em um arquivo

Na linha 104, a função encryptAndWriteToFile () chama encryptMessage () para criptografar a cadeia com a chave e cria o arquivo que contém o conteúdo criptografado.

```
104. def encryptAndWriteToFile (messageFilename, keyFilename, message,  
blockSize = Nenhum):  
105. # Usando uma chave de um arquivo de chave, criptografe a mensagem e  
salve-a em um  
arquivo 106. #. Retorna a string de mensagem criptografada.  
107. keySize, n, e = readKeyFile (keyFilename)
```

A função encryptAndWriteToFile () usa três argumentos de string: um nome de arquivo para gravar a mensagem criptografada em (messageFilename), um nome de arquivo da chave pública a ser usada (keyFilename) e uma mensagem a ser criptografada (mensagem). O parâmetro blockSize é especificado como o quarto argumento.

A primeira etapa do processo de criptografia é ler os valores de keySize , n e e do arquivo-chave chamando readKeyFile () na linha 107.

O parâmetro blockSize possui um argumento padrão de None :

108. if blockSize == None:

109. # Se blockSize não for fornecido, configure-o para o maior tamanho permitido pelo

tamanho da chave e pelo tamanho do conjunto de símbolos.

110. blockSize = int (math.log (2 ** keySize, len (SÍMBOLOS)))

Se nenhum argumento for passado para o parâmetro blockSize , o tamanho do bloco será configurado para o maior tamanho que o conjunto de símbolos e o tamanho da chave permitirão. Lembre-se de que o *tamanho do bloco de tamanho de conjunto de símbolos > tamanho da chave* da equação 2 deve ser verdadeiro. Para calcular o maior tamanho de bloco possível, a função math.log () do Python é chamada para calcular o logaritmo do *tamanho de 2 chaves* com uma base de len (SYMBOLS) na linha 110.

A matemática da cifra de chave pública funciona corretamente somente se o tamanho da chave for igual ou maior que o tamanho do bloco, portanto, é essencial verificarmos isso na linha 112 antes de continuar.

111. # Verifique se o tamanho da chave é grande o suficiente para o tamanho do bloco:

112. se não (math.log (2 ** keySize, len (SYMBOLS))> = blockSize):

113. sys.exit ('ERROR: tamanho do bloco é muito grande para o tamanho da chave e do conjunto de símbolos . Você especificou o arquivo de chave correto e o arquivo criptografado ? ')

Se keySize for muito pequeno, o programa sai com uma mensagem de erro. O usuário deve diminuir o valor passado para blockSize ou usar uma chave maior.

Agora que temos os n e e valores para a chave, chamamos a função EncryptMessage () na linha 115, que retorna uma lista de blocos inteiros.

114. # Criptografar a mensagem

115. encryptedBlocks = encryptMessage (message, (n, e), blockSize)

A função encryptMessage () espera uma tupla de dois inteiros para a chave, e é por isso que as variáveis n e e são colocadas dentro de uma tupla que é então passada como o segundo argumento para encryptMessage () .

Em seguida, convertemos os blocos criptografados em uma string que podemos gravar em um arquivo. Fazemos isso juntando os blocos em uma string com cada bloco separado por uma vírgula. Usar '''. Join (encryptedBlocks) para fazer isso não funcionará porque join () só funciona em listas com valores de string. Como o encryptedBlocks é uma lista de números inteiros, precisamos primeiro converter esses inteiros em strings:

117. # Converta os grandes valores int em um valor de string:

118. para i no intervalo (len (encryptedBlocks)):

119. encryptedBlocks [i] = str (encriptadoBlocks [i])

120. encryptedContent = '''. Join (encryptedBlocks)

O loop for na linha 118 itera através de cada índice em encryptedBlocks , substituindo o inteiro em encryptedBlocks [i] por uma forma de string do inteiro. Quando o loop for concluído, o encryptedBlocks deverá conter uma lista de valores de string em vez de uma lista de valores inteiros.

Então, podemos passar a lista de valores de string em encryptedBlocks para o método join () , que retorna as strings da lista unidas em uma única string com cada bloco separado por vírgulas. A linha 120 armazena essa string combinada na variável encryptedContent .

Também escrevemos o tamanho da mensagem e o tamanho do bloco no arquivo, além dos blocos inteiros criptografados:

122. # Grave a string criptografada no arquivo de saída:

123. encryptedContent = '% s_% s_% s%' (len (mensagem), blockSize, encryptedContent)

A linha 123 altera a variável encryptedContent para incluir o tamanho da mensagem como um inteiro, len (mensagem) , seguido por um sublinhado, o blockSize , outro sublinhado e, finalmente, os blocos inteiros criptografados (encryptedContent).

A última etapa do processo de criptografia é gravar o conteúdo no arquivo. O nome do arquivo fornecido pelo parâmetro messageFilename é criado com a chamada para open () na linha 124. Observe que, se um arquivo com esse nome já existir, o novo arquivo irá sobrescrevê-lo.

124. fo = open (messageFilename, 'w')

125. fo.write (encryptedContent)

126. fo.close ()

127. # Também retorna a string criptografada:

128. return encryptedContent

A string em encryptedContent é gravada no arquivo chamando o método write () na linha 125. Depois que o programa termina de gravar o conteúdo do arquivo, a linha 126 fecha o objeto file em fo .

Finalmente, a string em encryptedContent é retornada da função encryptAndWriteToFile () na linha 128. (Isso é para que o código que chama a função possa usar essa string para, por exemplo, imprimi-la na tela.)

Agora você sabe como a função encryptAndWriteToFile () criptografa uma sequência de mensagens e grava os resultados em um arquivo. Vamos ver como o programa usa a função readFromFileAndDecrypt () para descriptografar uma mensagem criptografada.

Descriptografando de um arquivo

Semelhante a encryptAndWriteToFile () , a função readFromFileAndDecrypt () possui parâmetros para o nome do arquivo do arquivo de mensagens criptografadas e o nome do arquivo do arquivo-chave. Certifique-se de passar o nome do arquivo da chave privada para keyFilename , não a chave pública.

131. def readFromFileAndDecrypt (messageFilename, keyFilename):

132. # Usando uma chave de um arquivo de chave, leia uma mensagem criptografada de um arquivo

133. # e então descriptografe-a. Retorna a string de mensagem descriptografada.

134. keySize, n, d = readKeyFile (keyFilename)

O primeiro passo é o mesmo que encryptAndWriteToFile () : a função readKeyFile () é chamada para obter os valores para as variáveis keySize , n e d .

O segundo passo é ler o conteúdo do arquivo. A linha 138 abre o arquivo messageFilename para leitura.

137. # Lê o tamanho da mensagem e a mensagem criptografada do arquivo:

138. fo = open (messageFilename)

139. content = fo.read ()

140. messageLength, blockSize, encryptedMessage = content.split ('_')

141 messageLength = int (messageLength)

142. blockSize = int (blockSize)

A chamada do método read () na linha 139 retorna uma string com o conteúdo

completo do arquivo, que é o que você veria se abrisse o arquivo de texto em um programa como o Notepad ou oTextEdit, copiasse todo o conteúdo e o colasse como um valor de string em seu programa.

Lembre-se de que o formato do arquivo criptografado tem três inteiros separados por sublinhados: um inteiro representando o comprimento da mensagem, um inteiro para o tamanho do bloco usado e os blocos inteiros criptografados. A linha 140 chama o método split () para retornar uma lista desses três valores, e o truque de atribuição múltipla coloca os três valores nas variáveis messageLength , blockSize e encryptedMessage , respectivamente.

Como os valores retornados por split () são strings, as linhas 141 e 142 usam int () para alterar messageLength e blockSize para seu formato inteiro, respectivamente.

A função readFromFileAndDecrypt () também verifica, na linha 145, que o tamanho do bloco é igual ou menor que o tamanho da chave.

144. # Verifique se o tamanho da chave é grande o suficiente para o tamanho do bloco:

145. se não (math.log (2 ** keySize, len (SYMBOLS))> = blockSize):

146. sys.exit ('ERROR: tamanho do bloco é muito grande para o tamanho da chave e do conjunto de símbolos . Você especificou o arquivo de chave correto e o arquivo criptografado ? ')

Essa verificação deve sempre passar, porque se o tamanho do bloco fosse muito grande, seria impossível criar o arquivo criptografado em primeiro lugar. Muito provavelmente, o arquivo de chave privada incorreto foi especificado para o parâmetro keyFilename , o que significa que a chave não teria descriptografado o arquivo corretamente de qualquer maneira.

A string encryptedMessage contém muitos blocos unidos por vírgulas, que convertemos de volta para números inteiros e armazenamos na variável encryptedBlocks .

148. # Converta a mensagem criptografada em grandes valores int:

149. encryptedBlocks = []

150. para block em encryptedMessage.split (','):

151. encryptedBlocks.append (int (block))

O loop for na linha 150 itera a lista criada a partir da chamada do método split ()

em `encryptedMessage`. Esta lista contém cadeias de caracteres individuais blocos. A forma inteira destas strings é anexada à lista `encryptedBlocks` (que era uma lista vazia na linha 149) cada vez que a linha 151 é executada. Depois que o loop for na linha 150 for concluído, a lista `encryptedBlocks` deverá conter valores inteiros dos números que estavam na cadeia `encryptedMessage`.

Na linha 154, a lista em `encryptedBlocks` é passada para a função `decryptMessage()` junto com `messageLength`, a chave privada (um valor de tupla de dois inteiros `n` e `d`) e o tamanho do bloco.

153. # Descriptografar os grandes valores int:

154. return `decryptMessage(encryptedBlocks, messageLength, (n, d), blockSize)`

A função `decryptMessage()` na linha 154 retorna um único valor de string da mensagem descriptografada, que em si é um valor retornado de `readFileAndDecrypt()`.

Chamando a função `main()`

Finalmente, as linhas 159 e 160 chamam a função `main()` se `publicKeyCipher.py` estiver sendo executado como um programa em vez de ser importado como um módulo por outro programa.

157. # Se `publicKeyCipher.py` for executado (em vez de importado como um módulo), chame

158. # a função `main()`.

159. if `__name__ == '__main__'`:

160. `main()`

Isso completa nossa discussão sobre como o programa `publicKeyCipher.py` executa a criptografia e a descriptografia usando a codificação de chave pública.

Resumo

Parabéns, você terminou o livro! Não há um capítulo “Hacking the Public Key Cipher” porque não há um simples ataque contra a criptografia de chave pública que não levaria trilhões de anos.

Neste capítulo, o algoritmo RSA foi bastante simplificado, mas ainda é uma cifra real usada em software de criptografia profissional. Quando você entra em um site ou compra algo na internet, por exemplo, cifras como essa mantêm as senhas e os números dos cartões de crédito em segredo de qualquer pessoa que possa

estar interceptando o tráfego da sua rede.

Embora a matemática básica usada para software de criptografia profissional seja a mesma descrita neste capítulo, você não deve usar este programa para proteger seus arquivos secretos. Os hacks contra um programa de criptografia como o *publicKeyCipher.py* são muito sofisticados, mas existem. (Por exemplo, como os números aleatórios `random.randint()` criados não são realmente aleatórios e podem ser previstos, um hacker pode descobrir quais números foram usados para os números primos de sua chave privada.)

Todas as cifras anteriores discutidas neste livro podem ser cortadas e tornadas inúteis. Em geral, evite escrever seu próprio código de criptografia para proteger seus segredos, porque você provavelmente cometerá erros sutis na implementação desses programas. Hackers e agências de espionagem podem explorar esses erros para hackear suas mensagens criptografadas.

Uma cifra é segura apenas se tudo, exceto a chave, puder ser revelado enquanto ainda mantém a mensagem em segredo. Você não pode confiar em um criptoanalista não ter acesso ao mesmo software de criptografia ou não saber qual código você usou. Sempre assuma que o inimigo conhece o sistema!

O software de criptografia profissional é escrito por criptógrafos que passaram anos estudando a matemática e as possíveis fraquezas de várias cifras. Mesmo assim, o software que eles escrevem é inspecionado por outros criptógrafos para verificar erros ou possíveis fraquezas. Você é perfeitamente capaz de aprender sobre esses sistemas de criptografia e matemática criptográfica. Não é sobre ser o hacker mais inteligente, mas gastar tempo para estudar e se tornar o hacker mais experiente.

Espero que este livro tenha lhe dado as fundações necessárias para se tornar um hacker e programador de elite. Há muito mais para programação e criptografia do que o que este livro cobre, então eu encorajo você a explorar e aprender mais! Eu recomendo altamente o *livro do código: A ciência do segredo do Egito antigo à criptografia quântica* por Simon Singh, que é um grande livro sobre a história geral da criptografia. Você pode acessar

<https://www.nostarch.com/crackingcodes/> para obter uma lista de outros livros e sites para saber mais sobre criptografia. Sinta-se à vontade para me enviar suas perguntas de programação ou criptografia para al@inventwithpython.com ou publicá-las em <https://reddit.com/r/inventwithpython/>.

Boa sorte!