КРИПТОГРАФИЯ И ВЗЛОМ ШИФРОВ НА PYTHON

ЭЛ СВЕЙГАРТ

no starch press

КРИПТОГРАФИЯ И ВЗЛОМ ШИФРОВ НА PYTHON

CRACKING CODES WITH PYTHON

by Al Sweigart

no starch
press

San Francisco

КРИПТОГРАФИЯ И ВЗЛОМ ШИФРОВ НА PYTHON

Эл Свейгарт

Москва • Санкт-Петербург
2020

ББК 32.973.26-018.2.75
С24
УДК 681.3.07

Компьютерное издательство "Диалектика"

Перевод с английского канд. хим. наук *А.Г. Гузикевича*

Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Свейгарт, Эл

С24 Криптография и взлом шифров на Python.: Пер. с англ. – СПб.: ООО "Диалектика", 2020. – 512 с. – Парал. тит. англ.

ISBN 978-5-907203-02-0 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Эл Свейгарт

Криптография и взлом шифров на Python

Подписано в печать 03.04.2020. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 41,3. Уч.-изд. л. 24,0
Тираж 300 экз. Заказ № 2150

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-02-0 (рус.)
ISBN 978-1-59327-822-9 (англ.)

Оглавление

Содержание

Посвящается Аарону Шварцу (1986–2013)

“Аарон был частью той армии граждан, которые верят, что настоящая демократия возможна только тогда, когда мы имеем свободный доступ к информации и хорошо осведомлены обо всех наших правах и обязанностях. Той армии, которая верит, что наше общество станет более совершенным, если справедливость и знания станут достоянием всех, а не только богачей и тех, кто дорвался до власти. Думая о нашей армии, я вижу Аарона, и мою душу охватывает скорбь. Мы утратили одного из наших лучших ангелов”.

Карл Маламуд

Об авторе

Эл Свейгарт – профессиональный разработчик, автор множества книг по программированию, включая бестселлер *Автоматизация рутинных задач с помощью Python*. Его любимый язык программирования – Python, для которого он разработал несколько модулей с открытым исходным кодом. Многие его книги доступны на условиях бесплатной лицензии Creative Commons на сайте `https://inventwithpython.com/`.

О технических рецензентах

Ари Лаценски разрабатывает мобильные приложения и модули Python.

Жан-Филипп Омассон (главы 22–24) – главный инженер швейцарской компании Kudelski Security. Регулярно выступает на конференциях по информационной безопасности, таких как Black Hat, DEF CON, Troopers и Infiltrate. Автор книги *Serious Cryptography* (No Starch Press, 2017).

ВВЕДЕНИЕ

"Мне не оставалось ничего иного, как подслушивать, ведь мы занимались прослушкой".

"Анонимус"

Если бы мы могли перенестись в начало 1990-х годов, то из-за главы 23, в которой реализуется алгоритм RSA, книгу нельзя было бы вывезти за пределы США. Поскольку сообщения, зашифрованные с помощью алгоритма RSA, не поддавались взлому, экспорт такого рода криптографических систем считался вопросом национальной безопасности и требовал специального разрешения Госдепа США. Фактически на законодательном уровне стойкие криптографические системы подчинялись тем же регуляторным нормам, что и танки, ракеты, огнеметы...

В 1990 году Даниэль Бернштейн, студент Калифорнийского университета в Беркли, захотел опубликовать академическую статью, включающую исходный код его криптосистемы Snuffle. Государственные органы США проинформировали его о том, что прежде, чем он сможет опубликовать исходный код, он должен получить лицензию на торговлю оружием. Ему также было сказано, что в случае подачи им заявки на экспортную лицензию он получит отказ ввиду высокой надежности этой технологии.

В судебном иске против Соединенных Штатов интересы Даниэля Бернштейна представляла молодая правозащитная организация — Фонд электронных рубежей (Electronic Frontier Foundation). Впервые в истории суд постановил, что на программный код распространяется действие первой поправки к Конституции США в части защиты свободы слова и что

ограничение пересылки зашифрованных сообщений законами об экспортном контроле нарушает права Бернштейна.

В наши дни на стойких криптосистемах базируется значительная часть мировой экономики. От них зависят сайты электронной торговли по всему миру, через которые ежедневно совершаются миллионы покупок. Опасения разведслужб относительно того, что зашифрованное программное обеспечение похоронит национальную безопасность, оказались беспочвенными.

Однако в начале 1990-х годов свободное распространение знаний (что и пытается делать автор книги) довело бы вас до тюрьмы за незаконную торговлю оружием.

Для кого предназначена эта книга

Обучению основам криптографии посвящено множество книг. Есть ряд книг, в которых новичков учат взламывать шифры. Но нет ни одной книги, в которой новичков учили бы писать компьютерные программы, способные взламывать шифры. Данная книга восполняет этот пробел.

Книга предназначена для тех, кто интересуется шифрованием, взломом шифров и криптографией. Все шифры, рассматриваемые в данной книге (за исключением криптосистем с открытым ключом, которым посвящены главы 23 и 24), существуют уже много столетий, и для их взлома достаточно вычислительных мощностей любого современного ноутбука. Ни одна организация и ни одно частное лицо уже не пользуется этими шифрами, однако в процессе их изучения вы освоите основы криптографии и узнаете, как хакеры взламывают слабые шифры.

Примечание

Изучение шифров, рассматриваемых в данной книге, станет увлекательным занятием, но помните, что они не обеспечивают подлинную безопасность. Не пытайтесь применять приведенные в книге программы для защиты собственных файлов. И вообще возьмите за правило не доверять шифрам, которые сами создаете. Профессиональные криптосистемы исследуются годами, прежде чем получают одобрение со стороны экспертов в области криптографии.

Книга ориентирована в том числе на новичков в программировании. Здесь излагаются основы программирования на Python — одного из лучших языков для начинающих. Python характеризуется настолько плавной кривой обучения, что освоить его смогут новички любого возраста, а его возможности удовлетворяют запросам даже самых требовательных про-

фессионалов. Python выполняется в среде Windows, macOS, Linux и даже Raspberry Pi, причем он доступен для свободной загрузки и использования (соответствующие указания будут приведены далее).

В литературе по криптографии часто употребляется термин *хакер*. Это слово имеет двоякий смысл. Оно может означать лицо, которое пытается досконально изучить систему (например, пошаговую процедуру шифрования или программный код) для того, чтобы преодолеть ее ограничения и творчески ее модифицировать. В то же время слово *хакер* также может означать преступника, который взламывает компьютерные системы и получает доступ к конфиденциальной информации, нанося ущерб. В данной книге, говоря о хакерах и взломе шифров, мы придерживаемся первого определения. Быть хакером — это круто! А вот преступники — это те, кто самоутверждаются за счет других.

Структура книги

В первых нескольких главах читатели познакомятся с основами Python и азами криптографии. В последующих главах поочередно объясняется, как запрограммировать тот или иной шифр и как его взломать. Кроме того, каждая глава завершается контрольными вопросами, которые помогут проверить, насколько хорошо вы усвоили прочитанный материал.

- **Глава 1 "Инструменты «бумажной» криптографии".** Обсуждаются простейшие инструменты "бумажных" криптотехнологий и демонстрируются способы шифрования, существовавшие в докомпьютерную эпоху.
- **Глава 2 "Программирование в интерактивной оболочке".** Объясняется, как применять интерактивную оболочку для построчного выполнения кода Python.
- **Глава 3 "Строковый тип данных и написание программ".** Рассказывается о том, как писать программы, и вводится строковый тип данных, используемый во всех программах в книге.
- **Глава 4 "Обратный шифр".** Показано, как написать простую программу шифрования.
- **Глава 5 "Шифр Цезаря".** Описывается простейший шифр, известный человечеству уже несколько тысяч лет.
- **Глава 6 "Взлом шифра Цезаря методом грубой силы".** Демонстрируется, как можно дешифровать сообщение, не имея ключа шифрования.

- **Глава 7 "Шифрование с помощью перестановочного шифра".** Рассматривается программа, шифрующая сообщения с помощью перестановочного шифра.
- **Глава 8 "Дешифрование перестановочного шифра".** Продолжение темы перестановочного шифра, только на этот раз демонстрируется процедура дешифрования сообщений.
- **Глава 9 "Написание тестов".** Введение в методики тестирования программ.
- **Глава 10 "Шифрование и дешифрование файлов".** Объясняется, как писать программы, осуществляющие чтение и запись файлов.
- **Глава 11 "Программное распознавание английских слов".** Описывается, как с помощью компьютера распознавать предложения на английском языке.
- **Глава 12 "Взлом перестановочного шифра".** Применяем полученные в предыдущих главах знания для взлома перестановочного шифра.
- **Глава 13 "Аффинное шифрование с помощью модульной арифметики".** Рассматриваются математические основы аффинного шифрования.
- **Глава 14 "Программирование аффинного шифра".** Демонстрируется написание программы аффинного шифрования.
- **Глава 15 "Взлом аффинного шифра".** Объясняется, как написать программу для взлома аффинного шифра.
- **Глава 16 "Программирование простого подстановочного шифра".** Рассматривается программа, реализующая шифрование на основе подстановочного шифра.
- **Глава 17 "Взлом простого подстановочного шифра".** Объясняется, как написать программу для взлома простого подстановочного шифра.
- **Глава 18 "Программирование шифра Виженера".** Описывается более сложный вариант подстановочного шифра — шифр Виженера.
- **Глава 19 "Частотный анализ".** Исследуется частотная структура английских слов и демонстрируется, как применить эту информацию для взлома шифра Виженера.
- **Глава 20 "Взлом шифра Виженера".** Рассматривается программа для взлома шифра Виженера.
- **Глава 21 "Одноразовый шифроблокнот".** Обсуждается концепция одноразовых шифроблокнотов и приводится математическое обоснование того, почему такой блокнот невозможно взломать.

- **Глава 22 "Нахождение и генерирование простых чисел".** Рассказывается, как написать программу для быстрого определения того, является ли число простым.
- **Глава 23 "Генерирование ключей для криптосистем с открытым ключом".** Обсуждаются принципы шифрования с открытым ключом и рассматривается программа, генерирующая открытый и закрытый ключи.
- **Глава 24 "Программирование шифров с открытым ключом".** Объясняется, как написать программу шифрования с открытым ключом, который невозможно взломать обычными методами.
- **Приложение А "Отладка кода Python".** Демонстрируется, как находить и устранять ошибки в программном коде с помощью отладчика IDLE.
- **Приложение Б "Ответы на контрольные вопросы".** Содержит ответы на упражнения к главам.

Как пользоваться книгой

Данная книга отличается от остальных книг по программированию тем, что основное внимание в ней уделено исходному коду готовых программ. Вместо того чтобы обучать читателей основам программирования, предлагая им самостоятельно понять, как решать конкретные задачи, автор демонстрирует полноценные программные решения и подробно объясняет, как они работают.

Главы желательно читать по порядку, так как концепции программирования объясняются на основе материала предыдущих глав. Однако язык Python настолько прост для изучения, что уже после прочтения нескольких начальных глав вы, вероятно, сможете перейти к любой из последующих глав и легко понять, как работает приведенный в ней программный код. Если же почувствуете, что нить изложения ускользает от вас, вы всегда сможете вернуться к предыдущим главам.

Ввод исходного кода

Читателям настоятельно рекомендуется *вводить весь исходный код вручную*, так как это определенно поможет лучше понять его.

Вводить начальные номера строк не нужно. Эти номера не являются частью самих программ и служат исключительно для того, чтобы можно было ссылаться на ту или иную строку в тексте книги. А вот собственно программный код, следующий за номерами строк, необходимо вводить

именно в таком виде, в каком он приведен в книге, с учетом регистра букв.

Кроме того, обратите внимание на то, что некоторые строки кода выделяются отступами длиной четыре, восемь или более пробелов. Во избежание ошибок следите за строгим соблюдением количества пробелов в начале каждой строки.

Если не хотите самостоятельно вводить исходный код программ, загрузите его на сайте издательства:

```
https://nostarch.com/crackingcodes
```

Исходные файлы доступны также на сайте издательства "Диалектика" (в этих файлах исправлен ряд авторских ошибок):

```
http://www.williamspublishing.com/Books/978-5-907203-02-0.html
```

Исправление опечаток

Самостоятельный ввод исходного кода программ действительно помогает в процессе изучения Python, но это чревато появлением опечаток, которые не всегда легко обнаружить, особенно если программа достаточно большая.

Чтобы облегчить и ускорить поиск ошибок во введенном вами исходном коде, скопируйте и вставьте текст программы в окно онлайн-утилиты `Diff`, доступной по следующим адресам:

```
https://inventwithpython.com/cracking/diff/
https://inventwithpython.com/hacking/diff/
```

В окне утилиты выделяются любые различия между исходным кодом, приведенным в книге, и кодом, набранным вами.

Правила оформления листингов, принятые в книге

Книга задумана не как справочник, а как пособие для начинающих, поэтому стиль оформления листингов иногда может не соответствовать устоявшейся практике. Это было сделано сознательно, чтобы облегчить изучение программ.

Опытные программисты наверняка смогут предложить немало изменений для повышения производительности кода. Но не забывайте: цель автора заключалась в том, чтобы получить работающую программу с наименьшими усилиями.

Ресурсы в Интернете

На сайте издательства вы найдете множество полезных ресурсов, включая все необходимые программные файлы, а также дополнительные ссылки:

`https://nostarch.com/crackingcodes/`

Загрузка и установка Python

Прежде чем заняться программированием, необходимо установить интерпретатор Python — программу, которая будет выполнять написанные вами инструкции. В дальнейшем при упоминании о Python мы будем подразумевать интерпретатор Python.

Версии Python для Windows, macOS и Linux/UNIX можно бесплатно загрузить по адресу `https://www.python.org/downloads/`. Все приведенные в книге программы должны без проблем выполняться в последней версии Python.

Примечание

Загружайте Python версии 3 (например, 3.6 или 3.8). Приведенные в книге программы рассчитаны именно на эту версию. В случае использования Python 2 они могут выполняться некорректно или вообще не работать.

Windows

В Windows загрузите установщик Python, файл которого должен иметь расширение *.msi*, и выполните двойной щелчок на нем. Установите Python в соответствии с приведенной ниже процедурой, следуя инструкциям, которые будут появляться в окне программы установки.

1. Выберите пункт Install Now, чтобы начать установку.
2. По завершении инсталляции щелкните на кнопке Close.

macOS

В macOS загрузите файл *.dmg* и дважды щелкните на нем. Установите Python в соответствии с приведенной ниже процедурой, следуя инструкциям, которые будут появляться в окне программы установки.

1. Когда DMG-пакет откроется в новом окне, выполните двойной щелчок на файле *Python.mpkg*. Возможно, вам придется ввести пароль администратора.
2. Щелкайте на кнопках Continue в разделе Welcome, а затем щелкните на кнопке Agree, чтобы принять условия лицензии.

3. Выберите HD Macintosh (или имя своего жесткого диска) и щелкните на кнопке Install.

Загрузка файла *pyperclip.py*

Почти все приведенные в книге программы импортируют авторский модуль *pyperclip.py*. Он содержит функции, позволяющие выполнять операции копирования и вставки шифротекстов с использованием буфера обмена. Модуль не входит в комплект поставки Python, поэтому его необходимо загрузить по следующему адресу:

```
https://inventwithpython.com/pyperclip.py
```

Файл модуля должен находиться в той же папке, что и файлы других программ. Если это не так, то при попытке запустить программу появится следующее сообщение об ошибке:

```
ImportError: No module named pyperclip
```

Теперь, когда вы знаете, как загрузить и установить интерпретатор Python и модуль *pyperclip.py*, познакомимся со средой, в которой нам предстоит писать программы.

Запуск IDLE

Интерпретатор Python — это специальная программа, предназначенная для запуска пользовательских приложений. Для написания программ требуется *интегрированная среда разработки* (Integrated Development and Learning Environment — IDLE), во многом напоминающая текстовый процессор. Она устанавливается вместе с Python. Для запуска IDLE необходимо выполнить следующие действия.

- Если вы работаете в Windows 7 или выше, щелкните на значке Пуск в левом нижнем углу экрана, введите **`IDLE`** в поле поиска и выберите в списке элемент IDLE (Python 3.6 64-bit).
- Если вы работаете в macOS, откройте окно программы Finder, щелкните на пункте меню Applications, затем — на элементе Python 3.6 и далее — на значке IDLE.
- Если вы работаете в Ubuntu, выберите пункты меню Applications⇨ Accessories⇨Terminal, а затем введите **`idle3`**.

Какую бы операционную систему вы ни использовали, окно IDLE будет выглядеть примерно одинаково (рис. 1), разве что текст заголовка

окна может несколько отличаться в зависимости от используемой версии Python.

Это окно называется *интерактивной оболочкой*. Оболочка представляет собой программу, которая позволяет непосредственно вводить инструкции подобно тому, как это делается в окне командной строки Windows. Вводимые здесь инструкции будут сразу же выполняться интерпретатором Python.

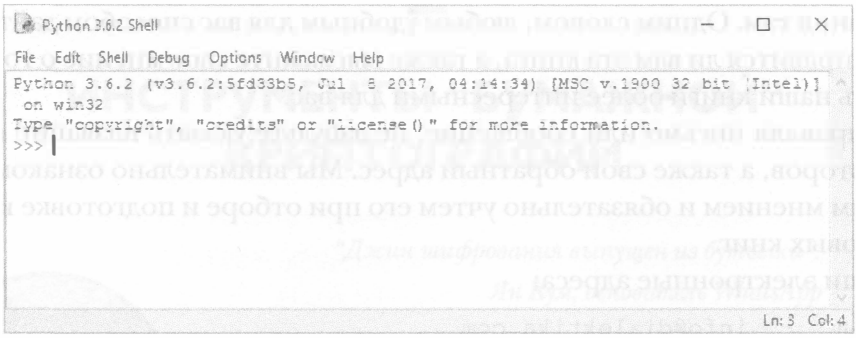


Рис. 1. *Окно IDLE*

Например, перейдите в окно интерактивной оболочки и введите сразу вслед за приглашением >>> следующую команду:

```
>>> print('Hello, world!')
```

Нажмите клавишу <Enter>, и интерактивная оболочка немедленно отреагирует на это, выдав следующий текст:

```
Hello, world!
```

Резюме

В былые времена взламывать шифры приходилось с помощью карандаша и бумаги. И хотя с появлением компьютеров многие классические шифры стали уязвимыми для атак, их изучение по-прежнему представляет интерес. Кроме того, написание программ криптоанализа, способных взламывать подобные шифры, — отличный способ научиться программировать.

В главе 1 вы познакомитесь с простейшими криптографическими инструментами, позволяющими шифровать и дешифровать сообщения, не прибегая к помощи компьютеров.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

1

ИНСТРУМЕНТЫ "БУМАЖНОЙ" КРИПТОГРАФИИ

"Джин шифрования выпущен из бутылки".
Ян Кум, основатель WhatsApp

Прежде чем приступать к написанию шифровальных программ, давайте рассмотрим, как реализуется процесс шифрования и дешифрования с помощью карандаша и бумаги. Это позволит вам лучше понять природу шифров и тех математических инструментов, которые применяются для создания секретных сообщений. В данной главе вы узнаете о том, что такое криптография и чем коды отличаются от шифров. Затем мы воспользуемся простым шифром, известным как *шифр Цезаря*, для шифрования и расшифровки бумажных сообщений.

В этой главе...

- Что такое криптография
- Коды и шифры
- Шифр Цезаря
- Шифровальные диски
- Криптография и арифметика
- Двойное шифрование

Что такое криптография

С незапамятных времен все, кому требовалось обмениваться тайными сведениями, например шпионы, военные, пираты, торговцы и дипломаты, прибегали к засекречиванию своих сообщений, чтобы тайны оставались надежно скрыты от посторонних глаз. *Криптография* — это наука, которая изучает способы создания и применения секретных кодов. Чтобы понять, как работают криптографические методы, рассмотрим следующие два фрагмента текста.

```
nyr N.vNwz5uNz5Ns6620Nz0N3z2v
N yvNwz9vNz5N6!9Nyvr9
y0QNnvNwv tyNz
Nw964N6!9N5vzxys690,N.vN2z5u-
3vNz Nr Ny64v,N.vNt644!5ztr vNz
N 6N6 yv90,Nr5uNz Nsvt64v0N
yvN7967v9 BN6wNr33Q N-m63 rz9v
```

```
!NN2 Nuwv,N9,vNN!vNrBN3zyN4vN
N6 Qvv0z6nvN.7N0yv4N 4 zzvNN
vyN,NN99z0zz6wz0y3vv26 9
w296vyNNrrNyQst.560N94Nu5y
rN5nz5vv5t6v63zNr5.
N75sz6966NNvw6 zu0 wtNxs6t
49NrN3Ny9Nvzy!
```

Текст слева — секретное сообщение, которое было *зашифровано*, т.е. преобразовано в секретный код. Любому человеку, не знающему способа его *дешифрования*, или *расшифровки*, оно кажется полной абракадаброй. Сообщение справа — случайный набор символов, не имеющий никакого скрытого смысла. Шифрование позволяет сохранить смысл сообщения в тайне от тех, кому не известен способ его расшифровки, даже если сообщение попадет им в руки. *Шифрованное сообщение воспринимается посторонними как случайный набор букв, не несущий в себе никакого смысла.*

Криптограф, или *шифровальщик*, использует и изучает секретные коды. Разумеется, секретные сообщения не всегда остаются секретными. *Криптоаналитик*, т.е. *взломщик кодов*, или *хакер*, способен взламывать и читать сообщения, зашифрованные другими людьми. Цель книги — научить вас зашифровывать и расшифровывать сообщения с помощью различных методов. К счастью, все те методы взлома, которые вы изучите, не настолько опасны, чтобы у вас из-за них могли возникнуть проблемы с законом.

Коды и шифры

В отличие от шифров *коды* изначально создаются такими, чтобы они были понятны и общедоступны. Коды заменяют буквы символами, которые любой человек может использовать для перевода в форму сообщения.

В начале XIX века развитие электрического телеграфа привело к созданию известного кода, обеспечивавшего почти мгновенный обмен сообщениями между континентами по проводам. Сообщения, отправляемые по телеграфу, достигали своих адресатов гораздо быстрее, чем прежняя лошадиная почта, перевозившая мешки с письмами. Однако телеграф не позволял отправлять сообщения в том же виде, в каком они были написаны на бумаге, т.е. в виде последовательностей букв. По нему могли пересылаться только два типа электрических импульсов: короткий, который назвали "точка", и длинный, который назвали "тире".

Для преобразования букв алфавита в электрические импульсы необходимо располагать системой кодов, с помощью которой можно было бы переводить привычные для нас буквы на язык точек и тире. Процесс преобразования букв алфавита в последовательности точек и тире для отправки по телеграфу называется *кодированием*, а обратный процесс преобразования электрических импульсов в буквы при получении сообщения — *декодированием*. Способ, применяемый для кодирования и декодирования телеграфных сообщений (а впоследствии и сообщений, передаваемых по радио), был изобретен Сэмюэлем Морзе и Альфредом Вейлем и получил название *код Морзе*, или *азбука Морзе* (табл. 1.1).

Таблица 1.1. Международная азбука Морзе

Латинский символ	Код	Латинский символ	Код	Цифра	Код
A	•–	N	–•	1	•————
B	–•••	O	———	2	••———
C	–•–•	O	•——•	3	•••——
D	–••	Q	——•–	4	••••–
E	•	R	•–•	5	•••••
F	••–•	S	•••	6	–••••
G	——•	T	–	7	——•••
H	••••	U	••–	8	———••
I	••	V	•••–	9	————•
J	•———	W	•——	0	—————
K	–•–	X	–••–		
L	•–••	Y	–•——		
M	——	Z	——••		

Используя телеграфный ключ для передачи точек и тире, телеграфист отправлял текстовые сообщения, способные почти мгновенно достигать адресата, находящегося на другом конце земного шара[1]!

В отличие от знакового кодирования *шифр* — специфический тип кодов, обеспечивающих сохранение содержания сообщения в тайне. Шифр можно использовать для того, чтобы преобразовать исходный текст, написанный на понятном языке (так называемый *простой текст*), в бессвязный набор символов, называемый *шифротекстом*, который скрывает смысл секретного сообщения. Шифр — это набор правил преобразования между простым и зашифрованным текстом. В правилах для шифрования и дешифрования часто используется один и тот же ключ, который называется *секретным* и известен только отправителю и получателю сообщения. В книге вы изучите несколько видов шифров и напишете программы для шифрования и дешифрования текста с их помощью. Но сначала следует научиться шифровать сообщения вручную, используя простые средства "бумажной" криптографии.

Шифр Цезаря

Первый из шифров, который мы изучим, — шифр Цезаря, названный так в честь Юлия Цезаря, который пользовался им 2000 лет тому назад. Хорошая новость состоит в том, что он прост и несложен для изучения. Но есть и плохая новость: в силу простоты этого шифра криптоаналитику не составит большого труда взломать его. Тем не менее ознакомление с ним даст вам полезный опыт.

Шифр Цезаря основан на замене одной буквы другой после предварительного смещения всего алфавита на определенное число позиций. Юлий Цезарь заменял буквы в своих сообщениях путем смещения алфавита на три позиции и последующей замены каждой буквы соответствующей буквой из смещенного алфавита.

Например, вместо каждой буквы 'A' он подставлял букву 'D', вместо каждой буквы 'B' — букву 'E' и т.д. Если Цезарю нужно было сдвинуть букву, находящуюся в конце алфавита, скажем, 'Y', то он возвращался в начало алфавита, смещаясь в целом на три позиции и подставляя букву 'B'. В этом разделе мы будем шифровать сообщения вручную, применяя шифр Цезаря.

[1] Более подробную информацию об азбуке Морзе можно найти в Википедии: `https://ru.wikipedia.org/wiki/Азбука_Морзе`.

Шифровальный диск

Чтобы упростить преобразование простого текста в зашифрованный с помощью шифра Цезаря, мы будем использовать *шифровальный диск*. Он состоит из двух колец, каждое из которых разбито на 26 ячеек (по числу букв английского алфавита). Внешнее кольцо представляет алфавит исходного текста, а внутреннее — соответствующие буквы зашифрованного текста. Буквы на внутреннем кольце пронумерованы числами от 0 до 25. Эти числа определяют ключ шифрования, в данном случае — количество позиций, на которое нужно перейти от буквы 'A' к соответствующей букве на внутреннем кольце. Поскольку сдвиг выполняется по кругу, смещение с ключом, значение которого превышает 25, продолжается с начала алфавита, соответственно, смещение на 26 позиций равносильно отсутствию сдвига, смещение на 27 позиций — сдвигу на 1 позицию и т.д.

Виртуальный шифровальный диск доступен по адресу `https://inventwithpython.com/cipherwheel/` (рис. 1.1). Чтобы провернуть диск, щелкните на нем один раз и перемещайте указатель мыши по кругу, пока не будет достигнута нужная конфигурация. Повторный щелчок останавливает дальнейшее вращение диска.

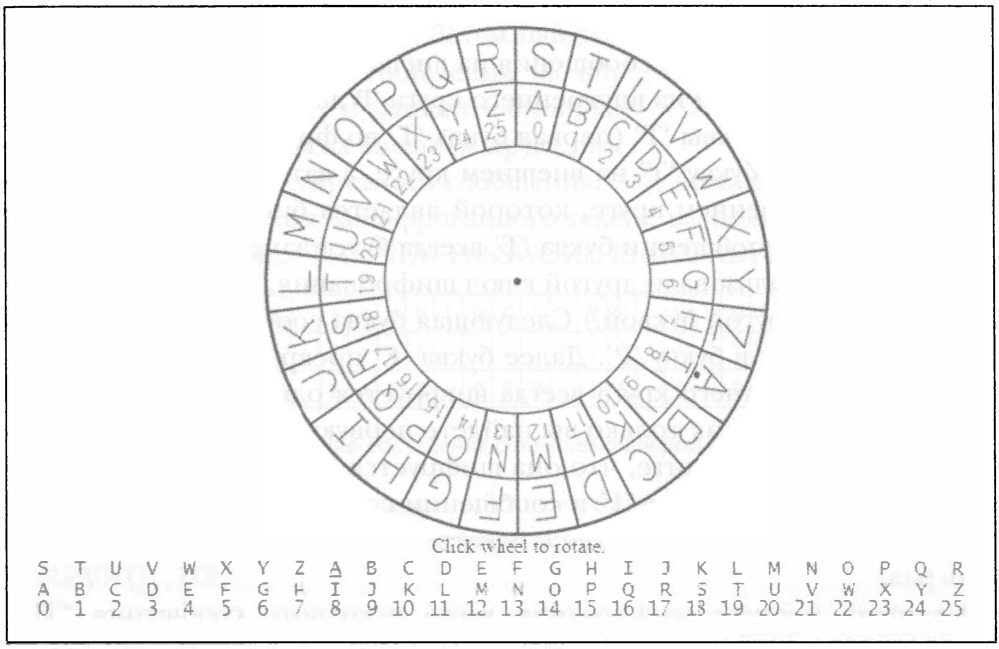


Рис. 1.1. *Онлайновый шифровальный диск*

Вы сможете получить бумажную версию шифровального диска, распечатав изображение, доступное по указанному адресу. Вырежьте два круга,

вложите меньший из них в больший и закрепите по центру булавкой, чтобы их можно было вращать.

Используя либо бумажную, либо виртуальную модель, вы сможете вручную зашифровать секретное сообщение.

Шифрование сообщения с помощью шифровального диска

В качестве примера зашифруем сообщение "THE SECRET PASSWORD IS ROSEBUD" с помощью онлайнового шифровального диска. Проверните диск так, чтобы деления внутреннего и внешнего кругов совпадали. Обратите внимание на изображение точки под буквой 'A'. Число на внутреннем круге под этой точкой и есть ключ шифрования, который будет применяться при данной конфигурации диска.

На рис. 1.1 таким числом является 8. Мы используем его в качестве ключа для того, чтобы зашифровать сообщение (рис. 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

Рис. 1.2. *Шифрование сообщения с помощью шифра Цезаря по ключу 8*

Найдите каждую букву сообщения на внешнем круге и замените ее соответствующей буквой из внутреннего круга. В данном примере сообщение начинается с буквы 'T' (первая буква 'T' во фразе "THE SECRET..."), поэтому находим букву 'T' на внешнем круге, а затем соответствующую ей букву на внутреннем круге, которой является буква 'B'. Таким образом, в исходном сообщении буква 'T' всегда будет заменяться буквой 'B'. (Если бы вы использовали другой ключ шифрования, то каждая буква 'T' заменялась бы другой буквой.) Следующая буква сообщения – 'H', которая превращается в букву 'P'. Далее буква 'E' превращается в букву 'M'. Каждая буква внешнего круга всегда шифруется одной и той же буквой внутреннего круга. Как только вы найдете первую букву 'T' в сообщении "THE SECRET..." и увидите, что она шифруется буквой 'B', можете заменить буквой 'B' все буквы 'T' в сообщении с целью экономии времени, чтобы каждую букву приходилось искать на шифровальном диске всего лишь раз.

Завершив процесс шифрования всего исходного сообщения "THE SECRET PASSWORD IS ROSEBUD", вы получите результирующее зашифрованное сообщение "BPM AMKZMB XIAAEWZL QA ZWAMJCL". Обратите внимание на то, что небуквенные символы, в данном случае пробелы, остаются неизменными.

Теперь вы сможете спокойно отправить зашифрованное сообщение нужному получателю (или оставить его при себе), и никто не сможет прочитать его, если вы не сообщите ему ключ шифрования. Убедитесь, что ключ хранится в надежном месте, поскольку любой человек, которому станет известно, что вы использовали ключ шифрования 8, сможет прочитать зашифрованное сообщение.

Дешифрование сообщения с помощью шифровального диска

Чтобы дешифровать зашифрованный текст, начните с внутреннего круга шифровального диска. Предположим, вы получили зашифрованный текст "IWT CTL EPHHLDGS XH HLDGSUXHW". Вы не сможете расшифровать его, пока не узнаете ключ (если только вы не криптоаналитик). К счастью, отправитель уже сообщил вам, что в своих сообщениях он использует ключ 15. Конфигурация шифровального диска для этого ключа приведена на рис. 1.3.

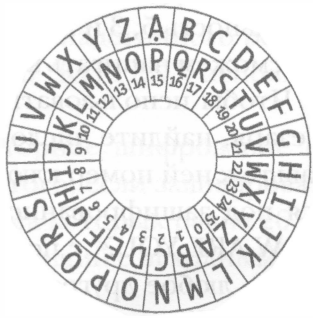


Рис. 1.3. *Конфигурация шифровального диска для ключа 15*

Установите букву 'A' на внешнем круге (помечена точкой) напротив буквы на внутреннем круге с номером 15 (буква 'P'). Затем найдите первую букву секретного сообщения на внутреннем круге (буква 'I') и запишите букву, которая соответствует ей на внешнем круге (буква 'T'). Вторая буква секретного сообщения, 'W', дешифруется в букву 'H'. Расшифровав все буквы зашифрованного текста, вы получите сообщение в виде простого текста: "THE NEW PASSWORD IS SWORDFISH" (рис. 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

Рис. 1.4. *Дешифрование сообщения с помощью шифра Цезаря по ключу 15*

Если использовать некорректный ключ, например 16, то расшифрованное сообщение превратится в бессмысленный набор символов "SGD MDV OZRRVNQC HR".

Шифрование и дешифрование средствами арифметики

Шифровальный диск — удобный инструмент для применения шифра Цезаря, но аналогичные операции можно выполнять и в арифметическом

виде. Запишите буквы алфавита от 'A' до 'Z' и пронумеруйте их числами от 0 до 25. Начните с нуля под 'A', единицей под 'B' и т.д., пока не поставите номер 25 под 'Z' (рис. 1.5).

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

Рис. 1.5. *Нумерация алфавита числами от 0 до 25*

Полученную таблицу перекодировки букв в цифры можно использовать для представления букв, что позволяет выполнять над буквами арифметические операции. Например, если вы запишете слово "CAT" цифрами 2, 0 и 19, то сможете прибавить к каждой из них 3, получив в результате последовательность 5, 3 и 22. Эти новые числа представляют буквы "FDW". Позже мы сможем написать компьютерную программу, которая все сделает за нас.

Чтобы использовать арифметику для шифрования с помощью шифра Цезаря, найдите число под буквой, которую хотите зашифровать, и прибавьте к ней номер ключа. Результирующая сумма и есть число, находящееся под зашифрованной буквой. Зашифруем, например, текст "HELLO. HOW ARE YOU?" с помощью ключа 13. (В качестве ключа можно использовать любое другое число от 1 до 25.) Прежде всего найдите число под буквой 'H', т.е. 7. Затем прибавьте 13 к этому числу: 7 + 13 = 20. Поскольку число 20 находится под буквой 'U', то буква 'H' превращается в букву 'U'.

Точно так же, чтобы зашифровать букву 'E' (4), выполните сложение 4 + 13 = 17. Буква над 17 — 'R', поэтому 'E' преобразуется в 'R' и т.д.

Этот процесс отлично работает до тех пор, пока мы не достигнем буквы 'O'. Число под 'O' — 14. Но 14 + 13 = 27, а список чисел доходит лишь до значения 25. Если сумма чисел буквы и ключа равна или превышает 26, придется вычесть из нее 26. В данном случае 27 – 26 = 1. Буква над цифрой 1 — 'B', поэтому, если используется ключ 13, буква 'O' превращается в 'B'. Зашифровав подобным образом каждую букву сообщения, получим зашифрованный текст "URYYB. UBJ NER LBH?".

Чтобы расшифровать зашифрованный текст, следует вместо прибавления ключа к каждой букве использовать вычитание. Букве 'B' в зашифрованном тексте соответствует число 1. Результатом вычитания 13 из 1 будет отрицательное число –12. Аналогично нашему правилу "вычитания 26", если результат меньше нуля при дешифровании, то к нему нужно прибавить 26. Поскольку –12 + 26 = 14, буква 'B' в зашифрованном тексте дешифруется в букву 'O'.

Как видите, чтобы воспользоваться шифром Цезаря, вовсе не обязательно иметь шифровальный диск. Все, что нам для этого нужно, — карандаш, бумага и минимальные знания арифметики!

Почему не работает двойное шифрование

У кого-то из читателей может возникнуть мысль, что использование двух разных ключей шифрования подряд позволит вдвое увеличить стойкость шифра. Однако в случае шифра Цезаря (и большинства других шифров) это не так. В действительности результат двойного шифрования будет эквивалентен тому, который вы могли бы получить обычным способом с помощью единственного ключа. Попробуем применить двойное шифрование, чтобы понять, почему так происходит.

Если вы зашифруете слово "KITTEN" с помощью ключа 3, то будете прибавлять 3 к кодам букв простого текста, и результирующим текстом будет "NLWWHQ". Если после этого зашифровать слово "NLWWHQ", только теперь с ключом 4, то получим слово "RPAALU". Но аналогичного результата можно достичь, если сразу зашифровать слово "KITTEN", используя ключ 7.

Для большинства шифров многократное повторное шифрование не приводит к повышению стойкости шифра. Более того, если зашифровать простой текст с использованием двух ключей, сумма которых равна 26, то получим зашифрованный текст, полностью совпадающий с исходным!

Резюме

Шифр Цезаря и другие подобные ему шифры не одно столетие применялись для шифрования секретной информации. Но если вы захотите вручную зашифровать длинное сообщение, например целую книгу, то на это у вас может уйти несколько дней или недель. Тут нам на помощь и приходит программирование. С шифрованием и дешифрованием больших объемов текста компьютер способен справиться менее чем за секунду!

Чтобы использовать компьютер для шифрования информации, следует освоить понятный ему язык и научиться писать программы, т.е. наборы инструкций, следуя которым компьютер будет делать то же самое, что сделали бы мы. К счастью, изучить язык программирования наподобие Python гораздо легче, чем, скажем, японский или испанский. Что касается знания математики, то от вас потребуется лишь умение выполнять операции сложения, вычитания и умножения.

В следующей главе вы узнаете о том, как применять интерактивную оболочку Python для построчного изучения программного кода.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Зашифруйте следующие фразы из книги Амброза Бирса "Словарь Сатаны" ("The Devil's Dictionary"), используя указанные ключи.
   А. "AMBIDEXTROUS: ABLE TO PICK WITH EQUAL SKILL A RIGHT-HAND POCKET OR A LEFT." (ключ 4).
   Б. "GUILLOTINE: A MACHINE WHICH MAKES A FRENCHMAN SHRUG HIS SHOULDERS WITH GOOD REASON." (ключ 17).
   В. "IMPIETY: YOUR IRREVERENCE TOWARD MY DEITY." (ключ 21).
2. Дешифруйте следующие зашифрованные фрагменты текста, используя указанные ключи.
   А. "ZXAI: P RDHIJBT HDBTIXBTH LDGCQN HRDIRWBTC XC PBTGXRP PCS PBTGXRPCHXC HRDIAPCS." (ключ 15).
   Б. "MQTSWXSV: E VMZEP EWTMVERX XSTYFPMG LSRSVW." (ключ 4).
3. Зашифруйте следующее предложение, используя ключ 0:
   "THIS IS A SILLY EXAMPLE."
4. Ниже приведены пары исходных слов и их зашифрованных версий. Какие ключи использовались в каждом случае?
   А. ROSEBUD — LIMYVOX
   Б. YAMAMOTO — PRDRDFKF
   В. ASTRONOMY — HZAYVUVTF
5. Как будет выглядеть приведенное ниже предложение, зашифрованное с помощью ключа 8, если дешифровать его с помощью ключа 9?
   "UMMSVMAA: CVKWUUWV XIBQMVKM QV XTIVVQVO I ZMDMVOMBPIB QA EWZBP EPQTM."

2

ПРОГРАММИРОВАНИЕ В ИНТЕРАКТИВНОЙ ОБОЛОЧКЕ

"Аналитическая машина Бэббиджа не претендует ни на какие открытия. Она может делать лишь то, что мы способны приказать ей исполнять".

Ада Лавлейс, октябрь 1842 г.

Прежде чем приступать к написанию собственных программ шифрования, следует ознакомиться с основами программирования, включая такие понятия, как значения, операторы, выражения и переменные.

В этой главе...

- Операторы
- Значения
- Целые и вещественные числа
- Выражения
- Хранение значений в переменных
- Изменение значений переменных

Для начала мы попрактикуемся в выполнении простых математических операций в интерактивной оболочке Python. Рекомендую читать книгу, сидя за компьютером, чтобы вы могли вводить короткие фрагменты кода и видеть результаты их работы. Мышечная память, формируемая в результате ручного ввода кода, поможет запомнить, как создается код на Python.

Простые арифметические выражения

Для начала откройте IDLE (см. введение). На экране появится окно интерактивной оболочки с подсказкой >>> и мерцающим курсором рядом с ней. Введите **2 + 2** и нажмите клавишу <Enter> (в некоторых системах это может быть клавиша <Return>). Компьютер отреагирует на это выводом на экран числа 4 (рис. 2.1).

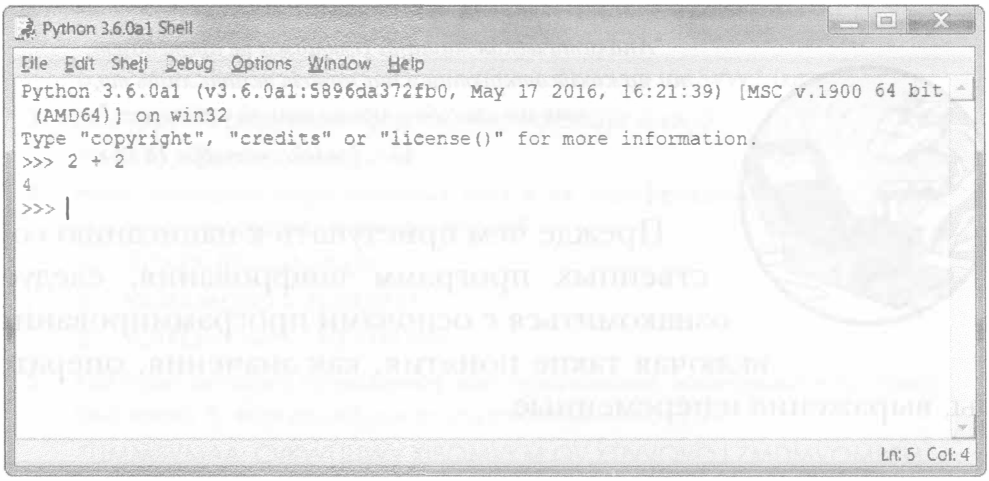


***Рис. 2.1.** Введите **2 + 2** в интерактивной оболочке*

В примере на рис. 2.1 знак + сообщает компьютеру о том, что к числу 2 нужно прибавить число 2. Python умеет выполнять и другие арифметические операции, такие как вычитание чисел (-), умножение (*) и (/). Когда символы +, -, * и / встречаются в таком контексте, они называются *операторами*, поскольку приказывают компьютеру выполнить ту или иную операцию над числами, между которыми стоят. Арифметические операторы Python приведены в табл. 2.1. Числа, окружающие знаки операций, называются *значениями* или *операндами*.

Таблица 2.1. Арифметические операторы Python

Оператор	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление

Сама по себе строка `2 + 2` не является программой, это всего лишь одиночная инструкция. Программа образуется множеством подобных инструкций.

Целые и вещественные числа

В программировании такие значения, как `4`, `0` и `99`, называют *целыми числами*. Значения с десятичной точкой (`3.5`, `42.1` и `5.0`) называют *вещественными числами* или *числами с плавающей точкой*. В Python число `5` — целое, но если записать его как `5.0`, то оно станет вещественным.

Целые и вещественные числа — это *типы данных*. Значение `42` — это целочисленный тип данных (`int`). Значение `7.5` — это вещественный тип данных, которому соответствуют числа с плавающей точкой (`float`).

Любое значение относится к тому или иному типу данных. Обычно для определения типа данных числа достаточно взглянуть на то, как оно записано. Числа без десятичной точки относятся к типу `int`, а числа, содержащие десятичную точку, — к типу `float`. Поэтому `42` — это тип `int`, тогда как `42.0` — тип `float`. Позже вы узнаете и о других типах данных (таких, как строки, о которых пойдет речь в главе 3).

Выражения

Вы уже видели один из примеров того, как в Python решаются математические задачи, но возможности языка в этом отношении гораздо шире. Введите в интерактивной оболочке следующие математические команды, каждый раз завершая ввод нажатием клавиши <Enter> (рис. 2.2).

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

Рис. 2.2. *Выражение состоит из значений (2) и операторов (+).*

Такие команды называются *выражениями*. Компьютеры способны решать миллионы подобных задач в секунду. Выражения состоят из значений (чисел), объединенных операторами (математическими знаками). Выражение может включать столько угодно значений (❶), если они объединяются операторами, причем в одном выражении можно использовать операторы разных типов (❷). Между значениями и операторами можно вставить любое количество пробелов (❸). Но следите за тем, чтобы выражения начинались у левого края строки без предшествующих им пробелов, поскольку отступ строки кода влияет на то, как Python будет интерпретировать инструкции. Подробнее об использовании пробелов в начале строк вы узнаете в разделе "Блоки" главы 4.

Порядок операций

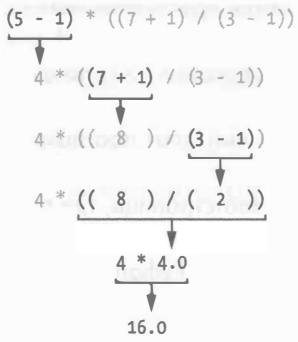
Возможно, вы помните из курса школьной математики, что *порядок операций* определяет очередность выполнения операций в пределах выражения. Например, умножение предшествует сложению. Результат вычисления выражения `2 + 4 * 3` равен `14`, поскольку сначала выполняется умножение `4 * 3`, а затем к полученному результату прибавляется `2`. Очередность операций можно изменить, используя круглые скобки. В выражении `(2 + 4) * 3` сначала выполняется сложение `(2 + 4)`, после чего полученная сумма умножается на `3`. Добавив круглые скобки, мы получили в результате не `14`, а `18`. Понятие *приоритета* (старшинства) операций в Python имеет тот же смысл, что и в математике. Сначала выполняются операторы в круглых скобках, затем операторы `*` и `/` в порядке их следования слева направо и лишь после этого операторы `+` и `-` (также в направлении слева направо).

Вычисление выражений

Если компьютер выполнил операцию `10 + 5` и выдал результат в виде числа `15`, то говорят, что он *вычислил* выражение. Вычисление выражения сводит его всего к одному значению точно так же, как решение арифметической задачи сводит ее к единственному числу: ответу.

Выражения `10 + 5` и `10 + 3 + 2` имеют одно и то же значение, поскольку результатом вычисления каждого из них является число `15`. Выражениями считаются даже одиночные числа: значением выражения `15` является `15`.

Python продолжает вычислять выражение до тех пор, пока оно не превратится в единственное значение, как показано ниже.



В Python вычисление выражения начинается с наиболее глубоко расположенных круглых скобок в направлении слева направо. Даже если пары скобок вложены друг в друга, заключенные в них части выражения вычисляются по тем же правилам, что и любое другое выражение. Поэтому, когда Python сталкивается с выражением `((7 + 1) / (3 - 1))`, первым вычисляется выражение, заключенное во внутреннюю пару скобок, расположенную слева, `(7 + 1)`, и только после этого вычисляется выражение в скобках справа, `(3 - 1)`. Выражения во внешних скобках вычисляются лишь после того, как каждое из выражений во внутренних скобках будет приведено к единственному значению. Обратите внимание на то, что результат деления представляет собой вещественное значение (с плавающей точкой). Наконец, когда больше не остается выражений, заключенных в скобки, Python выполняет оставшуюся часть вычислений в порядке следования операторов.

В выражении должно быть не менее двух значений, связанных операторами. Если ввести в интерактивной оболочке всего одно значение и оператор, то будет выдано сообщение об ошибке.

```
>>> 5 +
SyntaxError: invalid syntax
```

Эта ошибка возникла потому, что `5 +` не является выражением. В Python ожидается, что у оператора + будет два операнда. Термин *синтаксическая ошибка* означает, что компьютер не смог понять смысл введенной вами инструкции, которая оказалась некорректной. Не забывайте: программирование подразумевает не просто информирование компьютера о том, что он должен сделать, но и знание того, как правильно составлять инструкции, которым он должен следовать.

Ошибки — это нормально!

Ошибки в программах не являются преступлением! Компьютер не сломается из-за того, что вы введете код, вызывающий ошибку. Python просто известит вас о ее возникновении и вновь отобразит подсказку >>>, предлагающую продолжить ввод команд.

Пока вы не приобретете достаточный опыт программирования, смысл этих сообщений будет не слишком понятен для вас. Однако вы всегда сможете "погуглить" текст сообщения и получить ссылки на веб-страницы, где описывается природа конкретных ошибок. Ниже указан адрес веб-страницы, где содержится список наиболее часто встречающихся сообщений об ошибках Python и раскрывается их смысл:

```
https://inventwithpython.com/blog/2012/07/09/
16-common-python-runtime-errors-beginners-find/
```

Сохранение значений в переменных

Программы часто нуждаются в сохранении значений для их последующего использования. Значения можно сохранять в *переменных* с помощью знака равенства =, который также называют *оператором присваивания*. Например, чтобы сохранить значение `15` в переменной `spam`, введите в интерактивной оболочке следующую команду:

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

Переменную можно представить в виде ящика, в котором находится значение `15` (рис. 2.3). Имя переменной `spam` — это надпись на ящике (позволяющая отличать одну переменную от другой), а значение переменной — находящаяся в нем записка.

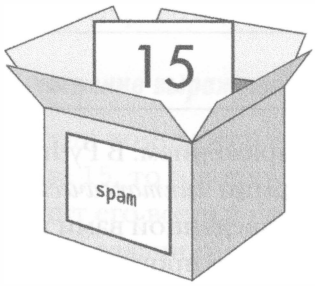


Рис. 2.3. *Переменные можно уподобить ящикам, способным хранить значения*

Нажав клавишу <Enter>, вы не увидите в ответ ничего, кроме пустой строки. Раз не появилось никаких сообщений об ошибках, можете быть уверены в том, что инструкция была выполнена успешно. Появление очередной подсказки >>> — сигнал о возможности ввода следующей инструкции.

В данном случае команда, содержащая оператор присваивания =, создает переменную `spam` и сохраняет в ней значение `15`. В отличие от выражений *инструкции* — это команды, которые не сводятся к вычислению какого-либо значения.

Они связаны с выполнением действий, поэтому в данном случае в интерактивной оболочке ничего не отображается.

Понять, какие команды являются выражениями, а какие — инструкциями, бывает непросто. Достаточно запомнить, что инструкция, выполнение которой дает в результате одно значение, является выражением. Все остальные команды — это инструкции.

Инструкция присваивания начинается с имени переменной, за которым следует знак =, а за ним — выражение (рис. 2.4). Результат вычисления этого выражения сохраняется в указанной переменной.

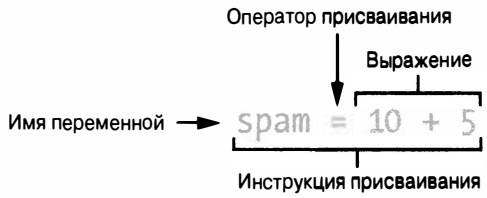


Рис. 2.4. *Составные части инструкции присваивания*

Имейте в виду, что переменные хранят значения, а не выражения, которые им присваиваются. Так, если вы введете инструкцию **`spam = 10 + 5`**, то сначала вычисляется результат выражения `10 + 5`, а затем значение `15` присваивается переменной `spam`, что можно проверить, введя в интерактивной оболочке имя этой переменной.

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

Переменная сама по себе является выражением, значением которого будет значение, хранящееся в этой переменной. Отдельное значение тоже является выражением, значение которого совпадает с ним самим.

```
>>> 15
15
```

А сейчас интересный момент! Если ввести инструкцию **`spam + 5`**, то будет выдано целочисленное значение `20`.

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

Как видите, переменные можно использовать в выражениях наравне со значениями. Поскольку значение `spam` равно 15, вычисление выражения `spam + 5` сводится к вычислению выражения `15 + 5`, что дает `20`.

Изменение значений переменных

Хранящееся в переменной значение можно изменить, введя еще одну инструкцию присваивания. Например, введите показанные ниже команды.

```
>>> spam = 15
❶ >>> spam + 5
❷ 20
❸ >>> spam = 3
❹ >>> spam + 5
❺ 8
```

После ввода первой инструкции `spam + 5` (❶) вы получите в качестве результата `20` (❷), поскольку ранее сохранили в переменной `spam` значение `15`. Но когда вы вводите инструкцию `spam = 3` (❸), то значение `15` *перезаписывается* значением `3` (рис. 2.5). Если теперь вы вновь введете инструкцию `spam + 5` (❹), то результатом вычисления выражения будет уже `8` (❺), поскольку выражение `spam + 5` вычисляется как `3 + 5`. Прежнее значение, хранившееся в переменной `spam`, оказалось забыто.

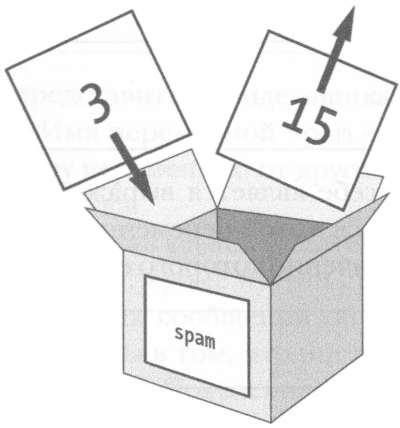


Рис. 2.5. *Значение, хранящееся в переменной `spam`, заменяется значением `3`*

Значение, хранящееся в переменной `spam`, можно использовать даже для того, чтобы присвоить новое значение самой этой переменной.

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

Инструкция `spam = spam + 5` сообщает компьютеру следующее: "Новое значение переменной `spam` равно текущему значению плюс пять". Переменной, указанной слева от знака =, присваивается значение, вычисляемое справа от него. Значение переменой `spam` можно увеличивать на 5 несколько раз подряд.

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

Значение переменной `spam` изменяется всякий раз, когда выполняется очередная инструкция `spam = spam + 5`. Окончательным значением, сохраненным в переменной `spam`, становится `30`.

Имена переменных

Несмотря на то что компьютеру абсолютно безразлично, как вы назовете свою переменную, вы должны взвешенно отнестись к тому, как ее назвать. Назначая переменным имена, отражающие, какого типа данные содержатся в этих переменных, вы облегчите себе и другим людям понимание программы. Вы вправе присвоить своим переменным имена наподобие `abrahamLincoln` или `monkey`, даже если в действительности ваша программа не имеет никакого отношения ни к Аврааму Линкольну, ни к обезьянам, — компьютеру все равно. Но если вы вернетесь к программе после длительного перерыва, то вам может оказаться трудно вспомнить, каково назначение той или иной переменной.

Удачно подобранные имена переменных описывают тип содержащихся в них данных. Представьте, что вы переехали в новый дом, а все ваше движимое имущество хранится в ящиках, помеченных как "Вещи". Так вы никогда ничего не найдете! В качестве типичных имен переменных в примерах книги, а также в значительной части документации к Python, используются имена `spam`, `eggs`, `bacon` и т.п. (по мотивам скетча "Спам" британской комик-группы "Монти Пайтон"), но в своих программах старайтесь давать переменным более описательные имена.

Имена переменных (как и все остальное в Python) чувствительны к регистру букв. *Чувствительность к регистру* (case sensitivity) означает, что одинаковые имена переменных, но записанные с использованием букв разного регистра, считаются разными. Например, в Python `spam`, `SPAM`, `Spam` и `sPAM` — четыре разные переменные. Они не являются взаимозаменяемыми и могут содержать независимые значения.

Резюме

Когда же мы приступим к написанию программ шифрования? Совсем скоро. Но прежде чем вы сможете взламывать шифры, следует изучить еще несколько базовых понятий программирования, которые мы рассмотрим в следующей главе.

В этой главе вы научились применять интерактивную оболочку для выполнения простейших инструкций Python. Вы должны предоставлять Python точные указания относительно того, что следует делать, поскольку компьютеры понимают лишь очень простые инструкции. Вы узнали о том, что Python способен вычислять выражения (т.е. сводить их к единственному значению) и что выражения — это значения (такие, как `2` или `5.4`), объединенные с помощью операторов (таких, как `+` или `-`). Вы также узнали о том, как сохранять значения в переменных, чтобы впоследствии их можно было использовать в программе.

Интерактивная оболочка — удобный инструмент для изучения Python, поскольку можно вводить по одной инструкции за раз и сразу же видеть результат. В главе 3 мы начнем создавать программы, которые содержат множество последовательно выполняющихся инструкций. Мы обсудим ряд других базовых концепций, что позволит вам написать свою первую программу!

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Как выглядит оператор деления: / или \?
2. Какое из приведенных ниже значений является целым, а какое — вещественным?

```
42
3.141592
```

3. Какие из приведенных ниже строк не являются выражениями?

```
4 x 10 + 2
3 * 7 + 1
2 +
42
2 + 2
spam = 42
```

4. Если ввести следующие команды в интерактивной оболочке, то какие результаты отобразятся на экране после выполнения строк ❶ и ❷?

```
spam = 20
❶ spam + 20
SPAM = 30
❷ spam
```

3

СТРОКОВЫЙ ТИП ДАННЫХ И НАПИСАНИЕ ПРОГРАММ

"Единственный способ выучить новый язык программирования – это писать программы на нем".

Брайан Керниган, Деннис Ритчи, "Язык программирования С"

В главе 2 вы узнали достаточно много о целых числах и арифметических операциях. Но Python – это же не калькулятор. Поскольку криптография основана на обработке текстовых значений путем преобразования простого текста в зашифрованный и наоборот, в этой главе вы узнаете о том, как хранить, комбинировать и выводить текст на экран. Вы также напишете свою первую программу, которая будет приветствовать пользователя словами "Hello, world!" и предлагать ему ввести свое имя.

В этой главе...

- Строки
- Конкатенация и репликация строк
- Индексы и срезы
- Функция `print()`
- Написание программ с помощью IDLE
- Сохранение и выполнение программ в IDLE
- Комментарии
- Функция `input()`

Использование строковых значений для работы с текстом

В Python небольшие фрагменты текста называются строковыми значениями или просто *строками*. Все наши программы шифрования и взлома будут работать со строковыми значениями, преобразуя простой текст наподобие `'One if by land, two if by space'` в зашифрованный текст наподобие `'b1rJvsJo!Jyn1q,J702JvsJo!J63nprM'`. Простой и зашифрованный тексты будут представлены в программе строковыми значениями, и Python предлагает множество способов для манипулирования этими значениями.

Строковые значения можно сохранять в переменных точно так же, как целые и вещественные числа. Когда вы вводите строку, заключайте ее в одинарные кавычки (`'`), показывающие, где она начинается и где заканчивается. Введите в интерактивной оболочке следующий код:

```
>>> spam = 'hello'
```

Обрамляющие кавычки не являются частью строкового значения. Python понимает, что `'hello'` — это строка, а `spam` — переменная, поскольку строки заключаются в одинарные кавычки, а имена переменных — нет.

Если вы введете **`spam`**, то увидите содержимое этой переменной (строку `'hello'`).

```
>>> spam = 'hello'
>>> spam
'hello'
```

Так происходит потому, что Python вычисляет значение переменной — в данном случае строку `'hello'`. Строки могут содержать любой алфавитно-цифровой символ. Рассмотрим несколько примеров строк.

```
>>> 'hello'
'hello'
>>> 'KITTENS'
'KITTENS'
>>> ''
''
>>> '7 apples, 14 oranges, 3 lemons'
'7 apples, 14 oranges, 3 lemons'
>>> 'Anything not pertaining to elephants is irrelephant.'
'Anything not pertaining to elephants is irrelephant.'
>>> 'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
```

Обратите внимание на то, что строка `''` не содержит никаких символов; между двумя кавычками ничего не было введено. Такие строки называются *пустыми*.

Конкатенация строк с помощью оператора +

Можно сложить две строки и получить новую с помощью оператора +. Такая операция называется *конкатенацией*. Введите в интерактивной оболочке следующий текст.

```
>>> 'Hello,' + 'world!'
'Hello,world!'
```

Python конкатенирует строки *в точности* в том виде, в каком вы их предоставляете, не вставляя между ними пробелы. Чтобы вставить пробел между `'Hello,'` и `'world!'`, введите пробел в конце строки `'Hello, '`, перед закрывающей кавычкой.

```
>>> 'Hello, ' + 'world!'
'Hello, world!'
```

В результате конкатенации двух строковых значений с помощью оператора + вы получаете новую строку (`'Hello, ' + 'world!'` превращается в `'Hello, world!'`) точно так же, как в результате сложения двух целых чисел получается новое целочисленное значение (`2 + 2` превращается в `4`). Python понимает, что именно должен делать оператор +, исходя из типа данных соответствующих значений. Как вы узнали в главе 2,

тип данных сообщает нам (и компьютеру), какого рода данные участвуют в операции.

Оператор + может использоваться в выражении при условии соответствия типов данных. Если попытаться сложить строку с числом, будет получено сообщение об ошибке. Введите в интерактивной оболочке следующие команды.

```
>>> 'Hello' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> 'Hello' + '42'
'Hello42'
```

Первая команда вызывает ошибку, поскольку `'Hello'` — строка, а `42` — целое число. Однако во второй команде `'42'` — уже строка, которую Python успешно конкатенирует.

Репликация строк с помощью оператора *

Оператор * позволяет умножить строку на целое число, в результате чего строка *реплицируется* (т.е. повторяется) несколько раз, причем кратность повторения определяется целочисленным множителем. Введите в интерактивной оболочке следующие команды.

```
❶ >>> 'Hello' * 3
  'HelloHelloHello'
  >>> spam = 'Abcdef'
❷ >>> spam = spam * 3
  >>> spam
  'AbcdefAbcdefAbcdef'
```

Чтобы реплицировать какую-либо строку, введите ее, затем оператор *, а вслед за ним — требуемое количество повторений строки (❶). Можно также сохранить строку в переменной и проделать то же самое с ней (❷).

В главе 2 вы узнали о том, что результатом применения оператора * к двум целым числам является их произведение. Однако попытка использовать оператор * в отношении двух строковых значений приведет к ошибке.

```
>>> 'Hello' * 'world!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

На примере конкатенации и репликации строк мы видим, что операторы в Python могут выполнять разные действия в зависимости от типов данных операндов. Оператор + реализует сложение и конкатенацию, а оператор * — умножение и репликацию строк.

Получение символов строки с помощью индексов

В программах шифрования часто будет требоваться получить отдельные символы строк, и это можно сделать с помощью *индексации*. Чтобы обратиться к отдельному символу строки, добавьте в конце строкового значения (или имени переменной, содержащей строку) пару квадратных скобок (`[` и `]`), заключив в них число, которое называется *индексом* и задает позицию интересующего вас символа в строке. В Python индексация начинается с 0, поэтому индексом первого символа строки будет 0. Индекс 1 соответствует второму символу, индекс 2 — третьему и т.д.

Введите в интерактивной оболочке следующие команды.

```
>>> spam = 'Hello'
>>> spam[0]
'H'
>>> spam[1]
'e'
>>> spam[2]
'l'
```

Обратите внимание на то, что результатом вычисления выражения `spam[0]` является строковое значение `'H'`, поскольку `H` — первый символ строки `'Hello'`, а нумерация индексов начинается с 0, а не с 1 (рис. 3.1).

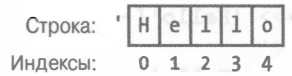


Рис. 3.1. *Строка `'Hello'` и ее индексы*

Индексацию можно применить к переменной, содержащей строковое значение, как это было сделано в предыдущем примере, или к самому строковому значению.

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

Значением выражения `'Zophie'[2]` является строковое значение `'p'`. Строка `'p'` ведет себя как любое другое строковое значение, и ее можно сохранить в переменной. Введите в интерактивной оболочке следующие команды.

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

Если ввести индекс, значение которого слишком велико для данной строки, Python выдаст сообщение об ошибке `"string index out of range"`.

```
>>> 'Hello'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

В данном случае ошибка обусловлена тем, что делается попытка использовать индекс `10` в отношении строки `'Hello'`, которая содержит всего 5 символов.

Отрицательные индексы

Отрицательные индексы начинаются с конца строки и отсчитываются в обратном направлении. Отрицательному индексу –1 соответствует последний символ строки, –2 — это индекс предпоследнего символа строки и т.д. (рис. 3.2).

Введите в интерактивной оболочке следующие команды.

```
>>> 'Hello'[-1]
'o'
>>> 'Hello'[-2]
'l'
>>> 'Hello'[-3]
'l'
>>> 'Hello'[-4]
'e'
>>> 'Hello'[-5]
'H'
>>> 'Hello'[0]
'H'
```

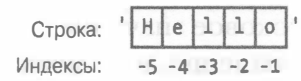


Рис. 3.2. *Строка `'Hello'` и ее отрицательные индексы*

Обратите внимание на то, что индексы –5 и 0 указывают на один и тот же символ. Чаще всего вы будете использовать положительные индексы, но иногда все же удобнее работать с отрицательными.

Получение нескольких символов строки с помощью срезов

Если хотите извлечь из строки более одного символа, используйте срезы вместо индексов. В случае *среза* (slice) также указывается пара квадратных скобок ([и]), но теперь в них задается не одно целочисленное значение, а два. Эти два значения разделяются двоеточием (:) и сообщают Python индексы первого и последнего символов среза. Введите в интерактивной оболочке следующую команду.

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

Строка, которая возвращается в результате вычисления среза, начинается с позиции, указанной первым индексом, и продолжается до позиции, указанной вторым индексом, но не включает ее. В строке `'Howdy'` индексу 0 соответствует символ `'H'`, а индексу 3 — символ `'d'`. Поскольку срез доходит до второго индекса, но не включает его, значением среза `'Howdy'[0:3]` становится строка `'How'`.

Введите в интерактивной оболочке следующие команды.

```
>>> 'Hello, world!'[0:5]
'Hello'
>>> 'Hello, world!'[7:13]
'world!'
>>> 'Hello, world!'[-6:-1]
'world'
>>> 'Hello, world!'[7:13][2]
'r'
```

Обратите внимание на то, что в выражении `'Hello, world!'[7:13][2]` сначала вычисляется промежуточный результат в виде `'world!'[2]`, вычисление которого дает окончательный результат в виде строкового значения `'r'`.

В отличие от индексов использование слишком больших индексов в срезе никогда не приводит к ошибке. В этом случае будет возвращаться результат, соответствующий наиболее широкому из всех возможных соответствий срезу.

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

Выражение `'Hello'[1000:2000]` возвращает пустую строку, поскольку индекс 1000 указывает на позицию за пределами конца строки, а раз так, то отсутствуют символы, которые могли бы войти в данный срез. И хотя здесь это не продемонстрировано, срезы также можно сохранять в переменных.

Опускание индексов в срезах

Если опустить первый индекс в обозначении среза, Python автоматически использует для него значение 0. Выражения `'Howdy'[0:3]` и `'Howdy'[:3]` возвращают одну и ту же строку.

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

Если опустить второй индекс, Python автоматически выберет оставшуюся часть строки, начиная с первого индекса.

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

Индексы можно опускать по-разному. Введите в интерактивной оболочке следующие команды.

```
>>> myName = 'Zophie the Fat Cat'
>>> myName[-7:]
'Fat Cat'
>>> myName[:10]
'Zophie the'
>>> myName[7:]
'the Fat Cat'
```

Как видите, можно использовать даже отрицательные индексы. Поскольку в первом примере –7 — начальный индекс, Python отсчитывает индексы с конца строки в обратном направлении, используя указанный индекс в качестве начального. Затем он возвращает все символы, начиная с данного индекса и до конца строки, поскольку второй индекс опущен.

Вывод значений с помощью функции `print()`

Рассмотрим, как работает инструкция другого рода: функция `print()`. Введите в интерактивной оболочке следующие команды.

```
>>> print('Hello!')
Hello!
>>> print(42)
42
```

Функции, такие как `print()`, содержат код, предназначенный для решения конкретной задачи. В данном случае это вывод значений на экран. В Python имеется множество полезных функций, благодаря которым вы сэкономите уйму времени. *Вызов* функции означает выполнение содержащегося в ней кода.

В данном примере в функцию `print()` передается значение, указанное в круглых скобках, и она выводит его на экран. Значения, которые передаются функции при ее вызове, называются *аргументами*. Когда вы начнете писать программы, вы будете использовать функцию `print()` для отображения текста на экране.

В функцию `print()` можно передать не одиночное значение, а выражение. Такое возможно, потому что значением, фактически переданным в функцию, будет результат вычисления этого выражения. Введите в интерактивной оболочке выражение, конкатенирующее строки, в качестве аргумента.

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

Выражение `'Hello, ' + spam` возвращает значение `'Hello, ' + 'Al'`, результатом вычисления которого является строка `'Hello, Al'`. Именно эта строка и передается в функцию `print()` при ее вызове.

Вывод экранированных символов

Иногда строка содержит символы, сбивающие Python с толку. Например, если в строке встретится одинарная кавычка, то вы получите сообщение об ошибке, поскольку Python посчитает, что она обозначает конец строки, и последующий текст — некорректный код, а вовсе не остаток строки. Введите в интерактивной оболочке следующий код, чтобы увидеть, как все происходит в реальности.

```
>>> print('Al's cat is named Zophie.')
SyntaxError: invalid syntax
```

Чтобы вставить одинарную кавычку в строку, следует *экранировать* ее. *Экранированный символ* (escape character) — это символ, которому предшествует обратная косая черта, например `\t`, `\n` или `\'`. Косая черта сообщает Python о том, что следующий за ней символ имеет особый смысл. Введите следующий код в интерактивной оболочке.

```
>>> print('Al\'s cat is named Zophie.')
Al's cat is named Zophie.
```

Теперь Python понимает, что одинарная кавычка — часть строки, а не признак ее конца.

В табл. 3.1 перечислено несколько экранированных символов, поддерживаемых в Python.

Таблица 3.1. Экранированные символы

Экранированный символ	Отображаемый результат
`\\`	Обратная косая черта
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\n`	Символ новой строки
`\t`	Табуляция

Экранированному символу всегда предшествует обратная косая черта. Если вы захотите включить в строку сам символ обратной косой черты, то недостаточно указать только его, поскольку следующий за ним символ будет интерпретироваться Python как экранированный. Например, следующая команда не будет корректно работать.

```
>>> print('It is a green\teal color.')
It is a green    eal color.
```

Здесь `'t'` в `'teal'` идентифицируется как экранированный символ, поскольку ему предшествует косая черта. Экранированный символ `\t` имитирует нажатие клавиши <Tab> на клавиатуре.

Введите вместо этого следующий код.

```
>>> print('It is a green\\teal color.')
It is a green\teal color.
```

На этот раз строка отобразится так, как и предполагалось, поскольку добавление второй косой черты превращает первоначальную в экранированный символ.

Одинарные и двойные кавычки

Строки не обязательно заключать в одинарные кавычки. Вместо них можно использовать и двойные кавычки. Следующие команды выводят один и тот же текст.

```
>>> print('Hello, world!')
Hello, world!
>>> print("Hello, world!")
Hello, world!
```

Однако смешивать одинарные и двойные кавычки нельзя. Следующая команда вызывает ошибку.

```
>>> print('Hello, world!")
SyntaxError: EOL while scanning string literal
```

В книге используются в основном одинарные кавычки, поскольку их немного проще вводить, хотя для Python это совершенно безразлично.

Точно так же, как для вставки одинарной кавычки в строку, заключенную в одинарные кавычки, следует использовать экранированный символ `\'`, понадобится использовать экранированный символ `\"` для вставки двойных кавычек в строку, ограниченную двойными кавычками. Рассмотрим, например, следующие две команды.

```
>>> print('Al\'s cat is Zophie. She says, "Meow."')
Al's cat is Zophie. She says, "Meow."
>>> print("Zophie said, \"I can say things other than 'Meow' you know.\"")
Zophie said, "I can say things other than 'Meow' you know."
```

В экранировании двойных кавычек в строке, ограниченной одинарными кавычками, и одинарных кавычек в строке, ограниченной двойными кавычками, нет необходимости. Интерпретатор Python понимает, что если строка начинается с кавычек одного типа, то кавычки другого типа не могут служить признаком ее окончания.

Написание программ в редакторе файлов IDLE

До сих пор вы вводили инструкции по одной за раз в окне интерактивной оболочки. Но когда вы пишете программы, вы вводите сразу несколько инструкций и запускаете на выполнение их все, не дожидаясь каждый раз ввода очередной инструкции. Пришло время и вам написать свою первую программу!

Приложение, которое предоставляет интерактивную оболочку, называется *IDLE* (Integrated Development and Learning Environment – интегрированная среда разработки). Помимо интерактивной оболочки в IDLE также есть редактор файлов, который мы сейчас откроем.

Выберите в верхней части окна оболочки Python пункты меню File⇨ New Window, чтобы открыть новое пустое окно редактора файлов, в котором вы сможете ввести код программы (рис. 3.3). В правом нижнем углу этого окна отображаются номера строки и столбца текущей позиции курсора.

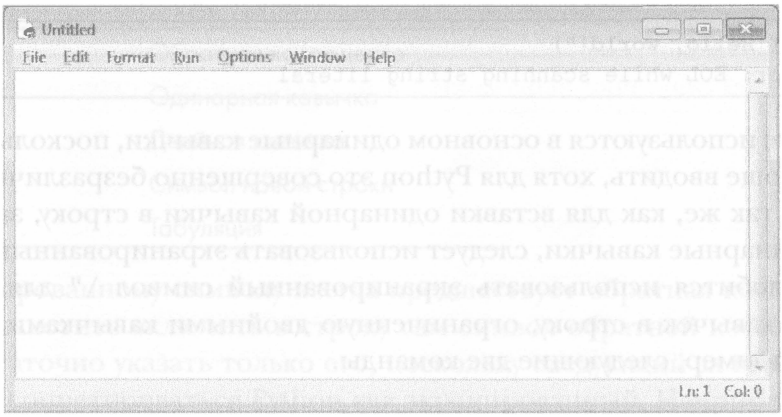


Рис. 3.3. *Окно редактора файлов с текущей позицией курсора в столбце 0 строки 1*

Вы всегда сможете узнать, в каком окне находитесь (редактор файлов или интерактивная оболочка), по наличию подсказки >>>. Она отображается только в окне интерактивной оболочки, тогда как в окне редактора файлов ее нет.

Исходный код программы "Hello, world!"

По давно сложившейся традиции первой программой, с написания которой новички начинают изучение языка, является программа, которая выводит на экран текст `"Hello, world!"`. Мы создадим эту программу, введя соответствующий текст в новом окне редактора файлов. Мы называ-

ем этот текст *исходным кодом* программы, поскольку он содержит инструкции, которым будет следовать Python.

Исходный код программы "Hello, world!" доступен для загрузки на сайте издательства (см. введение). Если вы столкнетесь с ошибками при вводе кода, сравните свой код с приведенным в книге, используя онлайн-утилиту `Diff` (описана в следующем разделе). Не забывайте о том, что вводить номера строк не следует; они указаны в книге исключительно для удобства ссылок на соответствующие строки кода при его описании.

hello.py

```
1. # This program says hello and asks for my name.
2. print('Hello, world!')
3. print('What is your name?')
4. myName = input()
5. print('It is good to meet you, ' + myName)
```

В IDLE инструкции различного типа выделяются разными цветами. Когда вы введете весь код, окно должно выглядеть примерно так, как показано на рис. 3.4.

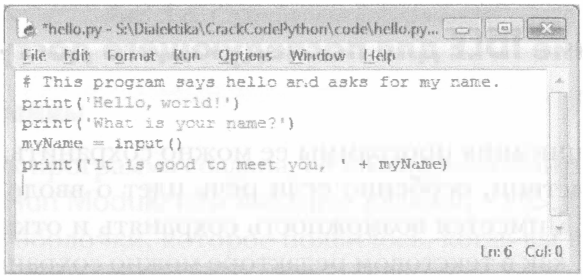


Рис. 3.4. *Окно редактора файлов по завершении ввода кода*

Проверка правильности исходного кода с помощью онлайн-утилиты `Diff`

Несмотря на то что можно скопировать и вставить текст программы *hello.py* или загрузить его на сайте книги, лучше ввести его вручную. Так вы ближе познакомитесь с исходным кодом программы. Однако в процессе ввода текста в файловом редакторе есть риск допустить ошибки.

Чтобы сравнить введенный вами код с кодом, используемым в книге, используйте онлайн-утилиту `Diff` (рис. 3.5). Скопируйте текст своей программы, а затем перейдите по адресу `http://inventwithpython.com/hacking/diff/`. Выберите программу *hello.py* в списке слева, вставьте свой код в текстовое поле справа и щелкните на кнопке Compare. Утилита по-

кажет любые различия между вашим и книжным вариантами кода. Это простой способ обнаружить опечатки, вызывающие появление ошибок в вашей программе.

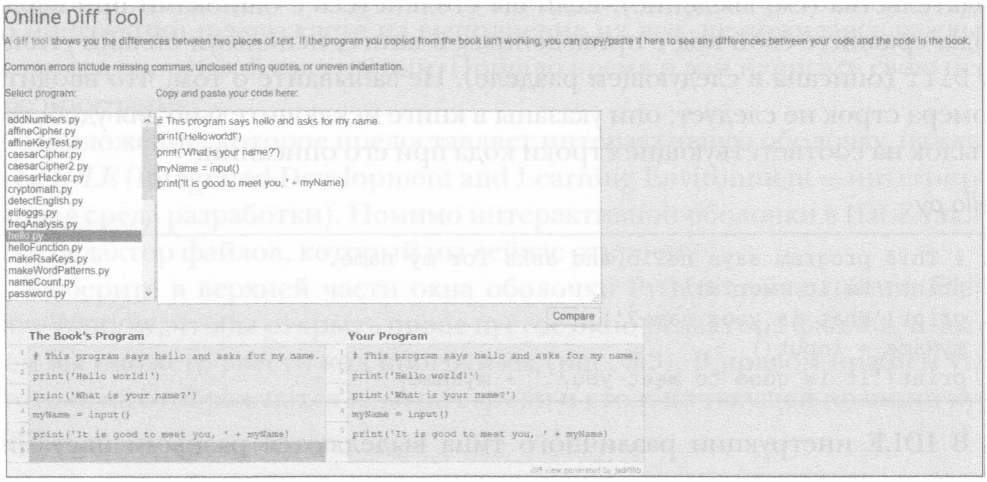


Рис. 3.5. *Онлайн-утилита* `Diff`

Использование IDLE для последующего доступа к программе

В процессе написания программы ее можно сохранить, чтобы вернуться к ней впоследствии, особенно если речь идет о вводе очень больших программ. В IDLE имеется возможность сохранять и открывать программы точно так же, как в текстовом редакторе можно сохранять и открывать документы.

Сохранение программы

Введя исходный код, сохраните его, чтобы вам не пришлось вводить его заново всякий раз, когда вы захотите выполнить программу. Выберите пункты меню File⇨Save As в верхней части окна редактора файлов, чтобы открыть диалоговое окно Сохранить как (рис. 3.6). Введите **`hello.py`** в поле Имя файла и щелкните на кнопке Сохранить.

Почаще сохраняйте свои программы в процессе их ввода, чтобы не потерять часть работы в случае сбоя компьютера или случайного выхода из IDLE. Для сохранения файла можно также использовать комбинацию клавиш <Ctrl+S> (Windows/Linux) или <⌘+S> (macOS).

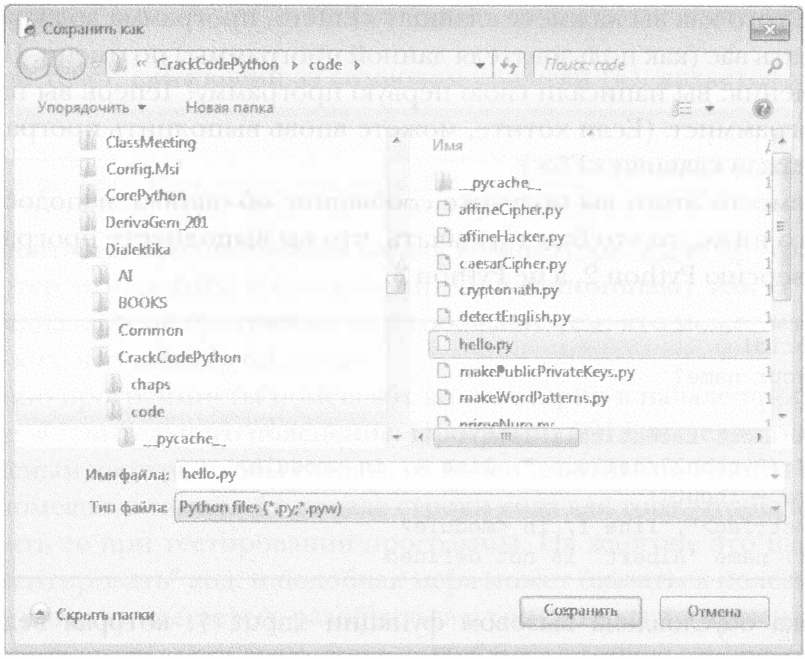


Рис. 3.6. *Сохранение программы*

Выполнение программы

После того как программа создана, ее можно выполнить. Выберите пункты меню Run⇨Run Module или нажмите клавишу <F5>. Программа будет запущена в окне оболочки, которое появилось, когда вы впервые запустили IDLE. Запомните: клавишу <F5> следует нажимать в окне редактора файлов, а не в окне интерактивной оболочки.

Когда программа выдаст запрос, введите свое имя (рис. 3.7).

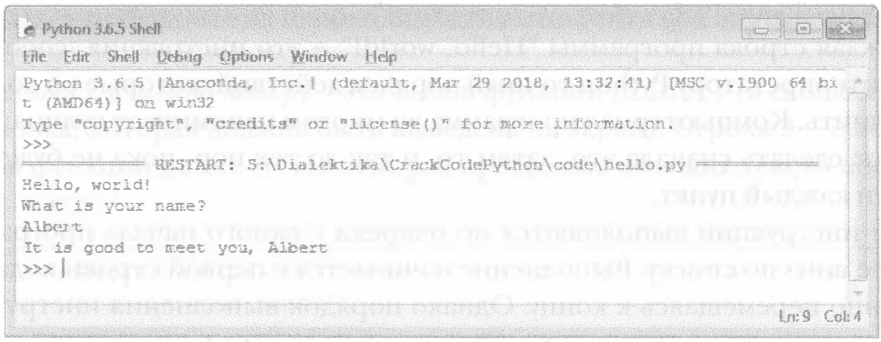


Рис. 3.7. *Приблизительно так выглядит окно интерактивной оболочки во время выполнения программы "Hello, world!"*

После того как вы нажмете клавишу <Enter>, программа должна поприветствовать вас (как пользователя данной программы) по имени. Примите поздравления: вы написали свою первую программу! Теперь вы начинающий программист. (Если хотите, можете вновь выполнить программу, повторно нажав клавишу <F5>.)

Если вместо этого вы получите сообщение об ошибке наподобие приведенного ниже, то это будет означать, что вы выполняете программу, используя версию Python 2, а не Python 3.

```
Hello, world!
What is your name?
Albert
Traceback (most recent call last):
  File "C:/Python27/hello.py", line 4, in <module>
    myName = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

Ошибка обусловлена вызовом функции `input()`, которая ведет себя по-разному в версиях Python 2 и 3. Прежде чем продолжать, установите версию Python 3, выполнив инструкции, которые приводились во введении.

Открытие программы, которую вы сохранили ранее

Закройте редактор файлов, щелкнув на кнопке × в правом верхнем углу окна. Чтобы повторно загрузить сохраненную программу, выберите пункты меню File⇨Open и укажите в открывшемся окне файл *hello.py*, после чего щелкните на кнопке Открыть. Сохраненная вами программа *hello.py* должна открыться в окне редактора файлов.

Как работает программа "Hello, world!"

Каждая строка программы "Hello, world!" — это инструкция, сообщающая интерпретатору Python точный порядок действий, которые он должен выполнить. Компьютерная программа во многом напоминает кулинарный рецепт: сделать сначала это, затем то, и так до тех пор, пока не будет выполнен каждый пункт.

Все инструкции выполняются по очереди с самого начала программы и далее вниз по списку. Выполнение начинается с первой строки кода, постепенно перемещаясь к концу. Однако порядок выполнения инструкций может отклоняться от прямолинейного, о чем пойдет речь в главе 4.

Проанализируем программу "Hello, world!" строка за строкой, чтобы понять ее работу, начиная со строки 1.

Комментарии

Любой текст, следующий за символом решетки (#), является *комментарием.*

```
1. # This program says hello and asks for my name.
```

Комментарии предназначены не для компьютера, а для программиста. Компьютер игнорирует комментарии. Они напоминают вам о том, для чего предназначена программа, или сообщают тем, кто может изучать исходный код, что именно он делает.

Обычно программисты помещают комментарии в начале программы в качестве вступительного пояснения. Программа IDLE выделяет комментарии красным цветом, чтобы сделать их более заметными. Иногда программисты помещают символ # в начале строки кода для того, чтобы временно отключить ее при тестировании программы. На жаргоне это называется "закомментировать" код, и подобная мера может оказаться полезной в тех случаях, когда вы пытаетесь разобраться в том, почему программа не работает. Впоследствии, когда вам вновь понадобится данная строка, вы сможете удалить символ #.

Вывод сообщений для пользователя

Следующие две команды отображают сообщения, предназначенные для пользователя, с помощью функции `print()`. Функция — это своего рода мини-программа в вашей программе. Огромное преимущество функций состоит в том, что они скрывают свой код. Нам достаточно знать лишь, что именно делает функция, а не как она это делает. Например, мы знаем, что функция `print()` отображает текст на экране, но как именно это происходит, от нас скрыто. Вызов функции — это команда, которая указывает программе на необходимость выполнить код внутри указанной функции.

Строка 2 в файле *hello.py* — это вызов функции `print()` (в скобках указана строка, которая должна быть выведена на экран). Строка 3 — еще один вызов функции `print()`. На этот раз программа выводит текст `'What is your name?'`.

```
2. print('Hello, world!')
3. print('What is your name?')
```

Мы добавляем скобки после имени функции, чтобы было понятно, что речь идет о функции `print()`, а не о переменной `print`. Круглые скобки

говорят Python о том, что мы вызываем функцию, аналогично тому, как кавычки в записи `'42'` говорят о том, что мы используем строку `'42'`, а не число `42`.

Ввод данных пользователем

Строка 4 содержит инструкцию присваивания, в которой в переменную (`myName`) записывается результат вызова функции `input()`:

```
4. myName = input()
```

Когда вызывается функция `input()`, программа ожидает, что пользователь введет какой-то текст и нажмет клавишу <Enter>. Введенная пользователем текстовая строка (имя пользователя) становится строковым значением, сохраняемым в переменной `myName`.

Как и в случае выражений, результатом вызова функции является единственное значение. Это значение называется *возвращаемым*. В данном случае возвращаемым значением функции `input()` является введенная пользователем строка, которой должно стать имя пользователя. Если пользователь введет имя `Albert`, то вызов функции `input()` вернет строку `'Albert'`.

В отличие от функции `print()`, функция `input()` не нуждается в аргументах, и именно поэтому мы вызываем ее, оставляя скобки пустыми.

Последняя строка кода программы *hello.py* — это еще один вызов функции `print()`.

```
5. print('It is good to meet you, ' + myName)
```

В строке 5 внутри вызова функции `print()` мы используем оператор `+` для конкатенации (объединения) строки `'It is good to meet you, '` и строки, сохраненной в переменной `myName`, т.е. имени, введенного в программу пользователем. Благодаря этому программа приветствует пользователя, обращаясь к нему по имени.

Завершение программы

После выполнения последней строки программа завершает работу. При этом компьютер "забывает", т.е. удаляет из памяти, все переменные, в том числе и строку, сохраненную в переменной `myName`. Если вы вновь запустите программу и введете другое имя, то именно оно и будет выведено программой.

```
Hello, world!
What is your name?
Zophie
It is good to meet you, Zophie
```

Помните, что компьютер выполняет строго те действия, которые вы запрограммировали для него. В данной программе он запрашивает ваше имя, позволяет ввести строку, а затем отображает на экране строку, которую вы только что ввели.

Однако компьютеры не способны к рассуждениям. Программе безразлично, введете ли вы свое имя, имя другого человека или вообще какую-либо несуразицу. Вы можете ввести любой текст, какой только захотите, и компьютер безропотно примет его.

```
Hello, world!
What is your name?
great
It is good to meet you, great
```

Резюме

Чтобы писать программы, необходимо владеть языком общения с компьютером. Вы уже узнали немного о том, как это делается, в главе 2, а сейчас даже смогли объединить несколько инструкций на языке Python в готовую программу, которая запрашивает имя пользователя и приветствует его.

В этой главе вы познакомились с несколькими новыми способами манипулирования строками, такими как использование оператора + для конкатенации строк. Вы также научились применять индексы и срезы для создания новых строк на основе имеющихся.

Последующие программы в книге будут намного более сложными, однако каждая их строка будет объясняться. У вас всегда есть возможность ввести инструкцию в интерактивной оболочке, чтобы понять, что в действительности она делает, прежде чем включать ее в программу.

В следующей главе мы приступим к написанию первой программы шифрования на основе обратного шифра.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Если выполнить инструкцию присваивания `spam = 'Cats'`, то что выведут следующие строки кода?

```
spam + spam + spam
spam * 3
```

2. Что выведут следующие строки кода?

```
print("Dear Alice,\nHow are you?\nSincerely,\nBob")
print('Hello' + 'Hello')
```

3. Если выполнить инструкцию присваивания `spam = 'Four score and seven years is eighty seven years.'`, то что выведет каждая из следующих строк кода?

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

4. В каком окне отображается подсказка >>>: в окне интерактивной оболочки или в окне редактора файлов?
5. Что выведет следующая строка кода?

```
#print('Hello, world!')
```

4

ОБРАТНЫЙ ШИФР

"Разве наше воспитание готовит нас к таким извращениям? Разве наши законы им потакают? Могли бы они остаться незамеченными в стране... где все люди добровольно подглядывают друг за другом?"

Джейн Остин, "Нортенгерское аббатство"

Обратный шифр зашифровывает сообщение, выводя его в обратном порядке, поэтому зашифрованной версией текста "Hello, world!" будет "!dlrow ,olleH". Чтобы дешифровать зашифрованное сообщение и получить исходный текст, необходимо просто обратить его. Операции шифрования и дешифрования выполняются с помощью одного и того же алгоритма.

Однако такой обратный шифр является слабым, и получить простой текст из зашифрованной версии не составит никакого труда. Одного взгляда на зашифрованный текст достаточно для того, чтобы определить, что это всего лишь сообщение, записанное в обратном порядке:

.оге ътатичорп онжолсен тедуб мав ,тскет йыннаворфишаз и отэ ътох ,ремирпаН

В то же время объяснить, как работает программа обратного шифрования, будет нетрудно, в связи с чем мы и используем обратный шифр в качестве основы для создания нашей первой программы шифрования.

В этой главе...

- Функция `len()`
- Цикл `while`
- Булев тип данных
- Операторы сравнения
- Условия
- Блоки

Исходный код программы `Reverse Cipher`

Выберите в IDLE пункты меню File⇨New Window для создания нового окна редактора файлов. Введите приведенный ниже код (естественно, без номеров строк), сохраните его в файле *reverseCipher.py* и выполните, нажав клавишу <F5>.

reverseCipher.py

```
 1. # Программа обратного шифрования
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. message = 'Three can keep a secret, if two of them are dead.'
 5. translated = ''
 6.
 7. i = len(message) - 1
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Пробный запуск программы `Reverse Cipher`

Выполнив программу *reverseCipher.py*, вы должны получить следующий результат:

```
.daed era meht fo owt fi ,terces a peek nac eerhT
```

Чтобы дешифровать это сообщение, скопируйте текст `.daed era meh t fo owt fi ,terces a peek nac eerhT` в буфер обмена, выделив сообщение и нажав комбинацию клавиш <Ctrl+C> (Windows/Linux) или <⌘+C>

(macOS). Затем вставьте скопированный текст, используя комбинацию клавиш <Ctrl+V> (Windows/Linux) или <⌘+V> (macOS), в качестве строкового значения, сохраняемого в переменной `message` в строке 4. Проследите за тем, чтобы при этом не были случайно удалены одинарные кавычки, ограничивающие строку. Обновленная строка 4 должна выглядеть так (изменение выделено полужирным шрифтом):

```
4. message = '.daed era meht fo owt fi ,terces a peek nac eerhT'
```

Теперь, когда вы запустите программу *reverseCipher.py*, она выведет исходное сообщение:

```
Three can keep a secret, if two of them are dead.
```

Ввод комментариев и установка переменных

Первые две строки программы *reverseCipher.py* — это комментарии, описывающие назначение программы и содержащие адрес веб-сайта, на котором ее можно найти.

```
1. # Программа обратного шифрования
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
```

Фраза `BSD Licensed` означает, что любой человек может свободно копировать и изменять эту программу при условии, что будет указано авторство исходной версии программы (в данном случае во второй строке указан адрес веб-сайта книги). Я хочу, чтобы исходный файл содержал эту информацию. Тогда при загрузке программы из Интернета пользователи будут знать, где находится ее источник. Они также будут знать, что это программа с открытым исходным кодом, а значит, ее можно свободно распространять.

Строка 3 — пустая, и Python игнорирует ее. В строке 4 хранится сообщение, которое мы хотим зашифровать и сохранить в переменной `message`.

```
4. message = 'Three can keep a secret, if two of them are dead.'
```

Всякий раз, когда мы хотим зашифровать или дешифровать новое сообщение, нам достаточно ввести его в строке 4.

Переменная `translated` в строке 5 — это переменная, в которой программа сохраняет обращенную строку.

```
5. translated = ''
```

Вначале переменная `translated` содержит пустую строку. (Не забывайте о том, что пустая строка представляет собой две одинарные кавычки, а не одну двойную.)

Определение длины строки

Строка 7 — инструкция присваивания, сохраняющая значение в переменной `i`.

```
7. i = len(message) - 1
```

Выражением, которое вычисляется и значение которого присваивается переменной, является `len(message) - 1`. Первая часть этого выражения, `len(message)`, представляет собой вызов функции `len()`, которая получает строковый аргумент, подобно функции `print()`, и возвращает целочисленное значение, равное количеству символов в строке (т.е. длину строки). В данном случае мы передаем функции `len()` переменную `message`, поэтому вызов `len(message)` возвращает количество символов в сообщении, сохраненном в `message`.

Давайте поэкспериментируем с функцией `len()`. Введите в интерактивной оболочке следующие команды.

```
>>> len('Hello')
5
>>> len('')
0
>>> spam = 'Al'
>>> len(spam)
2
>>> len('Hello,' + ' ' + 'world!')
13
```

Значения, возвращаемые функцией `len()`, говорят нам о том, что строка `'Hello'` содержит пять символов, а пустая строка — нуль символов. Если сохранить строку `'Al'` в переменной и передать эту переменную функции `len()`, то она вернет значение 2. Если передать функции `len()` выражение `'Hello,' + ' ' + 'world!'`, то она вернет 13. Это объясняется тем, что значением выражения `'Hello,' + ' ' + 'world!'` является строка `'Hello, world!'`, которая содержит 13 символов (пробел и восклицательный знак считаются символами).

Теперь, когда вы понимаете, как работает функция `len()`, вернемся к строке 7 программы *reverseCipher.py*. В ней определяется индекс последнего символа сообщения путем вычитания 1 из возвращаемого значения вызова

len(message). Мы должны вычесть 1, поскольку индексация ведется от нуля, и, например, в строке 'Hello' индексы изменяются от 0 до 4. Полученное целочисленное значение сохраняется в переменной i.

Знакомство с циклом while

Строка 8 — это инструкция Python, называемая *циклом* while.

```
8. while i >= 0:
```

Цикл while состоит из четырех частей (рис. 4.1).

Рис. 4.1. *Составные части цикла* while

Условие — это выражение, используемое в инструкции while. Блок кода в цикле while будет выполняться до тех пор, пока удовлетворяется указанное условие.

Чтобы понять, как работают циклы while, необходимо сначала узнать кое-что о булевых значениях, операторах сравнения и блоках.

Булев тип данных

Булев тип данных имеет только два возможных значения: True (истина) или False (ложь). Эти значения чувствительны к регистру (вы всегда должны использовать прописные буквы T и F, оставляя все остальные строчными). Они не являются строковыми значениями, поэтому заключать слова True и False в кавычки не следует.

Поработайте с булевыми значениями, введя следующие инструкции в интерактивной оболочке.

```
>>> spam = True
>>> spam
True
>>> spam = False
```

```
>>> spam
False
```

Как и значения любого другого типа, булевы значения можно сохранять в переменных.

Операторы сравнения

Взгляните на выражение, указанное в строке 8 программы *reverseCipher.py* после ключевого слова `while`.

```
8. while i >= 0:
```

Это выражение (`i >= 0`) содержит два значения (переменная `i` и целочисленное значение `0`), соединенные знаком >=, который является оператором "больше чем или равно". Оператор >= — один из операторов сравнения.

Мы используем операторы сравнения для того, чтобы сравнить два значения и получить ответ в виде булева значения `True` или `False`. Операторы сравнения приведены в табл. 4.1.

Таблица 4.1. Операторы сравнения

Оператор	Выполняемая операция
<	Меньше чем
>	Больше чем
<=	Меньше чем или равно
>=	Больше чем или равно
==	Равно
!=	Не равно

Введите следующие выражения в интерактивной оболочке, чтобы увидеть, какие значения они возвращают.

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10.5
False
>>> 10.5 < 11.3
True
>>> 10 < 10
False
```

Выражение `0 < 6` возвращает булево значение `True`, поскольку число `0` меньше числа `6`. Но поскольку число `6` не меньше числа `0`, то выражение `6 < 0` возвращает `False`. Выражение `50 < 10.5` равно `False`, поскольку `50` не меньше, чем `10,5`. Выражение `10 < 11.3` возвращает `True`, поскольку `10,5` меньше, чем `11,3`.

Обратимся к выражению `10 < 10`. Оно равно `False`, поскольку число `10` не меньше числа `10`. Эти числа строго равны. (Если бы Алиса была того же роста, что и Боб, то вы не могли бы сказать, что Алиса ниже Боба. Это утверждение было бы ложным.)

Введите другие выражения, используя операторы <= (меньше чем или равно) и >= (больше чем или равно).

```
>>> 10 <= 20
True
>>> 10 <= 10
True
>>> 10 >= 20
False
>>> 20 >= 20
True
```

Обратите внимание на то, что `10 <= 10` равно `True`, поскольку этот оператор проверяет, меньше *или равно* число `10` числу `10`. Не забывайте о том, что в операторах "меньше чем или равно" и "больше чем или равно" знаки < и > всегда предшествуют знаку =.

Далее введите в интерактивной оболочке несколько выражений, в которых используются операторы сравнения == (равно) и != (не равно), чтобы увидеть, как они работают.

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
```

Для целых чисел эти операторы работают так, как ожидалось. Сравнение равных между собой целых чисел с помощью оператора == дает `True`, а неравных — `False`. При сравнении таких чисел с помощью оператора != мы получаем прямо противоположные результаты.

Аналогичным образом работает и сравнение строк.

```
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Goodbye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Goodbye' != 'Hello'
True
```

Регистр букв имеет значение для Python, поэтому строковые значения, в которых регистры букв совпадают не полностью, не являются одной и той же строкой. Например, строки `'Hello'` и `'HELLO'` разные, поэтому результатом их сравнения с помощью оператора == будет `False`.

Обратите внимание на различие между оператором присваивания (=) и оператором проверки равенства (==). Одиночный знак равенства используется для присваивания значения переменной, тогда как двойной (==) — для проверки равенства двух значений. Если вы спрашиваете Python, равны ли два значения, используйте оператор ==. Если же вы приказываете Python установить для переменной определенное значение, то используйте оператор =.

В Python строковые и целочисленные значения всегда считаются разными и никогда не могут быть равны. Например, введите в интерактивной оболочке следующие команды.

```
>>> 42 == 'Hello'
False
>>> 42 == '42'
False
>>> 10 == 10.0
True
```

Несмотря на кажущуюся схожесть, целое число `42` и строка `'42'` не считаются равными, поскольку строка — это не то же самое, что число. В то же время целые и вещественные числа могут быть равными, поскольку и то, и то — числа.

Когда вы работаете с операторами сравнения, помните, что результатом вычисления каждого выражения всегда будет значение `True` или `False`.

Блоки

Блок — это одна или несколько строк кода, сгруппированных путем создания для них отступа одной и той же минимальной величины (определяемой количеством пробелов перед началом строки).

Первая строка блока выделяется отступом величиной в четыре пробела. Любая последующая строка с отступом не менее четырех пробелов является частью блока. Если строка выделяется дополнительными четырьмя пробелами (так что общее количество пробелов в ее начале становится равным восьми), то это означает начало нового блока внутри первого блока. Блок заканчивается, когда встречается строка кода с отступом той же величины, что и перед началом блока.

Рассмотрим некий воображаемый код (не имеет значения, что за код, поскольку нас интересует лишь отступ каждой строки). Чтобы упростить подсчет количества пробелов в отступах, отобразим их в виде жирных точек.

```
1. codecodecode          # 0 пробелов отступа
2. ····codecodecode      # 4 пробела отступа
3. ····codecodecode      # 4 пробела отступа
4. ········codecodecode  # 8 пробелов отступа
5. ····codecodecode      # 4 пробела отступа
6.
7. ····codecodecode      # 4 пробела отступа
8. codecodecode          # 0 пробелов отступа
```

Как видите, в строке 1 отступы отсутствуют; таким образом, количество пробелов перед строкой кода равно нулю. Однако отступ строки 2 равен четырем пробелам. Поскольку этот отступ больше отступа предыдущей строки, мы знаем, что здесь начинается новый блок. Отступ строки 3 также равен четырем пробелам, откуда следует, что она является продолжением блока.

Строка 4 имеет еще больший отступ (восемь пробелов), что означает начало нового блока. Этот блок находится внутри другого блока. В Python одни блоки могут находиться в других блоках.

В строке 5 отступ уменьшился до четырех пробелов, указывая на окончание блока, находящегося на предыдущей строке. Строка 4 является единственной строкой этого блока. Поскольку строка 5 имеет тот же отступ, что и блок, занимающий строки 2 и 3, она все еще является частью первоначального внешнего блока, хоть и не является частью блока, занимающего строку 4.

Строка 6 — пустая, поэтому мы игнорируем ее, так как она не оказывает на блоки никакого влияния.

Отступ строки 7 равен четырем пробелам, поэтому мы знаем, что блок, начавшийся в строке 2, продолжается и в строке 7.

Строка 8 имеет нулевой отступ, что меньше отступа предыдущей строки. Это уменьшение величины отступа говорит нам о том, что предыдущий блок, который начался в строке 2, закончился.

В этом коде имеются два блока. Первый блок занимает строки 2–7. Второй блок состоит только из строки 4 (и находится внутри другого блока).

Примечание

Величина отступов, выделяющих блоки, не обязательно должна быть кратной четырем пробелам. Вы вправе выбирать для этого любое количество пробелов, тогда как использование четырех пробелов является всего лишь общепринятым соглашением.

Инструкция `while`

Рассмотрим полную инструкцию `while`, начинающуюся в строке 8 программы *reverseCipher.py*.

```
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Инструкция `while` сообщает Python о том, что сначала необходимо проверить условие в строке 8, т.е. `i >= 0`. Инструкцию `while i >= 0:` можно трактовать следующим образом: "Пока переменная `i` больше или равна нулю, выполнять код следующего блока". Если условие равно `True`, программа входит в блок, следующий за инструкцией `while`. Проанализировав отступы, вы увидите, что этот блок включает строки 9 и 10. Достигнув конца блока, программа возвращается к инструкции `while` в строке 8 и вновь проверяет условие. Если оно по-прежнему равно `True`, выполнение переходит в начало блока, и блок вновь выполняется.

Если условие цикла `while` равно `False`, программа пропускает код блока, и выполнение передается первой строке, непосредственно следующей за этим блоком (т.е. строке 12).

"Наращивание" строки

Итак, сначала переменная `i` в строке 7 устанавливается равной длине сообщения минус 1, после чего цикл `while` в строке 8 продолжает выполнение инструкций, входящих в состав следующего блока, до тех пор пока условие `i >= 0` не станет равно `False`.

```
 7. i = len(message) - 1
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Строка 9 — это инструкция присваивания, которая сохраняет значение в переменной `translated`. Этим сохраняемым значением является текущее значение переменной `translated`, конкатенируемое с символом, позиция которого в сообщении определяется индексом `i`. В результате строковое значение, сохраняемое в переменной `translated`, "наращивается" по одному символу за раз до тех пор, пока не будет зашифрована вся строка.

Строка 10 — это тоже инструкция присваивания, в соответствии с которой сначала из текущего значения `i` вычитается `1` (эта операция называется *декрементом*), после чего полученный результат становится новым значением `i`.

Далее следует строка 12 (пустая строка 11 игнорируется), но поскольку она имеет меньший (нулевой) отступ, то Python знает, что это означает завершение блока цикла `while`. Поэтому, вместо того чтобы выполнить строку 12, программа возвращается в строку 8, в которой вновь проверяется, соблюдается ли условие продолжения цикла. Если результат проверки условия равен `True`, то это означает повторное выполнение инструкций блока (строки 9 и 10). Так будет продолжаться до тех пор, пока условие не станет равно `False` (т.е. значение `i` стало меньше 0), и только в этом случае программа перейдет к первой строке, непосредственно следующей за блоком (строка 12).

Детально проанализируем работу цикла, чтобы понять, сколько раз будет выполнен код блока. Начальным значением переменной `i` является индекс последнего символа сообщения, а начальным значением переменной `translated` — пустая строка. Затем в цикле к концу этой пустой строки присоединяется значение `message[i]` (а это последний символ сообщения, поскольку значением `i` является последний индекс).

После этого значение `i` декрементируется (т.е. уменьшается) на `1`, а это означает, что теперь значением `message[i]` будет предпоследний символ

сообщения. Таким образом, по мере перемещения `i` как индекса от конца строки сообщения к ее началу строка `message[i]` добавляется в конец строки `translated`. Именно так и получается, что символы строки сообщения сохраняются в переменной `translated` в обратном порядке. Когда `i` в конечном счете станет равно `-1` (это произойдет, когда в сообщении будет достигнут индекс `0`), условие цикла станет равно `False`, после чего выполнение перейдет к строке 12.

```
12. print(translated)
```

В строке 12 (последняя в программе) мы выводим содержимое переменной `translated` (т.е. строку `'.daed era meht fo owt fi ,terces a peek nac eerhT'`) на экран. Благодаря этому пользователь сможет увидеть, что собой представляет обращенная строка.

Если вам все еще не удается понять, каким образом код, выполняющийся в цикле `while`, обращает строку, добавьте в блок цикла следующую новую строку (выделена полужирным шрифтом).

```
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     print('i is', i, ', message[i] is', message[i],
              ', translated is', translated)
11.     i = i - 1
12.
13. print(translated)
```

В строке 10 на экран выводятся значения `i`, `message[i]` и `translated` при каждом прохождении цикла (т.е. на каждой его итерации). На этот раз мы используем вместо конкатенации строк нечто новое. Запятые сообщают функции `print()` о том, что мы выводим шесть различных элементов, поэтому функция добавляет пробелы между ними. Теперь, запустив программу, вы сможете наблюдать за процессом "наращивания" строки, хранящейся в переменной `translated`. Вывод будет выглядеть примерно так.

```
i is 48 , message[i] is . , translated is .
i is 47 , message[i] is d , translated is .d
i is 46 , message[i] is a , translated is .da
i is 45 , message[i] is e , translated is .dae
i is 44 , message[i] is d , translated is .daed
i is 43 , message[i] is   , translated is .daed
i is 42 , message[i] is e , translated is .daed e
i is 41 , message[i] is r , translated is .daed er
i is 40 , message[i] is a , translated is .daed era
i is 39 , message[i] is   , translated is .daed era
```

```
i is 38 , message[i] is m , translated is .daed era m
i is 37 , message[i] is e , translated is .daed era me
i is 36 , message[i] is h , translated is .daed era meh
i is 35 , message[i] is t , translated is .daed era meht
i is 34 , message[i] is   , translated is .daed era meht
i is 33 , message[i] is f , translated is .daed era meht f
i is 32 , message[i] is o , translated is .daed era meht fo
i is 31 , message[i] is   , translated is .daed era meht fo
i is 30 , message[i] is o , translated is .daed era meht fo o
i is 29 , message[i] is w , translated is .daed era meht fo ow
i is 28 , message[i] is t , translated is .daed era meht fo owt
i is 27 , message[i] is   , translated is .daed era meht fo owt
i is 26 , message[i] is f , translated is .daed era meht fo owt f
i is 25 , message[i] is i , translated is .daed era meht fo owt fi
i is 24 , message[i] is   , translated is .daed era meht fo owt fi
i is 23 , message[i] is , , translated is .daed era meht fo owt fi ,
i is 22 , message[i] is t , translated is .daed era meht fo owt fi ,t
i is 21 , message[i] is e , translated is .daed era meht fo owt fi ,te
i is 20 , message[i] is r , translated is .daed era meht fo owt fi ,ter
i is 19 , message[i] is c , translated is .daed era meht fo owt fi ,terc
i is 18 , message[i] is e , translated is .daed era meht fo owt fi ,terce
i is 17 , message[i] is s , translated is .daed era meht fo owt fi ,terces
i is 16 , message[i] is   , translated is .daed era meht fo owt fi ,terces
i is 15 , message[i] is a , translated is .daed era meht fo owt fi ,terces a
i is 14 , message[i] is   , translated is .daed era meht fo owt fi ,terces a
i is 13 , message[i] is p , translated is .daed era meht fo owt fi ,terces a p
i is 12 , message[i] is e , translated is .daed era meht fo owt fi ,terces a pe
i is 11 , message[i] is e , translated is .daed era meht fo owt fi ,terces a pee
i is 10 , message[i] is k , translated is .daed era meht fo owt fi ,terces a peek
i is 9 , message[i] is   , translated is .daed era meht fo owt fi ,terces a peek
i is 8 , message[i] is n , translated is .daed era meht fo owt fi ,terces a peek n
i is 7 , message[i] is a , translated is .daed era meht fo owt fi ,terces a peek na
i is 6 , message[i] is c , translated is .daed era meht fo owt fi ,terces a peek nac
i is 5 , message[i] is   , translated is .daed era meht fo owt fi ,terces a peek nac
i is 4 , message[i] is e , translated is .daed era meht fo owt fi ,terces a peek nac e
i is 3 , message[i] is e , translated is .daed era meht fo owt fi ,terces a peek nac ee
i is 2 , message[i] is r , translated is .daed era meht fo owt fi ,terces a peek nac eer
i is 1 , message[i] is h , translated is .daed era meht fo owt fi ,terces a peek nac eerh
i is 0 , message[i] is T , translated is .daed era meht fo owt fi ,terces a peek nac eerhT
```

Строка вывода `"i is 48 , message[i] is . , translated is ."` показывает, чему равны значения `i`, `message[i]` и `translated` после добавления строки `message[i]` в конец строки `translated`, но до декрементирования `i`. Как видите, при первом проходе через цикл значение `i` устанавливается равным `48`, поэтому `message[i]` (т.е. `message[48]`) представляет собой строку `'.'`. Переменная `translated` вначале содержит пустую строку, но после присоединения к ней значения `message[i]` в строке 9 она становится равна `'.'`.

На следующей итерации цикла выводится текст `"i is 47 , message[i] is d ,translated is .d"`. Мы видим, что значение `i` было декрементировано с `48` до `47`, поэтому теперь `message[i]` — это `message[47]`, т.е.

строка 'd' (вторая буква 'd' в слове 'dead'). Эта буква 'd' добавляется в конец строки, хранящейся в переменной translated, которая теперь содержит значение '.d'.

Мы имеем возможность наглядно убедиться в том, как строка, хранящаяся в переменной translated, медленно "наращивается", постепенно превращаясь из пустой строки в обращенную строку сообщения.

Усовершенствование программы за счет использования функции input()

Все программы, приведенные в книге, построены таким образом, что как шифруемые, так и дешифруемые строки вводятся непосредственно в исходном коде с помощью инструкций присваивания. Такой подход удобен в процессе разработки программы, но вы ведь не думаете, что пользователям будет удобно каждый раз самостоятельно вносить изменения в исходный код? Чтобы упростить применение программы и сделать ее более удобной для пользователей, можете изменить соответствующие инструкции присваивания, включив в них вызов функции input(). При этом можно включить в функцию input() подсказку, предлагающую пользователю ввести строку, подлежащую шифрованию. Например, строку 4 программы *reverseCipher.py* можно заменить следующим образом:

```
4. message = input('Enter message: ')
```

Когда вы запустите программу, она выведет на экран эту подсказку и будет ожидать, пока пользователь не введет сообщение. Введенное пользователем сообщение станет тем строковым значением, которое сохранится в переменной message. Теперь, выполняя программу, вы сможете ввести любую строку, которая будет зашифрована программой и выведена на экран, как показано ниже.

```
Enter message: Hello, world!
!dlrow ,olleH
```

Резюме

Мы только что завершили создание второй программы, которая преобразует предоставленную ей строку в новую строку, используя такие приемы, рассмотренные в главе 3, как индексирование и конкатенация строк. В этой программе важную роль играет функция len(), которая получает

строковый аргумент и возвращает целое число, показывающее, сколько символов содержит данная строка.

Вы также узнали о булевом типе данных, который поддерживает только два возможных значения: `True` и `False`. Операторы сравнения `==`, `!=`, `<`, `>`, `<=` и `>=` могут сравнивать два выражения и возвращать результат в виде булева значения.

Условия — это выражения, содержащие операторы сравнения и возвращающие булевы значения. Они используются в циклах `while`, которые выполняют блок кода, следующий за инструкцией `while`, до тех пор пока условие не станет равно `False`. Блок состоит из строк кода, выделенных отступом одной и той же величины, и может включать вложенные блоки.

Теперь, когда вы уже знаете достаточно много о том, как манипулировать текстом, мы можем приступать к созданию программ, способных взаимодействовать с пользователем. Это очень важно, поскольку текст — основной вид информации, позволяющий пользователю взаимодействовать с компьютером.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Что выведет на экран следующая инструкция?

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

2. Что выведет на экран следующий код?

```
i = 0
while i < 3:
    print('Hello')
    i = i + 1
```

3. А что выведет такой код?

```
i = 0
spam = 'Hello'
while i < 5:
    spam = spam + spam[i]
    i = i + 1
print(spam)
```

4. А такой?

```
i = 0
while i < 4:
    while i < 6:
        i = i + 2
        print(i)
```

5

ШИФР ЦЕЗАРЯ

"Большой брат следит за тобой".
Джордж Оруэлл, "1984"

В главе 1 для реализации шифра Цезаря мы использовали шифровальный диск и таблицы перекодировки букв в цифры. В этой главе мы реализуем шифр Цезаря с помощью компьютерной программы.

Обратный шифр, который мы создали в главе 4, всегда работает одинаковым образом. Однако шифр Цезаря основан на ключах, и сообщение шифруется по-разному в зависимости от того, какой ключ используется. В шифре Цезаря ключи могут иметь значения от 0 до 25. Даже если криптоаналитику известно, что перед ним шифр Цезаря, этой информации еще недостаточно для того, чтобы взломать шифр. Нужно также знать ключ.

В этой главе...

- Инструкция `import`
- Константы
- Цикл `for`
- Инструкции `if`, `else` и `elif`
- Операторы `in` и `not in`
- Строковый метод `find()`

Исходный код программы Caesar Cipher

Введите приведенный ниже код в редакторе файлов и сохраните его в файле *caesarCipher.py*. Затем загрузите модуль *pyperclip.py*, доступный по адресу `https://inventwithpython.com/pyperclip.py`, и поместите его в тот же каталог, в котором находится файл *caesarCipher.py*. Данный модуль будет импортироваться программой *caesarCipher.py*; подробнее об этом мы поговорим в разделе "Импорт модулей и установка переменных".

Когда файлы будут готовы, нажмите клавишу <F5>, чтобы запустить программу. Если столкнетесь с ошибками или другими проблемами, сравните свой код с кодом, приведенным в книге, воспользовавшись онлайн-утилитой Diff по адресу `https://inventwithpython.com/cracking/diff/`.

caesarCipher.py

```
 1. # Шифр Цезаря
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. # Строка, подлежащая шифрованию/дешифрованию
 7. message = 'This is my secret message.'
 8.
 9. # Ключ шифрования/дешифрования
10. key = 13
11.
12. # Установка режима работы: шифрование или дешифрование
13. mode = 'encrypt'  # задать 'encrypt' или 'decrypt'
14.
15. # Все символы, которые могут быть зашифрованы
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
              1234567890 !?.'
17.
18. # Переменная, хранящая зашифрованную/дешифрованную форму сообщения
19. translated = ''
20.
21. for symbol in message:
22.     # Примечание: шифровать/дешифровать можно только символы,
          включенные в строку SYMBOLS
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
25.
26.         # Выполнить шифрование/дешифрование
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
```

```
31.
32.         # Обработать "завертывание", если необходимо
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
37.
38.         translated = translated + SYMBOLS[translatedIndex]
39.     else:
40.         # Присоединить символ без шифрования/дешифрования
41.         translated = translated + symbol
42.
43. # Вывод преобразованной строки
44. print(translated)
45. pyperclip.copy(translated)
```

Пример выполнения программы `Caesar Cipher`

Выполнив программу *caesarCipher.py*, вы должны получить следующий результат:

```
guv6Jv6Jz!J6rp5r7Jzr66ntrM
```

Это строка `'This is my secret message.'`, зашифрованная шифром Цезаря с ключом 13. Программа шифрования, которую вы только что выполнили, автоматически копирует зашифрованную строку в буфер обмена, чтобы ее можно было вставить в электронное письмо или текстовый файл. Таким образом, вы сможете легко переслать зашифрованный текст другому человеку.

При выполнении программы возможно появление следующего сообщения об ошибке.

```
Traceback (most recent call last):
  File "C:\caesarCipher.py", line 4, in <module>
    import pyperclip
ImportError: No module named pyperclip
```

Если вы столкнетесь с таким сообщением, то это, вероятно, будет означать, что модуль *pyperclip.py* не был помещен в нужную папку. Если же файл *pyperclip.py* действительно находится в одной папке с файлом *caesarCipher.py*, но модуль почему-то не работает, закомментируйте код в строках 4 и 45 (содержащих ссылки на модуль `pyperclip`) программы *caesarCipher.py*, поместив перед ними символ #. Это заставит Python игнорировать код, который зависит от модуля *pyperclip.py*, и позволит программе успешно выполниться.

Учтите, что если вы закомментируете указанные строки, то зашифрованный или дешифрованный текст не будет скопирован в буфер обмена в конце программы. Точно так же вы сможете отключать код модуля `pyperclip` в программах, с которыми мы будем работать в последующих главах.

Чтобы дешифровать сообщение, достаточно вставить выведенный текст в качестве нового значения, сохраняемого в переменной `message` в строке 7. Также необходимо изменить инструкцию присваивания в строке 13 таким образом, чтобы в переменной `mode` сохранилась строка `'decrypt'`.

```
 6. # Строка, подлежащая шифрованию/дешифрованию
 7. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 8.
 9. # Ключ шифрования/дешифрования
10. key = 13
11.
12. # Установка режима работы: шифрование или дешифрование
13. mode = 'decrypt'  # задать 'encrypt' или 'decrypt'
```

Теперь, выполнив программу, вы получите следующий результат:

```
This is my secret message.
```

Импорт модулей и установка переменных

Несмотря на то что Python включает множество встроенных функций, некоторые функции хранятся в отдельных программах, называемых *модулями*. Модуль — это программа Python, содержащая дополнительные функции, которые могут использоваться в других программах. Модули импортируются с помощью инструкции `import`, в которой указывается имя импортируемого модуля.

Инструкция импорта содержится в строке 4.

```
1. # Шифр Цезаря
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
```

В данном случае мы импортируем модуль `pyperclip`, что дает нам возможность впоследствии вызвать в программе функцию `pyperclip.copy()`, которая будет автоматически копировать строки в буфер обмена, откуда их будет удобно вставлять в другие программы.

В нескольких следующих строках программы *caesarCipher.py* устанавливаются значения трех переменных.

```
 6. # Строка, подлежащая шифрованию/дешифрованию
 7. message = 'This is my secret message.'
 8.
 9. # Ключ шифрования/дешифрования
10. key = 13
11.
12. # Установка режима работы: шифрование или дешифрование
13. mode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

В переменной `message` сохраняется строка, которую необходимо зашифровать или дешифровать, а в переменной `key` — целочисленное значение ключа шифрования. В переменной `mode` хранится строка, определяющая режим работы программы: строка `'encrypt'` задает шифрование сообщения, строка `'decrypt'` — дешифрование.

Константы и переменные

Константы — это переменные, значения которых не должны изменяться во время работы программы. Например, нашей программе шифрования нужна строка, содержащая все допустимые символы, которые могут быть зашифрованы с помощью шифра Цезаря. Поскольку данная строка не должна изменяться, мы сохраняем ее в константе `SYMBOLS` в строке 16.

```
15. # Все символы, которые могут быть зашифрованы
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
              1234567890 !?.'
```

Символ — это общий термин, употребляемый в криптографии по отношению к одиночному знаку, который можно зашифровать или дешифровать. *Символьный набор* — это совокупность всех возможных символов, которые можно шифровать или дешифровать с помощью определенного шифра. Поскольку символьный набор используется в программе неоднократно и мы не хотим вводить его заново каждый раз, когда он понадобится (это чревато ошибками), мы сохраняем весь символьный набор в константе `SYMBOLS`, которую задаем один раз.

Обратите внимание на то, что все буквы в имени `SYMBOLS` — прописные, что является общепринятым соглашением для имен констант. И хотя *ничто не мешает нам* изменить значение переменной `SYMBOLS`, использование в ее имени исключительно прописных букв служит напоминанием о том, что ее значение не должно меняться.

Как и в случае любого другого общепринятого соглашения, мы *не обязаны* строго следовать данному правилу. Однако его соблюдение упрощает

другим программистам понимание того, как используются подобные переменные. (Это поможет даже вам, если вы вернетесь к работе с программой после длительного перерыва.)

В строке 19 программа сохраняет пустую строку в переменной `translated`, которая впоследствии будет использоваться для хранения зашифрованного или дешифрованного сообщения.

```
18. # Сохранить зашифрованную/дешифрованную форму сообщения
19. translated = ''
```

Как и в случае программы обратного шифрования, рассмотренной в главе 4, к моменту завершения работы программы переменная `translated` будет содержать полное зашифрованное (или дешифрованное) сообщение. Но ее начальным значением является пустая строка.

Цикл `for`

В строке 21 мы начинаем цикл `for`.

```
21. for symbol in message:
```

Вспомните, что выполнение цикла `while` продолжается до тех пор, пока условие цикла равно `True`. Цикл `for` имеет несколько иное назначение и не содержит условия, в отличие от цикла `while`. Вместо этого он проходит по строке или набору значений. На рис. 5.1 представлены шесть элементов этого цикла.

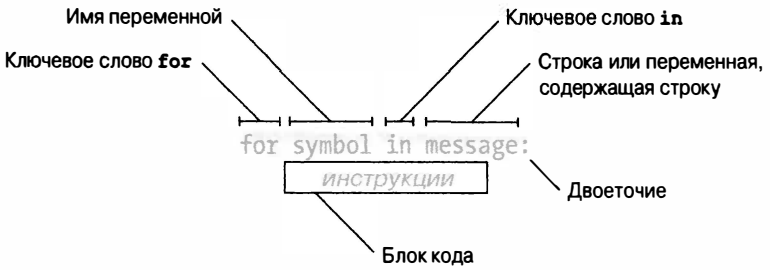


Рис. 5.1. *Шесть элементов цикла* `for`

Каждый раз, когда программа заходит в цикл (т.е. начинает очередную итерацию), переменная цикла (в строке 21 ее именем становится `symbol`) получает значение в виде следующего символа указанной строки (в данном случае строки `message`). Инструкция `for` аналогична инструкции присваивания в том смысле, что также создает переменную и присваивает ей зна-

чение, однако отличие в том, что переменной поочередно присваиваются различные значения, перебираемые в цикле.

Пример цикла `for`

Выполните в интерактивной оболочке приведенный ниже пример. Обратите внимание на то, что после ввода первой строки подсказка >>> исчезнет (вместо нее отобразится многоточие ...), поскольку интерпретатор ожидает, что вслед за двоеточием в инструкции `for` будет введен блок кода. В интерактивной оболочке блок будет считаться завершенным, когда вы введете пустую строку.

```
>>> for letter in 'Howdy':
...     print('The letter is ' + letter)
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

Цикл проходит по всем символам строки `'Howdy'`. В процессе этого значением переменной `letter` поочередно становится каждый из символов строки `'Howdy'`. Чтобы увидеть, как это работает, мы вставили в цикл код, который выводит значение переменной `letter` на каждой итерации.

Цикл `while`, эквивалентный циклу `for`

Цикл `for` в чем-то напоминает цикл `while`. Если вам нужно лишь пройти по всем символам строки, то цикл `for` более эффективен. То же самое можно сделать и с помощью цикла `while`, однако для этого потребуется немного больше кода.

```
❶ >>> i = 0
❷ >>> while i < len('Howdy'):
❸ ...     letter = 'Howdy'[i]
❹ ...     print('The letter is ' + letter)
❺ ...     i = i + 1
  ...
  The letter is H
  The letter is o
  The letter is w
  The letter is d
  The letter is y
```

Заметьте, что цикл `while` работает так же, как и цикл `for`, просто немного сложнее и длиннее. Сначала мы устанавливаем новую переменную `i` в `0` перед началом цикла `while` (❶). В цикле задается условие, равное `True` до тех пор, пока переменная `i` меньше длины строки `'Howdy'` (❷). Поскольку `i` — целое число, предназначенное для отслеживания текущей позиции в строке, мы должны объявить отдельную переменную `letter` для хранения символа, занимающего в строке позицию `i` (❸). После этого мы можем вывести текущее значение переменной, получая тот же результат, что и в случае цикла `for` (❹). По завершении итерации мы должны инкрементировать переменную `i` путем прибавления к ней `1` для того, чтобы перейти в следующую позицию (❺).

Чтобы понять смысл строк 23 и 24 программы *caesarCipher.py*, вам нужно узнать кое-что об инструкциях `if`, `elif` и `else`, а также об операторах `in` и `not in` и строковом методе `find()`, которым посвящены следующие разделы.

Инструкция `if`

Строка 23 содержит инструкцию другого рода — `if`.

```
23. if symbol in SYMBOLS:
```

Ее можно интерпретировать следующим образом: "Если указанное условие равно `True`, выполнить следующий блок. В противном случае пропустить этот блок". Инструкция `if` состоит из ключевого слова `if`, за которым следует условие, заканчивающееся двоеточием (`:`). Подлежащий выполнению код выделяется отступами в виде блока, как и в циклах.

Пример инструкции `if`

Обратимся к примеру, в котором используется инструкция `if`. Откройте новое окно в редакторе файлов, введите следующий код и сохраните его в файле *checkPw.py*.

checkPw.py

```
  print('Enter your password.')
❶ typedPassword = input()
❷ if typedPassword == 'swordfish':
❸     print('Access Granted')
❺ print('Done')
```

Когда вы запустите эту программу, она отобразит текст `'Enter your password. '` и предоставит пользователю возможность ввести пароль, который затем сохраняется в переменной `typedPassword` (❶). Далее инструкция `if` проверяет, совпадает ли введенный пароль со строкой `'swordfish'` (❷). Если это так, выполнение переходит в блок, следующий за инструкцией `if`, в котором отображается текст `'Access Granted'` (❸). В противном случае, если переменная `typedPassword` не равна `'swordfish'`, блок инструкции `if` пропускается. Так или иначе, программа переходит к строке, следующей за блоком `if`, в которой отображается текст `'Done'` (❹).

Инструкция `else`

Часто возникает необходимость проверить условие и выполнить один блок, если условие истинно (равно `True`), и другой, если условие ложно (равно `False`). Для этого предназначена инструкция `else`, следующая за блоком инструкции `if`. Блок кода инструкции `else` будет выполнен в том случае, если условие инструкции `if` ложно. Что касается самой инструкции `else`, то она записывается в виде ключевого слова `else`, за которым следует двоеточие (`:`). Она не нуждается в условии, поскольку будет выполняться, если условие инструкции `if` не является истинным. Подобный код можно интерпретировать следующим образом: "Если условие истинно, выполнить этот блок, иначе выполнить другой блок".

Измените программу *checkPw.py* следующим образом (новые строки выделены полужирным шрифтом):

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
      print('Access Granted')
  else:
❷     print('Access Denied')
❸ print('Done')
```

Данная версия программы работает почти так же, как предыдущая. Текст `'Access Granted'` по-прежнему будет отображаться, если условие инструкции `if` будет равно `True` (❶). Но теперь, если введенный пользователем текст будет отличаться от `'swordfish'` и условие инструкции `if` окажется равно `False`, выполнение перейдет к блоку инструкции `else`, который отобразит текст `'Access Denied'` (❷). В любом случае выполнение программы завершится выводом текста `'Done'` (❸).

Инструкция elif

С инструкцией if может соседствовать еще одна инструкция: elif. Она тоже содержит условие. Как и else, она следует за инструкцией if (или другой инструкцией elif) и выполняется в том случае, если условие предыдущей инструкции if (или elif) равно False. Инструкции if, elif и else можно интерпретировать следующим образом: "Если данное условие равно True, выполнить этот блок. В противном случае проверить, не является ли истинным следующее условие. Иначе необходимо выполнить только этот последний блок". За инструкцией if может следовать любое количество инструкций elif. Вновь измените программу *checkPw.py*, придав ей следующий вид.

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
❷     print('Access Granted')
❸ elif typedPassword == 'mary':
      print('Hint: the password is a fish.')
❹ elif typedPassword == '12345':
      print('That is a really obvious password.')
else:
    print('Access Denied')
print('Done')
```

Эта программа содержит четыре блока, связанных с инструкциями if, elif и else. Если пользователь введет '12345', то условие typedPassword == 'swordfish' станет равно False (❶), поэтому первый блок, содержащий инструкцию print('Access Granted') (❷), будет пропущен. Следующим проверяется условие typedPassword == 'mary', которое тоже равно False (❸), поэтому второй блок также пропускается. Зато условие typedPassword == '12345' оказывается истинным (❹), поэтому выполнение передается блоку, следующему за этой инструкцией elif, и в результате выполнится код print('That is a really obvious password.'), причем остальные инструкции elif и else будут пропущены. *Обратите внимание на то, что выполнен будет один и только один из этих блоков.*

Вслед за инструкцией if может следовать произвольное количество инструкций elif, или же они вообще могут отсутствовать. Инструкция else либо отсутствует, либо бывает только одна, причем она всегда должна быть последней, поскольку выполняется только в том случае, когда ни одно из условий не равно True. Выполняется блок той инструкции, условие кото-

рой первой оказалось истинным. Оставшиеся условия (даже если они тоже равны `True`) не проверяются.

Операторы `in` и `not in`

В строке 23 программы *caesarCipher.py* встречается также оператор `in`.

```
23.     if symbol in SYMBOLS:
```

У него два строковых операнда. Оператор возвращает `True`, если первая строка содержится во второй, или `False` в противном случае. Если к оператору `in` добавить ключевое слово `not`, то это приведет к получению противоположного результата. Введите в интерактивной оболочке следующие выражения.

```
>>> 'hello' in 'hello world!'
True
>>> 'hello' not in 'hello world!'
False
>>> 'ello' in 'hello world!'
True
❶ >>> 'HELLO' in 'hello world!'
False
❷ >>> '' in 'Hello'
True
```

Обратите внимание на то, что операторы `in` и `not in` чувствительны к регистру (❶). Кроме того, считается, что пустая строка входит в любую другую строку (❷).

Выражения с операторами `in` и `not in` удобно использовать в качестве условий в инструкциях `if` для выполнения кода в том случае, если одна строка содержится в другой.

Вернемся к программе *caesarCipher.py*. В строке 23 проверяется, содержится ли символ, хранящийся в переменной `symbol` (ее значение устанавливается в цикле `for` в строке 21 равным одиночному символу из строки сообщения), в константе `SYMBOLS` (в ней перечислен весь набор символов, которые могут быть зашифрованы или дешифрованы с помощью данной программы). Если символ `symbol` обнаружен в наборе `SYMBOLS`, то выполнение передается следующему блоку, который начинается в строке 24. В противном случае этот блок игнорируется, и выполнение передается блоку, следующему за строкой 39, которая содержит инструкцию `else`. В зависимости от того, входит ли конкретный символ в заданный символьный набор или не входит, программа шифрования будет выполнять разный код.

Строковый метод find()

В строке 24 определяется индекс строки SYMBOLS, соответствующий заданному символу.

```
24.         symbolIndex = SYMBOLS.find(symbol)
```

Здесь содержится вызов метода. *Метод* — это та же функция, но его имя присоединяется к значению (или, как в строке 24, к переменной, содержащей значение) с помощью оператора-точки (.). Имя этого метода — find(), и он вызывается для строкового значения, содержащегося в константе SYMBOLS.

Методы есть у большинства типов данных (таких, как строки). Метод find() получает строку в качестве аргумента и возвращает целочисленный индекс позиции аргумента в строке, для которой вызывается метод. Введите в интерактивной оболочке следующие команды.

```
>>> 'hello'.find('e')
1
>>> 'hello'.find('o')
4
>>> spam ='hello'
>>> spam.find('h')
❶ 0
```

Метод find() можно вызывать для любой строки или переменной, содержащей строковое значение. Помните, что нумерация индексов в Python начинается с 0, поэтому если индекс, возвращаемый методом find(), относится к первому символу в строке, то он будет равен 0 (❶).

Если строковый аргумент не удается найти, то метод find() возвращает значение -1. Введите в интерактивной оболочке следующие команды.

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

Заметьте, что метод find() также чувствителен к регистру символов (❶).

Строка, которую вы передаете методу find() в качестве аргумента, может содержать более одного символа. В таком случае целочисленным значением, возвращаемым методом find(), будет индекс первого символа позиции, в которой найден данный аргумент. Введите в интерактивной оболочке следующие команды.

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

Строковый метод `find()` напоминает более специализированную версию оператора `in`. Он сообщает не только о том, содержится ли одна строка в другой, но и том, начиная с какой именно позиции она встречается.

Шифрование и дешифрование символов

Теперь, когда мы разобрались в том, как работают инструкции `if`, `elif` и `else`, а также оператор `in` и строковый метод `find()`, вам будет проще понять, как работает оставшаяся часть программы шифрования на основе шифра Цезаря.

Программа способна шифровать и дешифровать только символы, входящие в состав заданного символьного набора.

```
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
```

Поэтому, прежде чем выполнить код в строке 24, программа должна определить, содержится ли символ в поддерживаемом наборе. После этого она определяет индекс строки `SYMBOLS`, начиная с которого располагается строка `symbol`. Индекс, возвращаемый методом `find()`, сохраняется в переменной `symbolIndex`.

Располагая индексом текущего символа, сохраненным в переменной `symbolIndex`, мы можем применить к нему операции шифрования/дешифрования. Шифр Цезаря добавляет значение ключа к индексу символа при его шифровании или вычитает значение ключа из индекса символа при его дешифровании. Полученное значение сохраняется в переменной `translatedIndex`, поскольку оно будет служить индексом в строке `SYMBOLS`, соответствующим преобразованному символу.

```
26.         # Выполнить шифрование/дешифрование
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
```

Переменная mode содержит строку, которая сообщает программе, что именно необходимо сделать с сообщением: зашифровать или дешифровать. Если переменная равна 'encrypt', то условие в строке 27 равно True, что приведет к выполнению строки 28, которая прибавит значение ключа к значению symbolIndex (блок, следующий за инструкцией elif, будет пропущен). В противном случае, если переменная mode равна 'decrypt', будет выполнена строка 30, которая вычтет значение ключа.

Обработка "завертывания" символьного набора

Когда мы реализовывали шифр Цезаря с помощью карандаша и бумаги в главе 1, то в некоторых случаях добавление или вычитание ключа приводило к значениям, превышающим или равным размеру символьного набора или даже меньшим нуля. В подобных случаях мы должны добавлять или вычитать длину символьного набора, чтобы автоматически возвращаться к началу или концу символьного набора. С этой целью мы используем вызов len(SYMBOLS), который возвращает значение 66, т.е. длину строки SYMBOLS. Обработка "завертывания" осуществляется в строках 33–36.

```
32.         # Обработать "завертывание", если необходимо
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
```

Если значение переменной translatedIndex превышает или равно 66, то условие в строке 33 оказывается истинным, и выполняется строка 34 (тогда как инструкция elif в строке 35 пропускается). Вычитание длины строки SYMBOLS из значения translatedIndex переводит индекс переменной обратно в начало строки SYMBOLS. В противном случае Python проверяет, не меньше ли нуля значение переменной translatedIndex. Если это так, то выполняется строка 36, и переменная translatedIndex "завертывается" вокруг конца строки SYMBOLS.

Возможно, вас удивляет, почему мы использовали вызов len(SYMBOLS), а не непосредственно значение 66. Это позволяет нам добавлять или удалять символы из набора SYMBOLS, не нарушая работу остальной части программы.

Теперь, когда в переменной translatedIndex хранится индекс преобразованного символа, мы можем получить сам символ с помощью выражения SYMBOLS[translatedIndex]. В строке 38 этот зашифрованный/дешифрованный символ добавляется в конец строки translated путем конкатенации строк:

```
38.             translated = translated + SYMBOLS[translatedIndex]
```

В конечном счете строка `translated` будет содержать полное зашифрованное или дешифрованное сообщение.

Обработка символов, не включенных в символьный набор

Строка `message` может содержать символы, отсутствующие в строке `SYMBOLS`. Эти символы не входят в символьный набор программы шифрования и не могут быть зашифрованы или дешифрованы. Вместо этого они будут непосредственно присоединяться к строке `translated`, что и происходит в строках 39–41.

```
39.     else:
40.         # Присоединить символ без шифрования/дешифрования
41.         translated = translated + symbol
```

Инструкция `else` в строке 39 выделена отступом величиной четыре пробела. Взглянув на расположенные выше строки, вы увидите, что эта строка имеет тот же отступ, что и строка 23. Несмотря на то что между этими двумя инструкциями `if` и `else` находится большое количество строк кода, весь этот код принадлежит к одному блоку.

Если условие инструкции `if` в строке 23 равно `False`, соответствующий блок игнорируется, и выполнение передается блоку инструкции `else`, начинающемуся в строке 41. Этот блок состоит всего лишь из одной инструкции, в которой символ присоединяется без какого-либо изменения в конец строки `translated`. В результате символы, не входящие в заданный символьный набор, такие как `'%'` или `'('`, добавляются в строку `translated`, не подвергаясь шифрованию или дешифрованию.

Вывод и копирование преобразованной строки

Строка 43 не имеет отступа, а это означает, что она является первой строкой, следующей за блоком, который начинается в строке 21 (блок цикла `for`). К тому времени, когда выполнение программы достигнет строки 44, цикл обработает все символы строки сообщения, т.е. зашифрует (или дешифрует) каждый из них и добавит в строку `translated`.

```
43. # Вывод преобразованной строки
44. print(translated)
45. pyperclip.copy(translated)
```

В строке 44 преобразованная строка выводится на экран с помощью функции `print()`. Обратите внимание на то, что это единственный вызов функции `print()` во всей программе. Компьютер выполняет большой объем работы, шифруя каждую букву сообщения, а также обрабатывая "завертывание" строк и небуквенные символы. Однако пользователь не должен всего этого знать, ему достаточно видеть только окончательную строку, сохраненную в переменной `translated`.

В строке 45 вызывается функция `copy()`, которая получает один строковый аргумент и копирует его в буфер обмена. Поскольку `copy()` — это функция из модуля `pyperclip`, мы должны сообщить об этом Python, поместив префикс `pyperclip.` перед именем функции. Если вы введете `copy(translated)` вместо `pyperclip.copy(translated)`, то Python выдаст сообщение об ошибке, поскольку не сможет найти такую функцию.

Сообщение об ошибке появится и в том случае, если вы забудете использовать инструкцию `import pyperclip` (строка 4), прежде чем пытаться выполнить вызов `pyperclip.copy()`.

Мы полностью проанализировали программу шифрования на основе шифра Цезаря. Когда вы запустите ее, обратите внимание на то, что компьютер проделывает всю сложную работу менее чем за секунду. Даже если вы сохраните в переменной `message` очень длинную строку, компьютер справится с шифрованием или дешифрованием сообщения не более чем за пару секунд. Сравните это с теми несколькими минутами, которые понадобились бы вам для того, чтобы воспользоваться шифровальным диском. Но наша программа еще и автоматически копирует зашифрованный текст в буфер обмена, чтобы пользователь мог просто вставить его в электронное письмо и отправить кому-нибудь.

Шифрование других символов

Одной из проблем реализованной нами программы шифрования является то, что она не умеет шифровать символы, не входящие в заданный символьный набор. Например, если вы зашифруете строку `'Be sure to bring the $$$.'` с помощью ключа 20, то сообщение будет преобразовано в строку `'VyQ?A!yQ.9Qv!381Q.2yQ$$$T'`. Здесь не скрывается тот факт, что в сообщении упоминаются доллары (`$$$`). Однако программу можно видоизменить таким образом, чтобы она справлялась и с другими символами.

Изменив строку, сохраняемую в константе `SYMBOLS`, и включив в нее дополнительные символы, мы добьемся того, что программа будет шифровать эти символы тоже, поскольку теперь вычисление условия `if`

`symbol in SYMBOLS` в строке 23 будет давать результат `True`. Значением `symbolIndex` будет индекс символа `symbol` в этой новой, увеличенной строке `SYMBOLS`. Операция "завертывания" должна будет добавлять и вычитать количество символов, соответствующее новой строке, но это уже учтено, поскольку мы использовали вызов `len(SYMBOLS)`, а не ввели `66` непосредственно в коде.

В частности, строку 16 можно переписать так:

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890
          !?.`~@#$%^&*()_+-=[]{}|;:<>,/'
```

Имейте в виду, что, для того чтобы все это работало, шифрование и дешифрование сообщения должны выполняться с использованием одного и того же символьного набора.

Резюме

Вы уже прочитали несколько глав книги и изучили ряд концепций программирования, но теперь у вас есть программа, которая реализует тайный шифр. И что более важно, вы понимаете, как работает такая программа.

В Python модули — это программы, которые содержат полезные функции. Чтобы задействовать расширенные функции, сначала их нужно импортировать с помощью инструкции `import`. Чтобы вызвать функцию, содержащуюся в модуле, поместите имя модуля вместе с оператором-точкой перед именем функции: *модуль*`.`*функция*`()`.

В соответствии с общепринятым соглашением константы записываются в коде с использованием прописных букв. Предполагается, что значения этих переменных не должны изменяться программой (хотя ничто не запрещает программистам это делать). Константы полезны тем, что позволяют присваивать имена конкретным значениям.

Методы — это функции, присоединяемые к значению определенного типа данных. Строковый метод `find()` возвращает целое число, указывающее позицию переданного методу строкового аргумента в строке, для которой он вызывается.

Вы изучили ряд программных конструкций, позволяющих управлять тем, какие строки кода и сколько раз должны быть выполнены. Цикл `for` проходит по всем символам строкового значения, записывая в переменную цикла каждый следующий символ в качестве нового значения на очередной итерации. Инструкции `if`, `elif` и `else` выполняют блоки кода на основании того, какое значение имеет заданное условие: `True` или `False`.

Операторы `in` и `not in` проверяют, содержится ли одна строка в другой, возвращая соответственно значение `True` или `False`.

Знание программирования позволит вам выполнять шифрование или дешифрование сообщений с помощью шифра Цезаря, используя язык, понятный компьютеру. И коль скоро компьютер понимает, какие действия он должен выполнить, он делает это гораздо быстрее человека, причем не допуская ошибок (если только ошибки не вкрались в саму программу). Несмотря на необычайную полезность шифра Цезаря, оказывается, что его нетрудно взломать тому, кто знаком с программированием. В главе 6 вы воспользуетесь полученными навыками, чтобы написать программу взлома шифра Цезаря. Это позволит вам читать тексты, зашифрованные другими людьми. Итак, нам предстоит научиться взламывать шифры!

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Используйте программу *caesarCipher.py* для шифрования следующих сообщений с помощью указанных ключей.

   А. `'"You can show black is white by argument," said Filby, "but you will never convince me."'` (ключ 8).

   Б. `'1234567890'` (ключ 21).

2. Используйте программу *caesarCipher.py* для дешифрования следующих зашифрованных сообщений с помощью указанных ключей.

   А. `'Kv?uqwpfu?rncwukdng?gpqwijB'` (ключ 2).

   Б. `'XCBSw88S18A1S 2SB41SE .8zSEwAS50D5A5x81V'` (ключ 22).

3. Какая инструкция Python будет импортировать модуль *watermelon.py*?

4. Что будут выводить на экран следующие фрагменты кода?

   А.

```
spam = 'foo'
for i in spam:
    spam = spam + i
print(spam)
```

Б.

```
if 10 < 5:
    print('Hello')
elif False:
    print('Alice')
elif 5 != 5:
    print('Bob')
else:
    print('Goodbye')
```

В.

```
print('f' not in 'foo')
```

Г.

```
print('foo' in 'f')
```

Д.

```
print('hello'.find('oo'))
```

6

ВЗЛОМ ШИФРА ЦЕЗАРЯ МЕТОДОМ ГРУБОЙ СИЛЫ

"Арабские ученые... изобрели криптоанализ, науку расшифровки сообщений без знания ключа".

Саймон Сингх, "Книга шифров"

Мы можем взломать шифр Цезаря, используя криптоаналитическую методику под названием *метод грубой силы* (brute force). Он предполагает дешифрование сообщения путем полного перебора всех возможных значений ключей. Ничто не мешает криптоаналитику взять какой-то один ключ, применить его для дешифрования зашифрованного текста, проанализировать результат и, если расшифровать секретное сообщение не удалось, перейти к использованию следующего ключа. Учитывая высокую эффективность метода грубой силы в отношении взлома шифра Цезаря, вам не следует применять данный шифр на практике для шифрования секретных сообщений.

В идеальном случае зашифрованный текст никогда не должен попадать в руки посторонних. Однако *принцип Керкгоффса* (названный так в честь криптографа Огюста Керкгоффса, жившего в XIX веке) утверждает, что шифр должен сохранять свою способность быть секретным даже тогда, когда любому человеку известно, как он работает, и зашифрованный текст оказывается в руках тех, для кого он не предназначен. В XX веке этот принцип был сформулирован математиком Клодом Шенноном в виде следую-

щего постулата (*максима Шеннона*): "Враг знает систему". Частью шифра, сохраняющей секретность сообщения, является ключ, а в случае шифра Цезаря эту информацию можно легко раскрыть.

В этой главе...

- Принцип Керкгоффса и максима Шеннона
- Метод грубой силы
- Функция `range()`
- Форматирование строк

Исходный код программы `Caesar Hacker`

Откройте в файловом редакторе новое окно, выбрав пункты меню File⇨New File. Введите в этом окне приведенный ниже код и сохраните его в файле *caesarHacker.py*.

Когда файл будет готов, нажмите клавишу <F5>, чтобы запустить программу. Если столкнетесь с ошибками или другими проблемами, сравните свой код с кодом, приведенным в книге, воспользовавшись онлайн-утилитой Diff по адресу `https://inventwithpython.com/cracking/diff/`.

caesarHacker.py

```
 1. # Программа взлома шифра Цезаря
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
              1234567890 !?.'
 6.
 7. # Цикл по всем возможным значениям ключа
 8. for key in range(len(SYMBOLS)):
 9.     # Важно присвоить пустую строку переменной translated,
10.     # чтобы очистить ее от значения из предыдущей итерации
11.     translated = ''
12.
13.     # Остальная часть программы почти не изменилась
14.
15.     # Цикл по всем символам сообщения
16.     for symbol in message:
17.         if symbol in SYMBOLS:
18.             symbolIndex = SYMBOLS.find(symbol)
```

```
            translatedIndex = symbolIndex - key

            # Обработка "заворачивания"
            if translatedIndex < 0:
                translatedIndex = translatedIndex + len(SYMBOLS)

            # Присоединить дешифрованный символ
            translated = translated + SYMBOLS[translatedIndex]

        else:
            # Присоединить символ без шифрования/дешифрования
            translated = translated + symbol

    # Отобразить каждый возможный вариант расшифровки
    print('Key #%s: %s' % (key, translated))
```

Обратите внимание на то, что значительная часть программы совпадает с исходной программой шифрования на основе шифра Цезаря. Это объясняется тем, что программа взлома шифра Цезаря использует те же действия для дешифрования сообщений.

Пример выполнения программы `Caesar Hacker`

Выполнив программу взлома шифра Цезаря, вы должны получить приведенный ниже результат. Программа взламывает зашифрованный текст `guv6Jv6Jz!J6rp5r7Jzr66ntrM`, поочередно используя все 66 возможных значений ключей.

```
Key #0: guv6Jv6Jz!J6rp5r7Jzr66ntrM
Key #1: ftu5Iu5Iy I5qo4q6Iyq55msqL
Key #2: est4Ht4Hx0H4pn3p5Hxp44lrpK
Key #3: drs3Gs3Gw9G3om2o4Gwo33kqoJ
Key #4: cqr2Fr2Fv8F2nl1n3Fvn22jpnI
--пропущено--
Key #11: Vjku?ku?o1?ugetgv?oguucigB
Key #12: Uijt!jt!nz!tfdsfu!nfttbhfA
Key #13: This is my secret message.
Key #14: Sghr0hr0lx0rdbqds0ldrrZfd?
Key #15: Rfgq9gq9kw9qcapcr9kcqqYec!
--пропущено--
Key #61: lz1 O1 O5CO wu0w!O5w sywR
Key #62: kyz0Nz0N4BN0vt9v N4v00rxvQ
Key #63: jxy9My9M3AM9us8u0M3u99qwuP
Key #64: iwx8Lx8L2.L8tr7t9L2t88pvtO
Key #65: hvw7Kw7K1?K7sq6s8K1s77ousN
```

Поскольку дешифрованный результат для ключа 13 представляет собой простой текст, мы видим, что оригинальным ключом шифрования должен быть ключ 13.

Установка переменных

В программе взлома создается переменная `message` для хранения зашифрованной строки, подлежащей дешифрованию. Константа `SYMBOLS` содержит полный набор символов, поддерживаемых программой шифрования.

```
1. # Программа взлома шифра Цезаря
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
5. SYMBOLS ='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
     1234567890 !?.'
```

Строка `SYMBOLS` должна быть такой же, как и константа `SYMBOLS` в программе шифрования, применявшейся для шифрования текста, который мы пытаемся дешифровать. Если это условие не будет соблюдено, программа взлома не сработает. Обратите внимание на то, что в этой строке между символами `0` и `!` содержится одиночный пробел.

Организация цикла с помощью функции `range()`

В строке 8 начинается цикл `for`, который проходит не по строке, а по значению, возвращаемому функцией `range()`.

```
7. # Цикл по всем возможным значениям ключа
8. for key in range(len(SYMBOLS)):
```

Функция `range()` имеет один целочисленный аргумент и возвращает значение типа `range` (диапазон). Значения этого типа могут использоваться в циклах для выполнения определенного количества итераций. Рассмотрим несколько примеров. Введите в интерактивной оболочке следующий код.

```
>>> for i in range(3):
...   print('Hello')
...
Hello
Hello
Hello
```

Цикл `for` выполняется три раза, поскольку мы передали функции `range()` число `3`.

Функция `range()` будет присваивать переменной цикла целочисленные значения в диапазоне от `0` до значения аргумента (но не включая его). Например, введите в интерактивной оболочке следующий код.

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
```

Здесь переменной `i` присваиваются значения от `0` до (но не включая его) `6`, аналогично тому, как это делается в строке 8 программы *caesarHacker.py*. В этой строке переменной `key` присваиваются значения от `0` до (но не включая его) `66`. Вместо жесткого кодирования значения `66` мы используем результат, возвращаемый вызовом `len(SYMBOLS)`, чтобы программа могла нормально работать даже в случае изменения строки `SYMBOLS`.

Когда программа впервые проходит через этот цикл, переменная `key` устанавливается равной `0`, и зашифрованный текст сообщения дешифруется с ключом `0`. (Разумеется, если `0` не является исходным ключом, то сообщение будет дешифровано в бессмыслицу.) Код в цикле `for`, занимающий строки с 9 по 31, аналогичен коду исходной программы шифрования на основе шифра Цезаря и выполняет всю работу по дешифрованию. На следующей итерации в строке 8 цикла `for` ключ устанавливается равным `1`.

Хоть мы и не используем в программе эту возможность, функции `range()` разрешается передавать не один аргумент, а два. Первый аргумент задает начало диапазона, второй — его конец (который сам не включается в диапазон). Аргументы разделяются запятой.

```
>>> for i in range(2, 6):
...     print(i)
...
2
3
4
5
```

Переменная `i` будет иметь значения от `2` до `5` (но не включая `6`).

Дешифрование сообщения

В теле цикла программа добавляет дешифрованный текст в конец строки, хранящейся в переменной `translated`. В строке 11 значение `translated` устанавливается равным пустой строке.

```
 7. # Цикл по всем возможным значениям ключа
 8. for key in range(len(SYMBOLS)):
 9.     # Важно присвоить пустую строку переменной translated,
10.     # чтобы очистить ее от значения из предыдущей итерации
11.     translated = ''
```

Очень важно то, что в начале цикла `for` мы каждый раз переустанавливаем переменную `translated` в пустую строку. Если этого не делать, то текст, дешифрованный с помощью текущего ключа, будет добавлен к хранящемуся в переменной `translated` дешифрованному тексту, оставшемуся после выполнения предыдущей итерации цикла.

Строки с 16 по 30 почти совпадают с аналогичными строками программы шифрования на основе шифра Цезаря, рассмотренной в главе 5, но они немного проще, поскольку предназначены только для дешифрования текста.

```
13.     # Остальная часть программы почти не изменилась
14.
15.     # Цикл по всем символам сообщения
16.     for symbol in message:
17.         if symbol in SYMBOLS:
18.             symbolIndex = SYMBOLS.find(symbol)
```

В строке 16 мы организуем цикл по всем символам зашифрованной строки, хранящейся в переменной `message`. На каждой итерации цикла в строке 17 проверяется, входит ли текущий символ в набор, хранящийся в константе `SYMBOLS`, и, если это так, символ дешифруется. В строке 18 с помощью метода `find()` мы находим индекс символа в строке `SYMBOLS` и сохраняем его в переменной `symbolIndex`.

Затем мы вычитаем значение ключа из переменной `symbolIndex` в строке 19.

```
19. .             translatedIndex = symbolIndex - key
20.
21.             # Обработка заворачивания
22.             if translatedIndex < 0:
23.                translatedIndex = translatedIndex + len(SYMBOLS)
```

В результате такого вычитания значение translatedIndex может оказаться меньшим нуля, что потребует от нас обработать "заворачивание" строки, хранящейся в константе SYMBOLS, для определения позиции, соответствующей дешифрованному символу. Необходимость этой операции проверяется в строке 22, и если значение translatedIndex меньше нуля, то в строке 23 к нему прибавляется 66 (т.е. значение, возвращаемое вызовом len(SYMBOLS)).

Дешифрованный символ определяется выражением SYMBOLS [translatedIndex]. В строке 26 этот символ добавляется в конец строки, хранящейся в переменной translated.

```
25.             # Присоединить дешифрованный символ
26.             translated = translated + SYMBOLS[translatedIndex]
27.
28.         else:
29.             # Присоединить символ без шифрования/дешифрования
30.             translated = translated + symbol
```

В строке 30 символ, не найденный в символьном наборе SYMBOLS, добавляется в неизменном виде в конец строки, хранящейся в переменной translated.

Использование строкового форматирования для отображения ключа и дешифрованных сообщений

Несмотря на то что в строке 33 содержится лишь один вызов функции print(), эта функция будет выполняться множество раз, поскольку она вызывается на каждой итерации цикла for, создаваемого в строке 8.

```
32.     # Отобразить каждый возможный вариант расшифровки
33.     print('Key #%s: %s' % (key, translated))
```

Аргументом функции print() является строковое значение, в котором используется *строковое форматирование* (*интерполяция*). Параметр %s служит для включения одной строки в другую. Первые символы %s в строке заменяются первым из значений, заключенных в круглые скобки в конце строки.

Введите следующие инструкции в интерактивной оболочке.

```
>>> 'Hello %s!' % ('world')
'Hello world!'
>>> 'Hello ' + 'world' + '!'
'Hello world!'
>>> 'The %s ate the %s that ate the %s.' % ('dog', 'cat', 'rat')
'The dog ate the cat that ate the rat.'
```

В данном примере первая строка, `'world'`, вставляется в строку `'Hello %s!'` вместо символов `%s`. Это работает так, словно вы конкатенируете часть строки, предшествующую символам `%s`, с интерполированной строкой и частью строки, следующей за символами `%s`. Если вы интерполируете несколько строк, то они последовательно подставляются вместо соответствующих символов `%s`.

Строковое форматирование иногда проще применять, чем конкатенацию строк с помощью оператора +, особенно в случае длинных строк. Кроме того, в отличие от конкатенации можно вставлять в строку, например, целые числа. Введите в интерактивной оболочке следующие инструкции.

```
>>> '%s had %s pies.' % ('Alice', 42)
'Alice had 42 pies.'
>>> 'Alice' + ' had ' + 42 + ' pies.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Целое число 42 без проблем вставляется в строку, если использовать интерполяцию, но попытка конкатенировать его приводит к ошибке.

В строке 33 программы *caesarHacker.py* строковое форматирование применяется для создания строки, содержащей значения обеих переменных, `key` и `translated`. Поскольку в переменной `key` хранится целочисленное значение, мы используем строковое форматирование для того, чтобы вставить это значение в строку, передаваемую функции `print()`.

Резюме

Критической уязвимостью шифра Цезаря является то, что количество возможных ключей, которые можно использовать для шифрования, ограничено. Любой компьютер сможет легко выполнить дешифрование с каждым из 66 возможных ключей, и криптоаналитику понадобится всего лишь несколько секунд для того, чтобы выбрать вариант, соответствующий исходному незашифрованному тексту. Чтобы повысить степень секретности сообщения, мы должны использовать шифр, допускающий большее количество возможных ключей шифрования. Такую секретность может обеспечить перестановочный шифр, который обсуждается в главе 7.

Контрольный вопрос

Ответы на контрольные вопросы приведены в приложении Б.

1. Взломайте следующий зашифрованный текст, дешифруя по одной строке за раз, поскольку строки зашифрованы с использованием разных ключей. Не забудьте о необходимости экранирования кавычек любого вида.

```
qeFIP?eGSeECNNS,
5coOMXXcoPSZIWoQI,
avnl1olyD4l'ylDohww6DhzDjhuDil,

z.GM?.cEQc. 70c.7KcKMKHA9AGFK,
?MFYp2pPJJUpZSIJWpRdpMFY,
ZqH8sl5HtqHTH4s3lyvH5zH5spH4t pHzqHlH3l5K

Zfbi,!tif!xpvme!qspcbcmz!fbu!nfA
```

7

ШИФРОВАНИЕ С ПОМОЩЬЮ ПЕРЕСТАНОВОЧНОГО ШИФРА

"Утверждение о том, что вас не должны волновать проблемы конфиденциальности личной информации, поскольку вам нечего скрывать, равносильно утверждению о том, что вас не должны волновать проблемы свободы слова, поскольку вам нечего сказать".

Эдвард Сноуден, 2015 г.

Шифр Цезаря не является стойким; компьютеру ничего не стоит перебрать методом грубой силы все 66 возможных ключей. С другой стороны, перестановочный шифр труднее поддается взлому методом грубой силы, поскольку количество возможных ключей зависит от длины сообщения. Существует множество различных типов перестановочных шифров, включая *шифр изгороди*, *шифр маршрутной перестановки*, *перестановочный шифр Мышковского* и *прерывистый перестановочный шифр*. В этой главе обсуждается простой перестановочный шифр — *шифр вертикальной перестановки*.

В этой главе...

- Создание функций с помощью инструкции `def`
- Аргументы и параметры
- Область видимости переменных
- Функция `main()`
- Списки
- Сходство списков и строк
- Вложенные списки
- Составные операторы присваивания (`+=`, `-=`, `*=`, `/=`)
- Строковый метод `join()`
- Возвращаемые значения и инструкция `return`
- Переменная `__name__`

Как работает перестановочный шифр

Если в подстановочном шифре одни символы заменяются другими, то в перестановочном шифре символы остаются теми же, но переставляются так, чтобы сообщение было невозможно прочитать. Поскольку каждый ключ создает свой способ упорядочения, или перестановки, символов, криптоаналитик не знает, как нужно переупорядочить шифротекст, чтобы восстановить исходное сообщение.

Чтобы зашифровать сообщение с помощью перестановочного шифра, выполните следующие действия.

1. Определите количество символов в сообщении и длину ключа.
2. Нарисуйте строку ячеек, количество которых равно длине ключа (например, 8 ячеек для ключа 8).
3. Начните заполнять ячейки слева направо, вводя в каждую ячейку по одному символу.
4. Если ячейки исчерпаны, но символы все еще остаются, нарисуйте дополнительную строку ячеек.
5. Введя последний символ, заштрихуйте неиспользованные ячейки в последней строке.
6. Начав с левой верхней ячейки и продвигаясь вниз по каждому столбцу, выпишите символы. Достигнув конца столбца, переходите к следующему столбцу справа. Заштрихованные ячейки пропускайте. Полученный текст и будет шифротекстом.

Чтобы увидеть, как это работает на практике, мы зашифруем сообщение вручную, а затем напишем программу на Python.

Шифрование сообщения вручную

Прежде чем приступить к написанию кода, зашифруем сообщение "Common sense is not so common.", используя лишь карандаш и бумагу. Длина этого сообщения, включая пробелы и знаки пунктуации, составляет 30 символов. В качестве ключа для этого примера мы используем число 8. Возможные значения ключей лежат в диапазоне от 2 до половины длины сообщения, т.е. 15. Чем больше длина сообщения, тем больше возможных значений ключей. В случае шифрования целой книги с использованием вертикального перестановочного шифра количество возможных ключей исчислялось бы тысячами.

Первое, что необходимо сделать, — это нарисовать восемь (в соответствии со значением ключа) ячеек, расположенных в ряд (рис. 7.1).

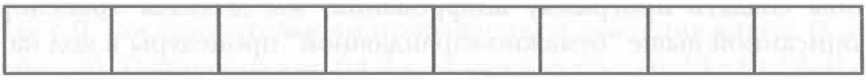


Рис. 7.1. *Количество ячеек в первой строке определяется значением ключа*

На втором шаге записываем шифруемое сообщение в ячейки, помещая в каждую ячейку по одному символу (рис. 7.2). Не забывайте о том, что пробелы тоже считаются символами (в данном случае они представлены символом ■).

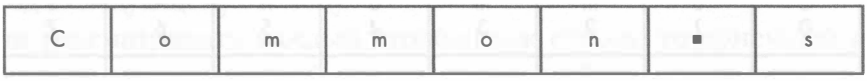


Рис. 7.2. *Заполните каждую ячейку одним символом, включая пробелы*

У вас всего восемь ячеек, а сообщение содержит 30 символов. Исчерпав ячейки, нарисуйте еще один ряд ячеек под предыдущим. Продолжайте этот процесс до тех пор, пока не запишете все сообщение (рис. 7.3).

Заштрихуйте две ячейки в последнем ряду как напоминание о том, что их следует игнорировать. Шифротекст состоит из букв, которые читаются, начиная с левой верхней ячейки, вниз по столбцу. Как показано на рисунке, первый столбец содержит буквы *C*, *e*, *n* и *o*. Достигнув последнего ряда в данном столбце, перейдите к верхнему ряду следующего столбца справа. Следующими символами являются *o*, *n*, *o*, *m*. Игнорируйте заштрихованные ячейки.

1-й	2-й	3-й	4-й	5-й	6-й	7-й	8-й
C	o	m	m	o	n	■	s
e	n	s	e	■	i	s	■
n	o	t	■	s	o	■	c
o	m	m	o	n	.		

Рис. 7.3. *Добавляйте ряды ячеек до тех пор, пока не запишете все сообщение*

Мы получаем шифротекст в виде фразы "Cenoonommstmme oo snnio. s s c", которая достаточно запутана для того, чтобы кому-то, глядя на нее, удалось восстановить исходное сообщение.

Создание программы шифрования

Чтобы создать программу шифрования, мы должны транслировать шаги описанной выше "бумажно-карандашной" процедуры в код на языке Python. Вернемся к рассмотрению того, как следует зашифровать строку `'Common sense is not so common.'`, используя ключ 8. Для Python позиция символа в строке определяется его числовым индексом, поэтому добавим индексы каждой буквы сообщения в соответствующие ячейки (рис. 7.4). (Не забывайте о том, что отсчет индексов начинается с 0, а не с 1.)

1-й	2-й	3-й	4-й	5-й	6-й	7-й	8-й
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		

Рис. 7.4. *Добавьте в каждую ячейку числовые индексы, начиная с 0*

Мы видим, что в первом столбце содержатся символы с индексами 0, 8, 16 и 24 (`'C'`, `'e'`, `'n'` и `'o'`). Следующий столбец содержит символы с индексами 1, 9, 17 и 25 (`'o'`, `'n'`, `'o'` и `'m'`). Обратите внимание на возникающую закономерность: n-й столбец содержит все символы строки, имеющие индексы $0 + (n - 1)$, $8 + (n - 1)$, $16 + (n - 1)$ и $24 + (n - 1)$ (рис. 7.5).

1-й	2-й	3-й	4-й	5-й	6-й	7-й	8-й
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		

Рис. 7.5. *Индексы в каждой ячейке ведут себя предсказуемым образом*

Исключение составляют 7-й и 8-й столбцы последнего ряда, поскольку индексы 24 + (7 – 1) и 24 + (8 – 1) превысили бы значение 29, являющееся максимальным индексом для данной строки. В этих столбцах мы ограничиваемся прибавлением к *n* лишь значений 0, 8 и 16 (опуская значение 24).

Что особенного связано с числами 0, 8, 16 и 24? Мы получаем эти числа, начиная с 0, последовательным прибавлением значения ключа (в данном примере — 8). Таким образом, 0 + 8 дает нам 8, 8 + 8 = 16, 16 + 8 = 24. В результате сложения 24 + 8 мы получили бы 32, но поскольку значение 32 больше, чем длина сообщения, мы не прибавляем 24.

Строка в *n*-м столбце начинается с индекса (*n* – 1), а значение каждого последующего индекса получается путем прибавления 8 (ключа) к предыдущему значению. Этот процесс продолжается до тех пор, пока значение индекса остается меньшим 30 (длины сообщения), после чего выполняется переход к следующему столбцу.

Если рассматривать каждый столбец как строку, то конечный результат будет представлять собой список, состоящий из восьми строк: `'Ceno'`, `'onom'`, `'mstm'`, `'me o'`, `'o sn'`, `'nio.'`, `' s '`, `'s c'`. Если конкатенировать эти строки в указанном порядке, то в результате получаем следующий шифротекст: `'Cenoonommstmme oo snnio. s s c'`.

Исходный код программы `Transposition Encrypt`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *transpositionEncrypt.py*. Не забудьте поместить модуль *pyperclip.py* в тот же каталог, в котором сохранили файл. Затем выполните программу, нажав клавишу <F5>.

transpositionEncrypt.py

```
 1. # Программа шифрования на основе перестановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. def main():
 7.     myMessage = 'Common sense is not so common.'
 8.     myKey = 8
 9.
10.     ciphertext = encryptMessage(myKey, myMessage)
11.
12.     # Отобразить зашифрованную строку, хранящуюся в переменной
13.     # ciphertext, вставив после нее символ '|' на случай, если
14.     # в конце зашифрованного сообщения имеются пробелы
15.     print(ciphertext + '|')
16.
17.     # Скопировать зашифрованную строку в буфер обмена
18.     pyperclip.copy(ciphertext)
19.
20.
21. def encryptMessage(key, message):
22.     # Каждая строка в списке ciphertext представляет столбец таблицы
23.     ciphertext = [''] * key
24.
25.     # Цикл по всем столбцам в списке ciphertext
26.     for column in range(key):
27.         currentIndex = column
28.
29.         # Цикл, пока значение currentIndex не превысит длину сообщения
30.         while currentIndex < len(message):
31.             # Поместить в конец текущего столбца в списке ciphertext
32.             # символ сообщения с индексом currentIndex
33.             ciphertext[column] += message[currentIndex]
34.
35.             # Увеличить значение currentIndex
36.             currentIndex += key
37.
38.     # Возврат списка ciphertext в виде единой строки
39.     return ''.join(ciphertext)
40.
41.
42. # Если файл transpositionEncrypt.py выполняется как программа
43. # (а не импортируется как модуль), вызвать функцию main()
44. if __name__ == '__main__':
45.     main()
```

Пример выполнения программы `Transposition Encrypt`

Выполнив программу *transpositionEncrypt.py*, вы должны получить следующий результат:

```
Cenoonommstmme oo snnio. s s c|
```

Символ вертикальной черты (`|`) обозначает конец шифротекста в том случае, если строка завершается пробелами. Сам шифротекст (без вертикальной черты в конце) также копируется в буфер обмена, чтобы его, например, можно было переслать по электронной почте. Если вы хотите зашифровать другое сообщение или использовать другой ключ, измените значения, присваиваемые переменным `myMessage` и `myKey` в строках 7 и 8. После этого вновь запустите программу.

Создание собственных функций с помощью инструкции `def`

После импорта модуля `pyperclip` мы используем инструкцию `def` для создания пользовательской функции `main()` в строке 6.

```
1. # Программа шифрования на основе перестановочного шифра
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
5.
6. def main():
7.     myMessage = 'Common sense is not so common.'
8.     myKey = 8
```

Инструкция `def` означает, что вы создаете, или *определяете*, новую функцию, которая будет вызываться далее в программе. Блок кода, следующий за инструкцией `def`, — это и есть тот код, который будет выполняться при вызове функции. Когда вы вызываете функцию, выполнение передается блоку, следующему за инструкцией `def`.

Как вы узнали в главе 3, у некоторых функций есть *аргументы* — значения, которые передаются в функцию. Например, в функцию `print()` может передаваться строковое значение в качестве аргумента, указываемого в круглых скобках. Определяя функцию с аргументами, вы помещаете одно или несколько имен переменных в круглые скобки в инструкции `def`. Эти переменные называются *параметрами*. В нашем случае функция `main()` не

имеет параметров, следовательно, она не получает никаких аргументов при вызове. Если попытаться вызвать функцию, указав аргументы в количестве, большем или меньшем, чем количество параметров, заданное в определении функции, то Python выдаст сообщение об ошибке.

Определение функции с параметрами

Давайте создадим функцию с одним параметром и вызовем ее. Откройте в редакторе файлов новое окно и введите следующий код.

helloFunction.py

```
❶ def hello(name):
❷     print('Hello, ' + name)
❸ print('Start.')
❹ hello('Alice')
❺ print('Call the function again:')
❻ hello('Bob')
❼ print('Done.')
```

Сохраните эту программу в файле *helloFunction.py* и запустите ее, нажав клавишу <F5>. Результат должен выглядеть так, как показано ниже.

```
Start.
Hello, Alice
Call the function again:
Hello, Bob
Done.
```

Когда вы запускаете программу *helloFunction.py*, она начинает выполняться с первой строки. Но инструкция `def` (❶) всего лишь определяет функцию `hello()` с одним параметром, каковым является переменная `name`. Программа не выполняет блок (❷), следующий за инструкцией `def`, — это происходит только при вызове функции. Сначала выполняется инструкция `print('Start.')` (❸), и именно поэтому первой строкой, которую выведет программа, будет строка `'Start.'`.

За инструкцией `print('Start.')` следует первый вызов функции `hello()`. Выполнение передается первой строке в блоке (❷) функции `hello()`. Строка `'Alice'` передается в качестве аргумента и присваивается параметру `name`. Этот вызов функции выводит на экран строку `'Hello, Alice'`.

Когда программа достигает конца блока инструкции `def`, выполнение возвращается в строку (❹), содержащую исходный вызов функции, и передается следующей инструкции (❺), выводящей текст `'Call the function again:'`.

Далее следует второй вызов функции `hello()` (❻). Программа вновь возвращается к определению функции `hello()` (❷) и еще раз выполняет ее код, отображая на экране текст `'Hello, Bob'`. После выхода из функции программа переходит к следующей строке, содержащей инструкцию `print('Done.')` (❼). Это последняя строка, на которой работа программы завершается.

Изменение параметров сказывается лишь внутри функции

Введите в интерактивной оболочке приведенный ниже код. Здесь определяется, а затем вызывается функция `func()`. Обратите внимание на то, что для завершения блока инструкции `def` в интерактивной оболочке следует ввести пустую строку после `param = 42`.

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

>>> spam = 'Hello'
>>> func(spam)
>>> print(spam)
Hello
```

Функция `func()` получает параметр `param` и устанавливает его значение равным `42`. Код вне этой функции создает переменную `spam` и присваивает ей строковое значение, после чего вызывает функцию с аргументом `spam` и выводит значение аргумента.

Функция `print()` в последней строке выведет строку `'Hello'`, а не число `42`. Когда функция `func()` вызывается с переменной `spam` в качестве аргумента, хранящееся в этой переменной значение копируется и присваивается переменной `param`. Любые изменения переменной `param` в теле функции не приведут к изменению значения переменной `spam`. (Исключением из этого правила являются случаи, когда в качестве аргумента функции передается список или словарь, о чем будет рассказано в разделе "Списковые переменные используют ссылки" главы 9.)

Каждый раз, когда вызывается функция, создается *локальная область видимости*. Переменные и параметры, созданные во время вызова функции, существуют только в этой локальной области видимости и называются *локальными*. Область видимости можно представить в виде контейнера, внутри которого существуют переменные. Когда происходит возврат из функции, локальная область видимости уничтожается, и информация о локальных переменных, которые хранились в ней, теряется.

Переменные, созданные вне любой функции, существуют в *глобальной области видимости* и называются *глобальными*. Когда происходит выход из программы, глобальная область видимости прекращает существование, и информация обо всех переменных программы теряется. (Все переменные в программах обратного шифрования и шифрования на основе шифра Цезаря, рассмотренных в главах 4 и 5, были глобальными.)

Переменная должна быть либо локальной, либо глобальной и не может быть одновременно локальной и глобальной. Две различные переменные могут иметь одинаковые имена при условии, что они принадлежат к разным областям видимости. Они по-прежнему считаются двумя разными переменными аналогично тому, как Мэйн-стрит в Сан-Франциско и Мэйн-стрит в Бирмингеме — совершенно разные улицы.

Важно понимать, что значение аргумента, передаваемое при вызове функции, копируется в параметр. Поэтому, даже если параметр изменяется, значение переменной, послужившей аргументом, остается неизменным.

Определение функции `main()`

В строках 6–8 программы *transpositionEncrypt.py* содержится определение функции `main()`, которая, будучи вызванной, устанавливает значения переменных `myMessage` и `myKey`.

```
6. def main():
7.     myMessage = 'Common sense is not so common.'
8.     myKey = 8
```

Остальные программы в книге также содержат функцию `main()`. Объяснение причин того, почему мы используем функцию `main()`, будет дано в конце главы, а пока что вам достаточно знать, что функция `main()` всегда вызывается сразу же после запуска любой из программ, описанных в книге.

Строки 7 и 8 — первые две строки блока кода, определяющего функцию `main()`. В этих строках в переменных `myMessage` и `myKey` сохраняется простое текстовое сообщение, которое необходимо зашифровать, и ключ шифрования. Строка 9 — пустая, но все еще является частью блока и отделяет строки 7 и 8 от строки 10 для улучшения читаемости кода. В строке 10 переменной `ciphertext` присваивается зашифрованное сообщение путем вызова функции с двумя аргументами:

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

Код, выполняющий фактическое шифрование, содержится в функции `encryptMessage()`, определяемой далее, в строке 21. Эта функция имеет два аргумента: целочисленное значение ключа и строка сообщения, подлежащего шифрованию. В данном случае мы передаем переменные `myMessage` и `myKey`, определенные перед этим в строках 7 и 8. Когда функция вызывается с несколькими аргументами, они разделяются запятой.

Функция `encryptMessage()` возвращает строку зашифрованного текста.

Шифротекст сообщения выводится на экран в строке 15 и копируется в буфер обмена в строке 18.

```
12.     # Отобразить зашифрованную строку, хранящуюся в переменной
13.     # ciphertext, вставив после нее символ '|' на случай, если
14.     # в конце зашифрованного сообщения имеются пробелы
15.     print(ciphertext + '|')
16.
17.     # Скопировать зашифрованную строку в буфер обмена
18.     pyperclip.copy(ciphertext)
```

После сообщения программа выводит символ `'|'`, чтобы пользователь мог видеть любые возможные пробелы в конце шифротекста.

Строка 18 — последняя в функции `main()`. Как только она завершается, выполнение передается строке, которая следует за строкой вызова функции `main()`.

Передача ключа и сообщения в качестве аргументов

Переменные `key` и `message`, указанные в скобках в строке 21, — это параметры функции.

```
21. def encryptMessage(key, message):
```

Когда функция `encryptMessage()` вызывается в строке 10, ей передаются два аргумента (значения, сохраненные в переменных `myKey` и `myMessage`). Эти значения присваиваются параметрам `key` и `message` при передаче выполнения в функцию.

Возможно, вас удивляет, почему мы ввели параметры `key` и `message`, если у нас уже имеются переменные `myKey` и `myMessage`, определенные в функции `main()`. Это требуется по той причине, что переменные `myKey` и `myMessage` принадлежат к локальной области видимости функции `main()` и не могут использоваться вне ее.

Списковый тип данных

В строке 23 программы *transpositionEncrypt.py* появляется новый тип данных: *список*.

```
22.     # Каждая строка в списке ciphertext представляет столбец таблицы
23.     ciphertext = [''] * key
```

Прежде чем двигаться дальше, необходимо узнать о том, как работают списки и что можно делать с их помощью. Список может содержать различные значения. Аналогично тому, как строки начинаются и заканчиваются кавычками, список начинается с открывающей квадратной скобки ([) и заканчивается парной скобкой (]). Значения, сохраненные в списке, указываются между этими скобками. Если список содержит несколько значений, то они разделяются запятыми.

Чтобы увидеть список в действии, введите в интерактивной оболочке следующий код.

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

Переменная `animals` хранит значение в виде списка, содержащего четыре строки. Индивидуальные значения, находящиеся в списке, называют *элементами* списка. Списки идеально подходят для тех случаев, когда в одной переменной необходимо сохранить несколько значений.

Многие операции, поддерживаемые для строк, допустимы и в случае списков. Например, индексация и взятие срезов в списках реализованы точно так же, как и в отношении строк. Разница лишь в том, что, например, вместо обращения к отдельным символам, как в случае строк, индексация списков позволяет обращаться к их отдельным элементам. Введите в интерактивной оболочке следующие команды.

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

Учитывайте, что первый индекс равен 0, а не 1 (❶). Аналогично тому, как применение среза к строке позволяет получить новую строку, являющуюся частью исходной, применение среза в отношении списка позволяет создать на его основе список меньшего размера. И помните о том, что если указан второй индекс, то срез доходит до элемента с этим индексом, но не включает его (❷).

Цикл `for` может проходить по элементам списка точно так же, как по символам строки. Значением, сохраняемым в переменной цикла `for`, является одиночный элемент из списка. Введите в интерактивной оболочке следующий код.

```
>>> for spam in ['aardvark', 'anteater', 'albert']:
...   print('For dinner we are cooking ' + spam)
...
For dinner we are cooking aardvark
For dinner we are cooking anteater
For dinner we are cooking albert
```

На каждой итерации цикла переменной `spam` присваивается новое значение из списка, начиная с элемента с индексом 0 и до конца списка.

Изменение элементов списка

Элементы списка можно изменять с помощью обычной инструкции присваивания, указывая нужный индекс. Введите в интерактивной оболочке следующие команды.

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

Чтобы изменить третий элемент списка `animals`, мы используем индексацию в виде выражения `animals[2]`, а затем с помощью оператора присваивания меняем значение с `'albert'` на `9999` (❶). Если затем проверить содержимое списка, то значения `'albert'` в нем уже не будет (❷).

Замена символов в строках

Несмотря на возможность переназначения элементов списка, не получится сделать то же самое в строке. Введите в интерактивной оболочке следующий код:

```
>>> 'Hello world!'[6] = 'X'
```

На экране появится следующее сообщение об ошибке.

```
Traceback (most recent call last):
  File <pyshell#0>, line 1, in <module>
    'Hello world!'[6] = 'X'
TypeError: 'str' object does not support item assignment
```

Причиной появления этой ошибки является то, что Python не позволяет использовать операторы присваивания в отношении индексированных строковых значений. Чтобы изменить символ в строке, необходимо создать новую строку с помощью срезов. Введите в интерактивной оболочке следующие команды.

```
>>> spam = 'Hello world!'
>>> spam = spam[:6] + 'X' + spam[7:]
>>> spam
'Hello Xorld!'
```

Сначала необходимо создать срез от начала строки и до символа, подлежащего замене, но не включая его. После этого можете конкатенировать полученный результат со строкой в виде нового символа и срезом, начинающимся со следующего символа и охватывающим весь остаток исходной строки. Тем самым вы получите исходную строку, в которой изменен всего лишь один символ.

Вложенные списки

Списки могут содержать другие списки. Введите в интерактивной оболочке следующие команды.

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

```
1
>>> spam[1][1]
2
```

Значением выражения `spam[0]` является список `['dog', 'cat']`, обладающий собственными индексами. Использование двойных индексов в квадратных скобках в выражении `spam[0][0]` указывает на то, что мы извлекаем первый элемент из первого списка: `spam[0]` дает нам список `['dog', 'cat']`, а `['dog', 'cat'][0]` — строку `'dog'`.

Применение функции `len()` и оператора `in` к спискам

Ранее мы применяли функцию `len()` для определения количества символов в строке (т.е. длины строки). Функция `len()` работает и в отношении списков, возвращая целое число, соответствующее количеству элементов списка.

Введите в интерактивной оболочке следующий код.

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

Мы также применяли операторы `in` и `not in` для проверки вхождения одной строки в другую. Оператор `in` также позволяет проверить, встречается ли в списке заданное значение, тогда как оператор `not in`, наоборот, позволяет удостовериться в том, что заданного значения нет в списке. Введите в интерактивной оболочке следующие команды.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'anteater' in animals
True
>>> 'anteater' not in animals
False
❶ >>> 'anteat' in animals
False
❷ >>> 'anteat' in animals[1]
True
>>> 'delicious spam' in animals
False
```

Почему выражение ❶ возвращает `False`, а выражение ❷ — `True`? Вспомните, что `animals` — это список, тогда как вычисление выражения `animals[1]` дает строку `'anteater'`. Результатом выражения ❶ является значение `False`, поскольку строка `'anteat'` не входит в список `animals`.

В то же время результат выражения ❷ равен `True`, поскольку `animals[1]` — это строка `'anteater'`, в которой встречается строка `'anteat'`.

Аналогично тому, как пустая пара кавычек представляет пустую строку, пустая пара квадратных скобок представляет пустой список. Введите в интерактивной оболочке следующий код.

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

Список `animals` — пустой, поэтому его длина равна нулю.

Конкатенация и репликация списков с помощью операторов + и *

Операторы + и * позволяют конкатенировать и реплицировать не только строки, но и списки. Введите в интерактивной оболочке следующие команды.

```
>>> ['hello'] + ['world']
['hello', 'world']
>>> ['hello'] * 5
['hello', 'hello', 'hello', 'hello', 'hello']
```

На этом мы закончим рассмотрение примеров сходства строк и списков. Просто помните, что большинство операций над строками можно также выполнять и в отношении списков.

Алгоритм шифрования с помощью перестановочного шифра

В нашем алгоритме шифрования мы используем списки для создания шифротекста. Вернемся к коду программы *transpositionEncrypt.py*. Как мы уже видели, в строке 23 переменная `ciphertext` представляет собой список пустых строк.

```
22.     # Каждая строка в списке ciphertext представляет столбец таблицы
23.     ciphertext = [''] * key
```

Каждый элемент списка `ciphertext` представляет столбец таблицы перестановочного шифра. Поскольку количество столбцов определяется значением ключа, можно реплицировать список с одним пустым строковым значением, умножив его на ключ. Именно таким способом в строке 23 формируется список, содержащий нужное количество пустых строк. Эле-

ментам списка будут присваиваться все символы, которые попадают в один столбец таблицы. В результате мы получим список строк, представляющих каждый столбец. Поскольку индексы списка отсчитываются от нуля, индексация столбцов также должна начинаться с нуля. Таким образом, элемент `ciphertext[0]` представляет крайний слева столбец, `ciphertext[1]` — столбец справа от него и т.д.

Чтобы увидеть, как это работает, вернемся к рассмотренной ранее сетке для примера "Common sense is not so common.", в которой добавленные сверху номера столбцов соответствуют индексам списка (рис. 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	.		

Рис. 7.6. *Пример сетки сообщения с индексами списка для каждого столбца*

Если мы вручную присвоим строковые значения переменной `ciphertext` для этой сетки, то она примет следующий вид.

```
>>> ciphertext = ['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.',
                  ' s ', 's c']
>>> ciphertext[0]
'Ceno'
```

Следующим шагом является добавление текста в каждый элемент списка `ciphertext`, как мы только что сделали вручную, но на этот раз программным способом.

```
25.     # Цикл по всем столбцам в списке ciphertext
26.     for column in range(key):
27.         currentIndex = column
```

Цикл `for` в строке 26 проходит по всем столбцам, а переменная `column` содержит целочисленное значение, которое будет использоваться для индексации списка `ciphertext`. На первой итерации цикла переменная `column` содержит `0`, на второй итерации — `1`, затем — `2` и т.д. Каждый раз мы получаем индекс строки, к которой впоследствии хотим обратиться, используя выражение `ciphertext[column]`.

Таким образом, в переменной currentIndex хранится индекс строки сообщения, которая требуется программе на текущей итерации цикла for. На каждой итерации этого цикла в строке 27 переменной currentIndex присваивается значение, соответствующее номеру столбца. Далее мы создадим шифротекст, конкатенируя перемешанные части сообщения по одному символу за раз.

Составные операторы присваивания

До сих пор мы выполняли конкатенацию и сложение, используя оператор +, который прибавлял к переменной новое значение. Но зачастую нам нужно, чтобы новое значение переменной базировалось на ее текущем значении, поэтому переменная становилась частью выражения, которое вычисляется и присваивается ей самой, как в приведенном ниже примере.

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

Но есть и другие способы манипулирования значениями переменных на основе их текущих значений. Например, это можно делать с помощью *составных операторов присваивания*. Инструкция spam += 2, в которой используется составной оператор присваивания +=, равносильна инструкции spam = spam + 2. Это всего лишь более короткий способ записи данной операции. Оператор += реализует сложение целых чисел, а также конкатенацию строк и списков. В табл. 7.1 приведены примеры составных операторов присваивания и эквивалентные им простые операции.

Таблица 7.1. Составные операторы присваивания

Составной оператор присваивания	Эквивалентная операция
spam += 42	spam = spam + 42
spam -= 42	spam = spam - 42
spam *= 42	spam = spam * 42
spam /= 42	spam = spam / 42

Мы будем использовать составные операторы присваивания для конкатенации символов в шифротекст.

Перемещение текущего индекса по строке сообщения

Переменная currentIndex хранит индекс следующего символа строки сообщения, который будет конкатенироваться с элементом списка ciphertext. На каждой итерации цикла while (строка 30) к переменной currentIndex прибавляется переменная key, позволяющая выбрать другой символ сообщения, а на каждой итерации цикла for (строка 26) для переменной currentIndex устанавливается новое значение переменной column.

Чтобы зашифровать строку, хранящуюся в переменной message, мы должны взять первый символ сообщения, т.е. 'C', и поместить его в первый элемент списка ciphertext. После этого следует пропустить восемь символов сообщения (поскольку значение переменной key равно 8) и конкатенировать соответствующий символ, т.е. 'e', с первым элементом шифротекста. Мы должны продолжать пропускать символы в соответствии со значением ключа и конкатенировать соответствующие символы до тех пор, пока не достигнем конца сообщения. В результате будет создана строка 'Ceno', являющаяся первым столбцом шифротекста. Затем мы должны повторить эти действия, начиная со второго символа сообщения, для создания второго столбца.

В цикле for, который начинается в строке 26, содержится цикл while, начинающийся в строке 30. В этом цикле мы находим подходящий символ сообщения и конкатенируем его для создания текущего столбца. Цикл выполняется до тех пор, пока значение переменной currentIndex остается меньше длины сообщения.

```
29.         # Цикл, пока значение currentIndex не превысит длину сообщения
30.         while currentIndex < len(message):
31.             # Поместить в конец текущего столбца в списке ciphertext
32.             # символ сообщения с индексом currentIndex
33.             ciphertext[column] += message[currentIndex]
34.
35.             # Увеличить значение currentIndex
36.             currentIndex += key
```

Для каждого столбца цикл while проходит по символам исходного сообщения, выбирая их с интервалом key путем добавления ключа к значению currentIndex. На первой итерации цикла for (в строке 27) значение переменной currentIndex устанавливается равным значению переменной column, которая первоначально равна нулю.

Как показано на рис. 7.7, значением выражения `message[currentIndex]` на первой итерации является первый символ сообщения. В строке 33 содержащийся в позиции `message[currentIndex]` символ конкатенируется со строкой `ciphertext[column]`, давая начало первому столбцу. В конце каждой итерации цикла `while` (строка 36) к значению переменной `currentIndex` прибавляется значение переменной `key` (т.е. 8). Таким образом, на первой итерации в столбец добавляется символ `message[0]`, на второй — `message[8]`, на третьей `message[16]` и на четвертой — `message[24]`.

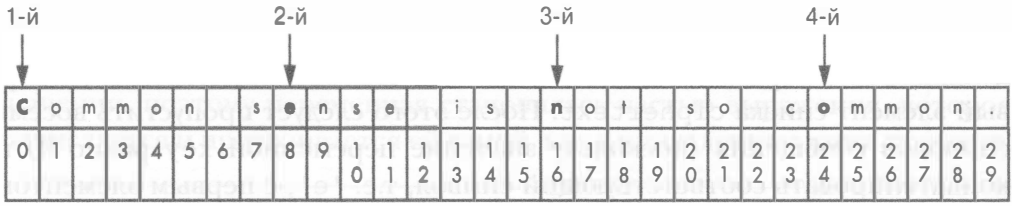


Рис. 7.7. *Стрелками указаны символы сообщения, извлекаемые с помощью выражения `message[currentIndex]` на первой итерации цикла `for`, когда значение переменной `column` установлено равным `0`*

Пока значение `currentIndex` меньше длины строки `message`, мы продолжаем конкатенацию символов `message[currentIndex]` до конца элемента с индексом `column` в списке `ciphertext`. Как только значение `currentIndex` превысит длину сообщения, цикл `while` завершится и продолжится цикл `for`. Поскольку в блоке цикла `for` отсутствует какой-либо другой код после цикла `while`, начинается очередная итерация цикла `for`, на которой значение переменной `column` становится равным `1`. Это же значение записывается в переменную `currentIndex`.

Теперь, когда на каждой итерации цикла `while` значение переменной `currentIndex` будет увеличиваться на `8` в строке 36, индексы будут последовательно принимать значения `1`, `9`, `17` и `25` (рис. 7.8).

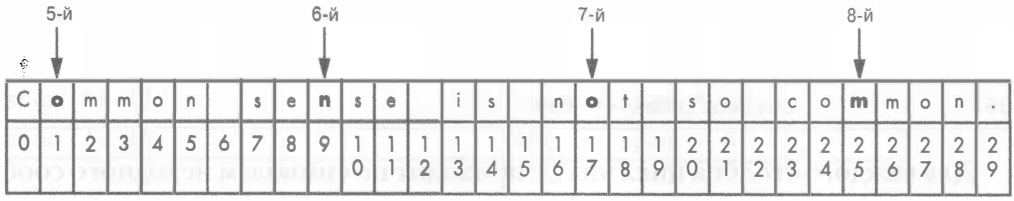


Рис. 7.8. *Стрелками указаны символы сообщения, извлекаемые с помощью выражения `message[currentIndex]` на второй итерации цикла `for`, когда значение переменной `column` установлено равным `1`*

После конкатенации символов `message[1]`, `message[9]`, `message[17]` и `message[25]` в элементе `ciphertext[1]` они образуют строку `'onom'`. Это второй столбец таблицы.

Когда цикл `for` завершит проход по оставшимся столбцам, список `ciphertext` будет содержать элементы `['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']`. Получив список строк, мы должны объединить их для создания одной строки, содержащей весь шифротекст: `'Cenoonommstmme oo snnio. s s c'`.

Строковый метод `join()`

Метод `join()` используется в строке 39 для конкатенации отдельных строк, содержащихся в списке `ciphertext`, в одну строку. Он вызывается для строкового значения и получает в качестве аргумента список строк. Возвращаемая строка будет содержать все элементы списка, разделенные строкой, для которой был вызван метод `join()`. (Если предполагается конкатенация строк, то строка-разделитель должна быть пустой.) Введите в интерактивной оболочке следующие команды.

```
>>> eggs = ['dogs', 'cats', 'moose']
❶ >>> ''.join(eggs)
'dogscatsmoose'
❷ >>> ', '.join(eggs)
'dogs, cats, moose'
❸ >>> 'XYZ'.join(eggs)
'dogsXYZcatsXYZmoose'
```

Когда метод `join()` вызывается для пустой строки (❶), вы получаете простую конкатенацию строк списка `eggs` без каких-либо других строк между ними. Но иногда требуется разделять элементы списка для наглядности, что мы и сделали (❷), вызвав метод `join()` для строки `', '`. Благодаря этому между элементами списка вставляется строка `', '`. Между элементами списка можно вставить любую строку (❸).

Возвращаемые значения и инструкция `return`

Результатом вызова функции (или метода) всегда является значение, называемое *возвращаемым*. Когда вы создаете собственную функцию с помощью инструкции `def`, инструкция `return` сообщает Python, что собой представляет возвращаемое значение функции.

```
38.     # Возврат списка ciphertext в виде единой строки
39.     return ''.join(ciphertext)
```

В строке 39 метод `join()` вызывается для пустой строки и получает в качестве аргумента переменную `ciphertext`, в результате чего строки списка `ciphertext` объединяются в одну строку.

Пример инструкции `return`

Инструкция `return` записывается в виде ключевого слова `return`, за которым следует возвращаемое значение. Вместо фиксированного значения можно указать выражение, как это сделано в строке 39. В таком случае возвращаемым значением является результат вычисления данного выражения. Откройте новое окно в редакторе файлов, введите приведенный ниже текст программы, сохраните его в файле *addNumbers.py* и запустите программу, нажав клавишу <F5>.

addNumbers.py

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

Выполнив программу *addNumbers.py*, вы должны получить следующий результат:

```
42
```

Это объясняется тем, что результатом вызова функции `addNumbers(2, 40)` в строке 4 является число 42. Инструкция `return` в строке 2 функции `addNumbers()` вычисляет выражение `a + b`, а затем возвращает полученный результат.

Возврат зашифрованного шифротекста

В программе *transpositionEncrypt.py* инструкция `return` в функции `encryptMessage()` возвращает строковое значение, которое создается путем объединения всех строк, являющихся элементами списка `ciphertext`. Если в переменной `ciphertext` содержится список `['Ceno', 'onom', 'mstm', 'me o', 'o sn','nio.', ' s ', 's c']`, то метод `join()` сформирует строку `'Cenoonommstmme oo snnio. ss c'`. Именно эта строка, являющаяся результатом работы программы шифрования, возвращается функцией `encryptMessage()`.

Функции обладают тем замечательным преимуществом, что программисту достаточно знать лишь, для чего именно предназначена функция, и вовсе не обязательно знать, как работает ее код. Программист может быть уверен в том, что если он вызовет функцию `encryptMessage()` и передаст ей целое число и строку в качестве значений параметров `key` и `message`, то она вычислит зашифрованную строку. Он не обязан знать что-либо о том, как именно достигается результат, точно так же, как вы, не имея ни малейшего понятия о том, как работает код функции `print()`, знаете, что если передать ей строку, то она выведет ее на экран.

Переменная `__name__`

Программу шифрования на основе перестановочного шифра можно превратить в модуль с помощью простого трюка, включающего использование функции `main()` и переменной `__name__`.

Когда вы запускаете программу на языке Python, переменной `__name__` (обратите внимание на префикс и суффикс в виде двойного подчеркивания) автоматически, еще до того как выполнится первая строка программы, присваивается строковое значение `'__main__'` (и вновь двойное подчеркивание до и после `main`).

В самом конце файла (и, что более важно, после всех инструкций `def`) можно проверить, содержит ли переменная `__name__` строку `'__main__'`. Если это действительно так, то вызывается функция `main()`.

Фактически инструкция `if` в строке 44 является одной из первых, выполняемых при запуске программы (сначала выполняются инструкции `import` и `def`).

```
42. # Если файл transpositionEncrypt.py выполняется как программа
43. # (а не импортируется как модуль), вызвать функцию main()
44. if __name__ == '__main__':
45.     main()
```

Подобная организация кода объясняется тем, что Python автоматически устанавливает для переменной `__name__` значение `'__main__'` только в том случае, если файл запускается как основная программа. В случае же импорта файла другой программой на языке Python для переменной `__name__` будет установлено строковое значение `'transpositionEncrypt'`. Аналогично тому как программа импортирует модуль `pyperclip`, чтобы можно было вызывать содержащиеся в нем функции, другим программам может понадобиться импортировать модуль *transpositionEncrypt.py*, чтобы можно было вызвать его функцию `encryptMessage()`, не запуская функцию `main()`.

Когда выполняется инструкция `import`, интерпретатор Python ищет файл модуля, присоединяя расширение *.py* к имени файла (вот почему инструкция `import pyperclip` импортирует файл *pyperclip.py*). Еще до того, как программа начнет выполняться, переменной `__name__` присваивается имя файла без расширения *.py*. Именно так наша программа выясняет, выполняется ли она в качестве основной программы или импортируется в виде модуля другой программой. (Мы будем импортировать файл *transpositionEncrypt.py* в виде модуля в главе 9.)

При импорте модуля `transpositionEncrypt` выполнятся все содержащиеся в нем инструкции `def`, и в результате будет определена функция `encryptMessage()`. А вот функция `main()` не будет вызвана, поэтому код шифрования строки `'Common sense is not so common.'` с ключом 8 не будет выполнен.

Именно по этой причине код, реализующий шифрование строки `myMessage` с помощью ключа `myKey`, помещен в тело функции (которой по общепринятому соглашению обычно присваивается имя `main()`). Он не будет выполнен, если файл *transpositionEncrypt.py* импортируется другими программами, в то же время программы по-прежнему смогут вызывать его функцию `encryptMessage()`. Это обеспечивает повторное использование кода данной функции другими программами.

Примечание

Полезный способ изучить работу программы – выполнить ее пошагово, отслеживая промежуточные результаты. Вы сможете проследить за работой функции, выводящей приветствие, и программы шифрования на основе перестановочного шифра, воспользовавшись трассировщиками по адресу `https://goo.gl/KzXaRr` *(helloFunction.py) и* `https://goo.gl/b5nTeu` *(transpositionEncrypt.py). Трассировщик в наглядной форме представит вам результаты выполнения каждой строки кода.*

Резюме

В этой главе мы изучили новые функции, операторы и типы данных, что позволило нам реализовать более сложные алгоритмы обработки данных. Программа шифрования на основе перестановочного шифра сложнее, чем программа шифрования на основе шифра Цезаря, рассмотренная в главе 6, зато перестановочный шифр намного более стойкий. Главное — помнить, что залог успешного понимания кода — умение пошагово анализировать его, как это делает интерпретатор Python.

Программный код можно группировать в блоки, называемые функциями, которые создаются с помощью инструкции `def`. Функция может иметь параметры, значения которым передаются в виде аргументов вызова. Параметры — это локальные переменные. Переменные, создаваемые вне функций, являются глобальными. Локальные переменные отличаются от глобальных, даже если их имена совпадают. То же самое относится и к локальным переменным разных функций, имеющим одинаковые имена.

Списки могут хранить множество значений, в том числе другие списки. Многие операции, привычные для строк (такие, как индексация, извлечение срезов и использование функции `len()`), также применимы и к спискам. Метод `join()` позволяет объединять элементы списка, содержащего строки, в одну строку.

Составные операторы присваивания обеспечивают более короткий способ записи стандартных операций присваивания.

В главе 8 вы узнаете о том, как дешифровать текст, зашифрованный с помощью перестановочного шифра.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Зашифруйте перестановочным шифром с ключом 9 приведенные ниже сообщения, используя карандаш и бумагу. Количество символов в каждом из них указано для удобства.
   - "Underneath a huge oak tree there was of swine a huge company," (61 символ).
   - "That grunted as they crunched the mast: For that was ripe and fell full fast." (77 символов).
   - "Then they trotted away for the wind grew high: One acorn they left, and no more might you spy." (94 символа).
2. В приведенной ниже программе переменные `spam` являются глобальными или локальными?

```
spam = 42
def foo():
    global spam
    spam = 99
    print(spam)
```

3. Каким будет результат вычисления каждого из приведенных ниже выражений?

```
[0, 1, 2, 3, 4][2]
[[1, 2], [3, 4]][0]
[[1, 2], [3, 4]][0][1]
['hello'][0][1]
[2, 4, 6, 8, 10][1:3]
list('Hello world!')
list(range(10))[2]
```

4. Каким будет результат вычисления каждого из приведенных ниже выражений?

```
len([2, 4])
len([])
len([,', ,', ,'])
[4, 5, 6] + [1, 2, 3]
3 * [1, 2, 3] + [9]
42 in [41, 42, 42, 42]
```

5. Назовите четыре составных оператора присваивания.

8

ДЕШИФРОВАНИЕ ПЕРЕСТАНОВОЧНОГО ШИФРА

"Ослабление шифрования или создание бэкдоров, чтобы хорошие парни могли получать доступ к защищенным устройствам или данным, фактически создает уязвимости, которыми не преминут воспользоваться плохие парни".

Тим Кук, CEO Apple, 2015 г.

В отличие от шифра Цезаря, в случае перестановочного шифра процесс дешифрования принципиально отличается от процесса шифрования. В этой главе мы напишем отдельную программу *transpositionDecrypt.py*, предназначенную для дешифрования текстов, зашифрованных с помощью перестановочного шифра.

В этой главе...

- Дешифрование перестановочного шифра
- Функции `round()`, `math.ceil()` и `math.floor()`
- Булевы операторы `and` и `or`
- Таблицы истинности

Как дешифровать на бумаге текст, зашифрованный с помощью перестановочного шифра

Представьте, что вы отправили шифротекст "Cenoonommstmme oo snnio. s s c" своему другу, которому известен секретный ключ 8. Первое, что он должен сделать для того, чтобы дешифровать шифротекст, — это подсчитать количество ячеек, которые нужно начертить. Для этого необходимо разделить длину зашифрованного сообщения на ключ и округлить результат до ближайшего целого числа, если он не целый. Длина шифротекста (которая совпадает с длиной простого текста) составляет 30 символов, а значение ключа равно 8, так что в результате деления 30 на 8 получаем 3,75.

Округлив 3,75 до 4, ваш друг начертит таблицу с четырьмя столбцами и восемью строками (ключ).

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

Рис. 8.1. *Дешифрование сообщения путем обращения сетки*

Ему также придется рассчитать количество ячеек, которые необходимо заштриховать. Взяв общее количество ячеек (32), он вычтет из него длину шифротекста, равную 30 (32 – 30 = 2), и, исходя из этого, заштрихует две *нижние* ячейки в *крайнем правом* столбце.

После этого он начнет последовательно заполнять ячейки символами шифротекста. Начиная с крайней слева ячейки в верхней строке, он заполнит их в направлении слева направо, как это делали вы, когда шифровали текст. Зашифрованный текст — это строка "Cenoonommstmme oo snnio. s s c", поэтому строка "Ceno" попадет в первую строку, "onom" — во вторую и т.д. По завершении этого процесса ячейки будут выглядеть так, как показано на рис. 8.1 (символ ■ представляет пробел).

Ваш друг, получивший шифротекст, замечает, что чтение текста по столбцам позволяет восстановить исходный текст: "Common sense is not so common.".

Подытожим действия, которые нужно выполнить для расшифровки текста.

1. Рассчитайте необходимое количество столбцов, разделив длину сообщения на значение ключа и округлив результат до целого числа.

2. Начертите ячейки, расположив их в виде таблицы, состоящей из строк и столбцов. Используйте количество столбцов, рассчитанное в п. 1. Количество строк совпадает со значением ключа.
3. Рассчитайте количество ячеек, которые необходимо заштриховать, вычтя длину зашифрованного сообщения из общего количества ячеек (количества строк, умноженного на количество столбцов).
4. Заштрихуйте определенное в п. 3 количество ячеек в нижней части крайнего справа столбца.
5. Заполните символами шифротекста ячейки, начиная с верхней строки слева направо. Заштрихованные ячейки пропустите.
6. Восстановите простой текст, начав читать его сверху вниз с крайнего слева столбца и поступая аналогичным образом с оставшимися столбцами.

Обратите внимание на то, что в случае использования другого ключа у вас получится таблица с неверным количеством строк. Даже если вы корректно выполните оставшиеся этапы процесса дешифрования, результирующий простой текст будет представлять собой бессмысленный набор символов (аналогично тому, как если бы вы использовали не тот ключ в случае шифра Цезаря).

Исходный код программы `Transposition Decrypt`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨New File. Введите в этом окне приведенный ниже код и сохраните его в файле *transpositionDecrypt.py*. Не забудьте поместить модуль *pyperclip.py* в тот же каталог, в котором сохранили файл. Затем выполните программу, нажав клавишу <F5>.

transpositionDecrypt.py

```
 1. # Программа дешифрования перестановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import math, pyperclip
 5.
 6. def main():
 7.     myMessage = 'Cenoonommstmme oo snnio. s s c'
 8.     myKey = 8
 9.
10.     plaintext = decryptMessage(myKey, myMessage)
11.
12.     # Вывести текст с завершающим символом '|' на случай,
13.     # если в конце дешифрованного сообщения имеются пробелы
```

```
14.     print(plaintext + '|')
15.
16.     pyperclip.copy(plaintext)
17.
18.
19. def decryptMessage(key, message):
20.     # Функция расшифровки перестановочного шифра будет имитировать
21.     # столбцы и строки таблицы, в которые вписан простой текст,
22.     # используя список строк. Сначала вычислим несколько значений.
23.
24.     # Количество столбцов в перестановочной таблице
25.     numOfColumns = int(math.ceil(len(message) / float(key)))
26.     # Количество строк в таблице
27.     numOfRows = key
28.     # Количество "заштрихованных" ячеек в последнем столбце
29.     numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
30.
31.     # Каждая строка в списке plaintext представляет столбец
32.     plaintext = [''] * numOfColumns
33.
34.     # Переменные column и row указывают, в какую ячейку должен
35.     # быть помещен символ зашифрованного сообщения
36.     column = 0
37.     row = 0
38.
39.     for symbol in message:
40.         plaintext[column] += symbol
41.         column += 1  # указывает на следующий столбец
42.
43.         # Если больше нет столбцов ИЛИ мы оказались в "заштрихованной"
44.         # ячейке, перейти к первому столбцу следующей строки
45.         if (column == numOfColumns) or (column == numOfColumns - 1
               and row >= numOfRows - numOfShadedBoxes):
46.             column = 0
47.             row += 1
48.
49.     return ''.join(plaintext)
50.
51.
52. # Если файл transpositionDecrypt.py выполняется как программа
53. # (а не импортируется как модуль), вызвать функцию main()
54. if __name__ == '__main__':
55.     main()
```

Пример выполнения программы `Transposition Decrypt`

Выполнив программу расшифровки перестановочного шифра, вы должны получить следующий результат:

```
Common sense is not so common.|
```

Если хотите дешифровать другое сообщение или использовать другой ключ, измените значения, присваиваемые переменным `myMessage` и `myKey` в строках 7 и 8.

Импорт модулей и функция `main()`

Первая часть программы *transpositionDecrypt.py* аналогична первой части программы *transpositionEncrypt.py*.

```
 1. # Программа дешифрования перестановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import math, pyperclip
 5.
 6. def main():
 7.     myMessage = 'Cenoonommstmme oo snnio. s s c'
 8.     myKey = 8
 9.
10.     plaintext = decryptMessage(myKey, myMessage)
11.
12.     # Вывести текст с завершающим символом '|' на случай,
13.     # если в конце дешифрованного сообщения имеются пробелы
14.     print(plaintext + '|')
15.
16.     pyperclip.copy(plaintext)
```

В строке 4 импортируются модули `pyperclip` и `math`. С помощью одной инструкции `import` можно импортировать сразу несколько модулей, разделив их имена запятыми.

В функции `main()`, определение которой начинается в строке 6, создаются переменные `myMessage` и `myKey`, а затем вызывается функция `decryptMessage()`. Возвращаемым значением функции `decryptMessage()` будет дешифрованный простой текст, восстановленный из шифротекста с помощью ключа. Этот текст сохраняется в переменной `plaintext`, которая выводится на экран (с завершающим символом верти-

кальной черты на случай, если в конце сообщения содержатся пробелы), а затем копируется в буфер обмена.

Дешифрование сообщения с помощью ключа

Функция `decryptMessage()` реализует шесть этапов описанной ранее процедуры дешифрования и возвращает результат в виде строки. Чтобы упростить процесс дешифрования, мы будем использовать ряд функций модуля `math`, импортированного программой ранее.

Функции `round()`, `math.ceil()` и `math.floor()`

Функция `round()` округляет вещественное число (число с десятичной точкой) до ближайшего целого числа. Функции `math.ceil()` и `math.floor()` (входящие в состав модуля `math`) выполняют округление вверх (до ближайшего большего по модулю целого числа) и вниз (до ближайшего меньшего по модулю целого числа) соответственно.

Результатом деления чисел с использованием оператора `/` является целое число в вещественном формате (число с плавающей точкой). Это происходит даже в тех случаях, когда числа делятся без остатка. Например, введите в интерактивной оболочке следующие инструкции.

```
>>> 21 / 7
3.0
>>> 22 / 5
4.4
```

Если хотите округлить число до ближайшего целого, используйте функцию `round()`. Чтобы увидеть, как она работает, введите следующие инструкции.

```
>>> round(4.2)
4
>>> round(4.9)
5
>>> round(5.0)
5
>>> round(22 / 5)
4
```

Если необходимо выполнить округление вверх, используйте функцию `math.ceil()`, если же требуется округление вниз — функцию `math.floor()`. Обе функции содержатся в модуле `math`, который необходимо импортировать, прежде чем вызывать их.

```
>>> import math
>>> 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
```

Функция `math.floor()` удаляет из вещественного числа десятичную точку и преобразует его в целое с округлением вниз, тогда как функция `math.ceil()` увеличивает вещественное число на единицу, преобразуя его в целое с округлением вверх.

Функция `decryptMessage()`

Функция `decryptMessage()` реализует все этапы процедуры дешифрования. В качестве аргументов ей передается целочисленное значение ключа (`key`) и строка сообщения (`message`). При вычислении количества столбцов для определения нужного числа ячеек используется функция `math.ceil()`.

```
19. def decryptMessage(key, message):
20.     # Функция расшифровки перестановочного шифра будет имитировать
21.     # столбцы и строки таблицы, в которые вписан простой текст,
22.     # используя список строк. Сначала вычислим ряд значений.
23.
24.     # Количество столбцов в перестановочной таблице
25.     numOfColumns = int(math.ceil(len(message) / float(key)))
26.     # Количество строк в таблице
27.     numOfRows = key
28.     # Количество "заштрихованных" ячеек в последнем столбце
29.     numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
```

В строке 25 для определения количества столбцов результат, возвращаемый функцией `len(message)`, делится на целочисленное значение ключа (`key`). Это значение передается функции `math.ceil()`, и полученный результат сохраняется в переменной `numOfColumns`. Чтобы обеспечить совместимость данной программы с Python 2, мы вызываем функцию `float()` для преобразования целого значения `key` в вещественное.

В Python 2 результат деления двух целых чисел автоматически округляется вниз. Вызов функции `float()` позволяет избежать этого, не влияя на поведение, свойственное версии Python 3.

В строке 27 вычисляется количество строк, которое равно целому значению, сохраненному в переменной `key`. Это значение записывается в переменную `numOfRows`.

В строке 29 вычисляется количество "заштрихованных" ячеек сетки, равное числу строк, умноженному на число столбцов, за вычетом длины сообщения.

При расшифровке сообщения "Cenoonommstmme oo snnio. s s c" с ключом 8 переменная `numOfColumns` будет равна 4, переменная `numOfRows` — 8, а переменная `numOfShadedBoxes` — 2.

Аналогично тому как в программе шифрования была предусмотрена переменная `ciphertext`, представляющая список строк в виде таблицы шифротекста, в функции `decryptMessage()` тоже предусмотрена переменная `plaintext`, представляющая список строк.

```
31.     # Каждая строка в списке plaintext представляет столбец
32.     plaintext = [''] * numOfColumns
```

Сначала этот список содержит пустые строки, по одной на каждый столбец таблицы. Используя операцию репликации, можно умножить список, содержащий одну строку, на количество столбцов (`numOfColumns`) и получить список из нескольких пустых строк, количество которых равно числу столбцов.

Имейте в виду, что эта переменная `plaintext` отличается от переменной `plaintext` в функции `main()`. Поскольку каждая из функций `decryptMessage()` и `main()` имеет собственную локальную область видимости, обе переменные `plaintext` совершенно разные, даже несмотря на то что их имена совпадают.

Вспомните, как выглядит таблица с текстом "Cenoonommstmme oo snnio. s s c" (см рис. 8.1). Переменная `plaintext` будет содержать список строк, и каждая строка в этом списке будет представлять собой отдельный столбец этой таблицы. Мы ожидаем, что список `plaintext` будет выглядеть так:

```
['Common s', 'ense is ', 'not so c', 'ommon.']
```

После этого мы сможем объединить все строки этого списка и вернуть строку `'Commonsense is not so common.'`.

Чтобы создать указанный список, прежде всего необходимо поместить каждый символ сообщения в соответствующую строку в списке `plaintext` по одному символу за раз. Для этого мы создадим две переменные, `column` и `row`, с помощью которых будем отслеживать столбец и строку ячейки, в которую должен быть помещен очередной символ. Начальными значениями этих переменных будут нули, что соответствует первому столбцу и первой строке. Это делается в строках 36 и 37.

```
34.     # Переменные column и row указывают, в какую ячейку должен
35.     # быть помещен символ зашифрованного сообщения
36.     column = 0
37.     row = 0
```

В строке 39 организуется цикл, который проходит по символам в строке `message`. В теле цикла содержится код, который регулирует переменные `column` и `row` таким образом, чтобы символ, содержащийся в переменной `symbol`, конкатенировался с соответствующей строкой в списке `plaintext`.

```
39.     for symbol in message:
40.         plaintext[column] += symbol
41.         column += 1  # указывает на следующий столбец
```

В строке 40 переменная `symbol` конкатенируется со строкой с индексом `column` в списке `plaintext`, поскольку каждая строка в этом списке представляет один столбец. Затем в строке 41 переменная `column` инкрементируется (т.е. ее значение увеличивается на 1), чтобы на следующей итерации цикла переменная `symbol` конкатенировалась со следующей строкой в списке `plaintext`.

Мы справились с инкрементированием переменных `column` и `row`, однако в некоторых случаях их значения необходимо сбрасывать в нуль. Но прежде чем перейти к рассмотрению соответствующего кода, нам предстоит узнать, что собой представляют булевы операторы.

Булевы операторы

Булевы операторы сравнивают булевы значения (или выражения, вычисление которых дает булево значение) и возвращают результат в виде булевого значения. С помощью булевых операторов `and` (логическое И) и `or` (логическое ИЛИ) можно формировать более сложные условия для инструкций `if` и `while`. Оператор `and` возвращает `True`, если результатом вычисления каждого из операндов является `True`; в противном случае

возвращается False. Оператор or возвращает True, если результатом вычисления хотя бы одного из операндов является True; в противном случае возвращается False. Чтобы увидеть, как работает оператор and, введите в интерактивной оболочке следующие выражения.

```
>>> 10 > 5 and 2 < 4
True
>>> 10 > 5 and 4 != 4
False
```

Результат первого выражения равен True, поскольку оба операнда равны True. Иными словами, поскольку значением каждого из выражений 10 > 5 и 2 < 4 является True, общий результат равен True.

Однако во второй инструкции, в которой также имеется выражение 10 > 5, равное True, выражение 4 != 4 равно False. Таким образом, операнды равны True и False соответственно. Поскольку результат операции and может быть равен True только в том случае, когда оба операнда равны True, в данном случае выражение в целом равно False.

Если вдруг забудете, как работает тот или иной булев оператор, взгляните на его *таблицу истинности*, отображающую результаты операции при различных комбинациях используемых булевых значений. Таблица истинности для оператора and приведена в табл. 8.1.

Таблица 8.1. Таблица истинности оператора and

A and B	Результат
True and True	True
True and False	False
False and True	False
False and False	False

Чтобы увидеть, как работает оператор or, введите в интерактивной оболочке следующие выражения.

```
>>> 10 > 5 or 4 != 4
True
>>> 10 < 5 or 4 != 4
False
```

В случае использования оператора or лишь один операнд должен быть равен True, чтобы результат был вычислен как True, и именно поэтому выражение 10 > 5 or 4 != 4 возвращает True. В то же время, поскольку

каждое из выражений `10 < 5` и `4 != 4` равно `False`, второе выражение вычисляется как `False or False`, что дает `False`.

Таблица истинности для оператора `or` приведена в табл. 8.2.

Таблица 8.2. Таблица истинности оператора `or`

A or B	Результат
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Есть еще третий булев оператор — `not` (логическое НЕ). Оператор `not` возвращает булево значение, противоположное операнду. Поэтому `not True` равно `False`, а `not False` равно `True`. Введите в интерактивной оболочке следующие инструкции.

```
>>> not 10 > 5
False
>>> not 10 < 5
True
>>> not False
True
>>> not not False
False
>>> not not not not not False
True
```

Как показывают последние два выражения, допускается использовать несколько операторов `not` подряд. Таблица истинности оператора `not` приведена в табл. 8.3.

Таблица 8.3. Таблица истинности оператора `not`

not A	Результат
`not True`	`False`
`not False`	`True`

Сокращение выражений с помощью операторов `and` и `or`

Аналогично тому как цикл `for` позволяет выполнить ту же работу, что и цикл `while`, но при меньшем объеме кода, операторы `and` и `or` также дают возможность немного сократить код. Введите в интерактивной оболочке

следующие два фрагмента кода, которые приводят к одному и тому же результату.

```
>>> if 10 > 5:
...   if 2 < 4:
...     print('Hello!')
...
Hello!
>>> if 10 > 5 and 2 < 4:
...   print('Hello!')
...
Hello!
```

Оператор and может заменить собой две инструкции if, которые проверяют каждую часть выражения по отдельности (где вторая инструкция if располагается внутри первой).

Можно также заменять инструкции if и elif оператором or. Чтобы убедиться в этом, введите в интерактивной оболочке следующие два равнозначных фрагмента.

```
>>> if 4 != 4:
...   print('Hello!')
... elif 10 > 5:
...   print('Hello!')
...
Hello!
>>> if 4 != 4 or 10 > 5:
...   print('Hello!')
...
Hello!
```

Каждая из инструкций if и elif проверяет разные части выражения, тогда как оператор or может выполнить всю проверку в одной строке.

Приоритет операций для булевых операторов

Вы уже знаете, что арифметические операторы имеют определенный приоритет, и операторы and, or и not не являются исключением. Сначала вычисляется оператор not, затем and и в последнюю очередь or. Введите в интерактивной оболочке следующие выражения.

```
>>> not False and False      # not False вычисляется раньше
False
>>> not (False and False)    # (False and False) вычисляется раньше
True
```

В первом случае сначала вычисляется выражение `not False`, поэтому строка приобретает вид `True and False`, и конечным результатом будет `False`. Во втором случае сначала вычисляется выражение в скобках, даже если ему предшествует оператор `not`, так что результатом выражения `False and False` будет `False`, а результатом выражения `not (False)` будет `True`.

Настройка переменных `column` и `row`

Теперь, когда вы знаете, как работают булевы операторы, мы можем перейти к вопросу о том, как сбрасываются значения переменных `column` и `row` в программе *transpositionDecrypt.py*.

Есть два случая, когда нам может понадобиться сброс значения переменной `column` в 0, чтобы на следующей итерации цикла символ из переменной `symbol` добавлялся в первую строку списка `plaintext`. Во-первых, такая необходимость возникает тогда, когда в результате инкрементирования значения `column` оно превысит последний индекс списка `plaintext`. В этой ситуации значение `column` будет равно `numOfColumns`. (Вспомните, что последним индексом в списке `plaintext` является `numOfColumns - 1`. Поэтому, если значение `column` равно `numOfColumns`, значит, мы превысили последний индекс.)

	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	. 23
6	■ 24	s 25	■ 26	
7	s 27	■ 28	c 29	

Рис. 8.2. *Таблица дешифровки с индексами столбцов и строк*

Вторая ситуация, требующая сброса переменной `column`, возникает тогда, когда последний индекс списка и переменная `row` указывают на заштрихованную ячейку в последнем столбце. В качестве наглядной иллюстрации на рис. 8.2 приведен вид таблицы с индексами столбцов, указанными сверху, и индексами строк, указанными слева.

Как видите, заштрихованные ячейки находятся в последнем столбце (индекс которого равен `numOfColumns - 1`) в строках 6 и 7. Чтобы вычислить индексы строк, которые будут содержать заштрихованные ячейки, используйте выражение `row >= numOfRows - numOfShadedBoxes`. В данном примере с восемью строками (с индексами от 0 до 7) заштрихованы

строки 6 и 7. Количество незаштрихованных строк равно общему количеству строк (в данном примере их 8) за вычетом заштрихованных ячеек (в данном примере их 2). Если текущий индекс строки `row` равен или превышает это число (8 – 2 = 6), то мы знаем, что попадаем в заштрихованную ячейку. Если указанное выражение истинно (равно `True`) и при этом значение `column` равно `numOfColumns - 1`, то мы переходим в заштрихованную ячейку, а значит, переменную `column` нужно обнулить для следующей итерации.

```
43.         # Если больше нет столбцов ИЛИ мы оказались в "заштрихованной"
44.         # ячейке, перейти к первому столбцу следующей строки
45.         if (column == numOfColumns) or (column == numOfColumns - 1
               and row >= numOfRows - numOfShadedBoxes):
46.             column = 0
47.             row += 1
```

Именно этими двумя случаями и объясняется появление в строке 45 условия `(column == numOfColumns) or (column == numOfColumns - 1 and row >= numOfRows - numOfShadedBoxes)`. И хотя данное выражение кажется большим и сложным, его можно разбить на меньшие части. Выражение `(column == numOfColumns)` проверяет, не выходит ли индекс столбца, содержащийся в переменной `column`, за допустимые пределы, а второе выражение следит за тем, чтобы пропускались заштрихованные ячейки. В случае истинности хотя бы одного из этих условий блок кода установит для переменной `column` нулевое значение, соответствующее первому столбцу. При этом также будет инкрементирована переменная `row`.

К тому времени, когда цикл `for`, запущенный в строке 39, завершит проход по всем символам, хранящимся в переменной `message`, список строк в переменной `plaintext` будет изменен таким образом, что в нем будет содержаться дешифрованный текст (при условии, что использовался корректный ключ). В строке 49 строки списка `plaintext` объединяются в одну строку с помощью строкового метода `join()`.

```
49.     return ''.join(plaintext)
```

Строка, сформированная функцией `decryptMessage()`, возвращается с помощью оператора `return`.

Таким образом, список `['Common s', 'ense is ', 'not so c', 'omm on.']`, содержащийся в переменной `plaintext`, будет преобразован вызовом `''.join(plaintext)` в строку `'Common sense is not so common.'`.

Вызов функции main()

Первой инструкцией, которая будет выполнена после импорта модулей и выполнения инструкции def, будет инструкция if в строке 54.

```
52. # Если файл transpositionDecrypt.py выполняется как программа
53. # (а не импортируется как модуль), вызвать функцию main()
54. if __name__ == '__main__':
55.     main()
```

Как и в случае программы шифрования перестановочного шифра, Python выясняет, был ли запущен данный файл в виде программы (а не импортирован как модуль). Для этого проверяется, содержит ли переменная __name__ строку '__main__'. Если это действительно так, выполняется функция main().

Резюме

На этом рассмотрение нашей программы дешифрования заканчивается. Большую часть программы составляет функция decryptMessage(). Написанные нами программы позволяют зашифровать и расшифровывать сообщение "Common sense is not so common." с ключом 8. В то же время вы должны перепробовать несколько других сообщений и ключей и убедиться в том, что сообщения после дешифровки совпадают с их оригинальными версиями. Если полученный результат не будет совпадать с ожидаемым, значит, не работает либо код, ответственный за шифрование сообщения, либо код, выполняющий его дешифровку. В главе 9 мы автоматизируем процесс проверки, написав программу, которая будет тестировать наши программы.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Воспользовавшись карандашом и бумагой, расшифруйте приведенные ниже сообщения с помощью ключа 9. Символ ■ обозначает одиночный пробел. Для удобства в конце каждого сообщения указано количество содержащихся в нем символов.
   - H■cb■■irhdeuousBdi■■■prrtyevdgp■nir■■eerit■eatoreechadihf■pak en■ge■b■te■dih■aoa.da■tts■tn (89 символов).
   - A■b■■drottthawa■nwar■eci■t■nlel■ktShw■leec,hheat■.na■■e■soog mah■a■■ateniAcgakh■dmnor■■ (86 символов).
   - Bmmsrl■dpnaua!toeboo'ktn■uknrwos.■yaregonr■w■nd,tu■■oiady■h gtRwt■■■A■hhanhhasthtev■■e■t■e■■eo (93 символа).
2. Если ввести в интерактивной оболочке приведенные ниже выражения, то что будет выведено на экран каждой строкой?

```
>>> math.ceil(3.0)
>>> math.floor(3.1)
>>> round(3.1)
>>> round(3.5)
>>> False and False
>>> False or False
>>> not not True
```

3. Составьте таблицы истинности для операторов `and`, `or` и `not`.
4. Какая из приведенных ниже инструкций корректна?

```
if __name__ == '__main__':
if __main__ == '__name__':
if _name_ == '_main_':
if _main_ == '_name_':
```

9

НАПИСАНИЕ ТЕСТОВ

"Нельзя считать хорошей общественную практику, поощряющую внедрение технологий, которые в один прекрасный день могут обернуться полицейским государством".

Брюс Шнайер, "Секреты и ложь"

По всей видимости, наши программы для работы с перестановочным шифром неплохо справляются с шифрованием и дешифрованием самых разных сообщений с различными ключами, но откуда нам знать, *всегда* ли так будет? Нельзя быть абсолютно уверенным в том, что программы всегда будут работать правильно, если только не протестировать функции `encryptMessage()` и `decryptMessage()` со всеми возможными видами сообщений и значениями ключа. Но на это уйдет масса времени, так как придется набирать текст сообщения в программе шифрования, устанавливать ключ, запускать программу, копировать шифротекст в программу дешифрования, задавать ключ там и запускать дешифрование. Причем все это придется делать многократно с различными ключами и сообщениями, а это огромный объем утомительной работы!

Вместо этого мы напишем другую программу, которая будет генерировать случайные сообщения и случайные ключи для тестирования программ шифрования. Эта программа будет шифровать сообщение с помо-

щью функции `encryptMessage()` из программы *transpositionEncrypt.py*, а затем передавать шифротекст функции `decryptMessage()` из программы *transpositionDecrypt.py*. Если простой текст, возвращенный функцией `decryptMessage()`, совпадет с оригинальным сообщением, то программа-тестер будет знать, что программы шифрования и дешифрования работают нормально. Процесс автоматического тестирования программы с помощью другой программы называется *автоматизированным тестированием*.

Испытанию должны подвергнуться многочисленные комбинации сообщений и ключей, и с тестированием тысяч таких комбинаций компьютер справится всего за одну-две минуты. Если все тесты будут пройдены успешно, то можно быть уверены в том, что наш код работает правильно.

В этой главе...

- Функция `random.randint()`
- Функция `random.seed()`
- Ссылки на списки
- Функции `copy.deepcopy()`
- Функция `random.shuffle()`
- Перемешивание строки
- Функция `sys.exit()`

Исходный код программы `Transposition Test`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *transpositionTest.py*. Затем выполните программу, нажав клавишу <F5>.

transpositionTest.py

```
 1. # Тестирование перестановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import random, sys, transpositionEncrypt, transpositionDecrypt
 5.
 6. def main():
 7.     random.seed(42)  # задание "затравочного" значения
 8.
 9.     for i in range(20):  # выполняем 20 тестов
10.         # Генерирование случайных сообщений для тестирования
11.
12.         # Сообщение будет иметь произвольную длину
```

```
13.         message = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' * random.randint(4, 40)
14.
15.         # Преобразование строки в список для перемешивания
16.         message = list(message)
17.         random.shuffle(message)
18.         message = ''.join(message)  # обратное преобразование в строку
19.
20.         print('Test #%s: "%s..."' % (i + 1, message[:50]))
21.
22.         # Проверяем все возможные ключи для каждого сообщения
23.         for key in range(1, int(len(message)/2)):
24.             encrypted =
                    transpositionEncrypt.encryptMessage(key, message)
25.             decrypted =
                    transpositionDecrypt.decryptMessage(key, encrypted)
26.
27.             # Если дешифрованный текст не совпадает с оригинальным,
28.             # вывести сообщение об ошибке и завершиться
29.             if message != decrypted:
30.                 print('Mismatch with key %s and message %s.' %
                          (key, message))
31.                 print('Decrypted as: ' + decrypted)
32.                 sys.exit()
33.
34.     print('Transposition cipher test passed.')
35.
36.
37. # Если файл transpositionTest.py выполняется как программа
38. # (а не импортируется как модуль), вызвать функцию main()
39. if __name__ == '__main__':
40.     main()
```

Пример выполнения программы `Transposition Test`

Выполнив программу *transpositionTest.py*, вы должны получить следующие результаты.

```
Test #1: "JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQ..."
Test #2: "SWRCLUCRDOMLWZKOMAGVOTXUVVEPIOJMSBEQRQOFRGCCKENINV..."
Test #3: "BIZBPZUIWDUFXAPJTHCMDWEGHYOWKWWWSJYKDQVSFWCJNCOZZA..."
Test #4: "JEWBCEXVZAILLCHDZJCUTXASSZZRKRPMYGTGHBXPQPBEBVCODM..."
--опущено--
Test #17: "KPKHHLPUWPSSIOULGKVEFHZOKBFHXUKVSEOWOENOZSNIDELAWR..."
Test #18: "OYLFXXZENDFGSXTEAHGHPBNORCFEPBMITILSSJRGDVMNSOMURV..."
Test #19: "SOCLYBRVDPLNVJKAFDGHCQMXIOPEJSXEAAXNWCCYAGZGLZGZHK..."
Test #20: "JXJGRBCKZXPUIEXOJUNZEYYSEAEGVOJWIRTSSGPUWPNZUBQNDA..."
Transposition cipher test passed.
```

Работа программы-тестера начинается с импорта программ *transpositionEncrypt.py* и *transpositionDecrypt.py* как модулей. Из них будут вызываться функции `encryptMessage()` и `decryptMessage()`. Далее программа-тестер создает случайное сообщение и случайный ключ. Тот факт, что это сообщение представляет собой лишь случайный набор букв, не имеет никакого значения, так как программе нужно всего-навсего проверить, что в результате шифрования и последующего дешифрования сообщения мы получим его исходную версию.

Программа повторяет этот тест в цикле 20 раз. Если на каком-то этапе окажется, что строка, возвращенная функцией `transpositionDecrypt()`, не совпадает с исходным сообщением, программа выдаст сообщение об ошибке и завершит работу.

Перейдем к более подробному изучению приведенного выше исходного кода.

Импорт модулей

Работа программы начинается с импорта модулей, в том числе тех двух, с которыми вы уже встречались и которые поставляются вместе с Python: `random` и `sys`.

```
1. # Тестирование перестановочного шифра
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import random, sys, transpositionEncrypt, transpositionDecrypt
```

Мы также должны импортировать программы для работы с перестановочным шифром (т.е. *transpositionEncrypt.py* и *transpositionDecrypt.py*), введя только их имена без расширения *.py*.

Создание псевдослучайных чисел

Для создания случайных чисел, с помощью которых будут генерироваться сообщения и ключи, мы воспользуемся функцией `seed()` из модуля `random`. Но прежде чем обсуждать, что именно она делает, рассмотрим, как в Python организуется работа со случайными числами на примере функции `random.randint()`, которая тоже будет задействована в нашей программе. Эта функция имеет два целочисленных аргумента и возвращает случайное целое число из заданного интервала (включая значения самих аргументов). Введите в интерактивной оболочке следующие инструкции.

```
>>> import random
>>> random.randint(1, 20)
20
>>> random.randint(1, 20)
18
>>> random.randint(100, 200)
107
```

Разумеется, полученные вами числа, в силу того что они случайные, наверняка будут другими.

Однако числа, генерируемые функцией `random.randint()`, не являются в полном смысле случайными. Они создаются с помощью алгоритма генерации псевдослучайных чисел, который инициализируется начальным числом и формирует последующие числа на основании формулы.

Начальное число, с которого начинает свою работу генератор псевдослучайных чисел, называется *затравкой* (seed). Если известна затравка, то остальные числа, создаваемые генератором, будут предсказуемыми, поскольку они генерируются в одном и том же порядке. Такие числа, которые выглядят как случайные, но на самом деле являются предсказуемыми, называются *псевдослучайными*. Программы на языке Python, для которых вы не задаете затравку, используют в качестве начального значения текущее системное время. Случайную затравку Python можно изменить, вызвав функцию `random.seed()`.

Чтобы убедиться в том, что псевдослучайные числа не являются случайными в полном смысле этого слова, введите в интерактивной оболочке следующие инструкции.

```
  >>> import random
❶ >>> random.seed(42)
❷ >>> numbers = []
  >>> for i in range(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)
  >>> numbers = []
  >>> for i in range(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]
```

Здесь мы дважды генерируем 20 случайных чисел, используя одну и ту же затравку. Прежде всего мы импортируем модуль `random` и задаем `42` в качестве затравки (❶). Затем мы создаем список `numbers` (❷), в котором

сохраняются сгенерированные числа. С помощью цикла `for` мы генерируем 20 чисел и присоединяем каждое из них к списку `numbers`, значение которого автоматически выводится на экран, так что мы можем увидеть каждое из сгенерированных чисел (❸).

Если в качестве затравки для генератора псевдослучайных чисел Python установлено значение 42, то первым случайным числом в интервале между 1 и 10 всегда будет 2. Вторым числом всегда будет 1, третьим — 5 и т.д. Если вновь установить для затравки значение 42 и сгенерировать цепочку чисел, то функция `random.randint()` вернет тот же набор псевдослучайных чисел, в чем нетрудно убедиться, сравнив списки ❸ и ❹.

Случайные числа будут особенно важны для шифров в последующих главах, поскольку они находят применение не только в тестировании, но и в самих программах шифрования и дешифрования на основе более сложных шифров. Важность случайных чисел очень велика, так как наиболее распространенной уязвимостью шифровальных программ является использование предсказуемых цепочек чисел. Если случайные числа в вашей программе окажутся предсказуемыми, то криптоаналитик сможет воспользоваться этой информацией для взлома шифра.

Выбор истинно случайных чисел в качестве ключей шифрования крайне необходим для того, чтобы обеспечить безопасность шифра, но если речь идет о задачах другого рода, таких как тестирование, то псевдослучайные числа можно считать вполне приемлемыми. Псевдослучайные числа будут использоваться нами для генерирования тестовых строк, передаваемых программе-тестеру. Чтобы сгенерировать истинные случайные числа с помощью Python, следует использовать функцию `random.SystemRandom().randint()`, описание которой доступно по следующему адресу:

`https://docs.python.org/3/library/random.html#random.SystemRandom`

Создание случайной строки

Теперь, когда вы узнали о том, как использовать функции `random.randint()` и `random.seed()` для генерирования случайных чисел, вернемся к исходному коду. Чтобы полностью автоматизировать процесс шифрования и дешифрования, нам необходимо научиться автоматически генерировать случайные строки сообщений.

Для этого возьмем строку символов, которые будут использоваться в сообщениях, продублируем ее произвольное количество раз и сохраним результат в виде строки. После этого перемешаем символы в результирующей строке, чтобы сделать ее более случайной. Мы будем генерировать новую случайную строку для каждого теста, чтобы протестировать множество различных комбинаций букв.

Прежде всего определим функцию `main()`, которая будет содержать код тестирования. Она начинается с задания затравочного значения для генератора псевдослучайной строки.

```
6. def main():
7.     random.seed(42)  # задание "затравочного" значения
```

Задавать в программе-тестере затравку с помощью функции `random.seed()` очень удобно, поскольку получение предсказуемого набора чисел позволяет получать одни и те же псевдослучайные сообщения и ключи при каждом запуске программы. Таким образом, если вы заметите, что одно из сообщений не удается зашифровать и дешифровать надлежащим образом, вы сможете воспроизвести эту ошибку и попытаться устранить причину ее возникновения.

Следующим шагом является дублирование строки в цикле `for`.

Дублирование строки произвольное число раз

Для запуска 20 тестов и генерирования случайных сообщений мы будем использовать цикл `for`.

```
 9.     for i in range(20):  # выполняем 20 тестов
10.         # Генерирование случайных сообщений для тестирования
11.
12.         # Сообщение будет иметь произвольную длину
13.         message = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' * random.randint(4, 40)
```

На каждой итерации цикла `for` программа будет создавать и тестировать новое сообщение. Мы хотим, чтобы программа выполнила множество тестов, поскольку чем больше тестов будет успешно пройдено, тем выше будет наша уверенность в том, что программа работает корректно.

В строке 13 создается сообщение произвольной длины. Для этого берется строка, содержащая буквы в верхнем регистре, которая реплицируется путем умножения на случайное число в интервале от 4 до 40, возвращаемое функцией `randint()`. Результирующая строка сохраняется в переменной `message`.

Если мы оставим строку `message` в том виде, в каком она есть, то она всегда будет представлять собой строку из букв алфавита, повторенную случайное количество раз. Поскольку мы хотим тестировать различные комбинации символов, мы должны сделать еще один шаг и перемешать символы в переменной `message`. Но прежде чем это сделать, следует поближе познакомиться со списками.

Списковые переменные используют ссылки

Списки сохраняются в переменных не так, как другие типы значений. Переменная содержит не сам список, а ссылку на него. *Ссылка* — это значение, которое указывает на ячейку с данными, а *ссылка на список* — это значение, которое указывает на список. Работа со ссылками имеет свои особенности.

Вы уже знаете, что в переменных могут храниться строки и целые числа. Введите в интерактивной оболочке следующие инструкции.

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

Мы присваиваем значение `42` переменной `spam`, а затем копируем значение `spam` и присваиваем его переменной `cheese`. Если впоследствии мы заменим значение `spam` значением `100`, то это новое число не повлияет на значение `cheese`, поскольку `spam` и `cheese` — разные переменные, которые хранят разные значения.

Однако списки работают иначе. Присваивая список переменной, мы в реальности присваиваем ей ссылку на этот список. Попробуем во всем разобраться. Введите в интерактивной оболочке следующие инструкции.

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

Поведение этого кода может показаться вам несколько странным. Мы меняем лишь список `cheese`, однако это приводит к одновременному изменению списка `spam`.

Когда мы создаем список (❶), мы присваиваем ссылку на него переменной `spam`. Однако следующая инструкция (❷) копирует в переменную `cheese` лишь ссылку на список, а не сами значения списка. Это означает, что значения, хранящиеся в переменных `spam` и `cheese`, теперь указывают на один и тот же список. Существует лишь один базовый список, поскольку собственно список даже не копировался. В результате, если мы изменяем

элемент `cheese[1]` (❸), то изменяем тот же список, на который ссылается переменная `spam`.

Вспомните о том, что переменные можно уподобить ящикам, содержащим значения. Однако списковые переменные фактически не содержат сами списки — они содержат ссылки на них. (Ссылки основаны на внутренних идентификаторах Python, но об этом можно не думать.) На рис 9.1 показано, что происходит, когда список присваивается переменной `spam`.

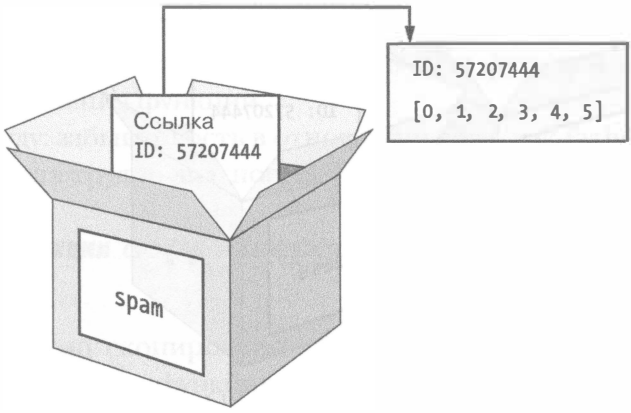


Рис. 9.1. *Инструкция `spam = [0, 1, 2, 3, 4, 5]` сохраняет в переменной не сам список, а ссылку на него*

Затем ссылка, сохраненная в переменной `spam`, копируется в переменную `cheese` (рис. 9.2). При этом в переменной `cheese` создается и сохраняется не новый список, а ссылка на него.

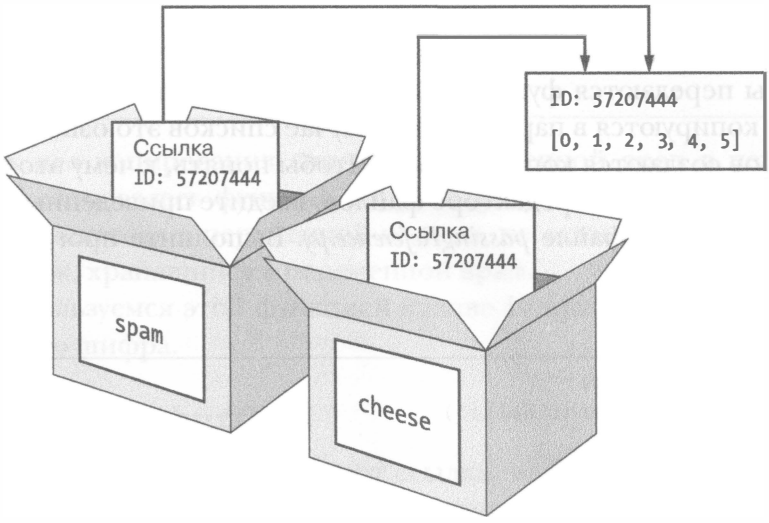


Рис. 9.2. *Инструкция `cheese = spam` копирует ссылку на список, а не сам список*

Если мы изменим список, на который ссылается переменная `cheese`, то список, на который ссылается переменная `spam`, тоже изменится, поскольку обе переменные ссылаются на один и тот же список (рис. 9.3).

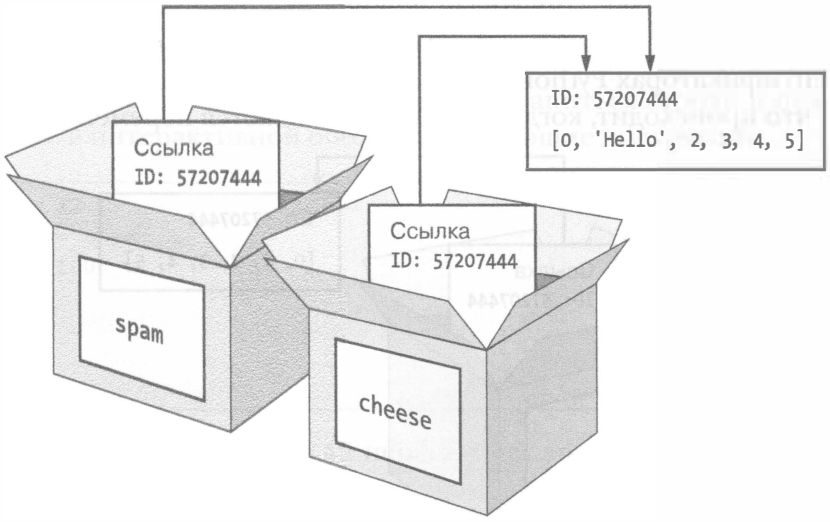


Рис. 9.3. *Инструкция* `cheese[1] = 'Hello!'` *изменяет список, на который ссылаются обе переменные*

Несмотря на то что с технической точки зрения переменные Python содержат ссылки на списки, на это часто не обращают внимания и просто говорят, что переменная "содержит список".

Передача ссылок

Знание ссылок особенно важно для понимания того, каким образом аргументы передаются функции. Во время вызова функции значения аргументов копируются в параметры. В случае списков это означает, что для параметров создаются копии ссылок. Чтобы понять, к чему это приводит, откройте новое окно в редакторе файлов, введите приведенный ниже код и сохраните его в файле *passingReference.py*. Выполните программу, нажав клавишу <F5>.

passingReference.py

```
def eggs(someParameter):
    someParameter.append('Hello')
spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Обратите внимание на то, что значение переменной spam изменяется не путем присваивания ей значения, возвращаемого функцией eggs(), а путем непосредственного изменения списка. В результате запуска программы вы должны получить следующий результат:

```
[1, 2, 3, 'Hello']
```

Несмотря на то что переменные spam и someParameter содержат разные ссылки, они обе ссылаются на один и тот же список. Вот почему результат вызова метода append('Hello') в коде функции остается в силе даже после завершения функции.

Имейте в виду: забывчивость в отношении того, как Python обрабатывает списки, чревата трудно диагностируемыми ошибками.

Использование функции copy.deepcopy() для дублирования списка

Если необходимо скопировать содержимое списка, то импортируйте модуль copy и вызовите функцию copy.deepcopy(), которая выполняет так называемое *глубокое копирование* и возвращает копию переданного ей списка.

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> import copy
>>> cheese = copy.deepcopy(spam)
>>> cheese[1] = 'Hello!'
>>> spam
[0, 1, 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Так как для копирования списка из переменной spam в переменную cheese использовалась функция copy.deepcopy(), изменение элемента списка, хранящегося в переменной cheese, не оказывает никакого влияния на список, хранящийся в переменной spam.

Мы воспользуемся этой функцией в главе 17 при взломе простого подстановочного шифра.

Функция random.shuffle()

Теперь, зная о том, как работают ссылки, вам будет легче понять, как работает функция random.shuffle(), которую мы используем далее. Эта функция, являющаяся частью модуля random, получает аргумент в виде

списка, переставляя его элементы случайным образом. Чтобы увидеть, как она работает, введите в интерактивной оболочке следующие инструкции.

```
>>> import random
>>> 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]
```

Следует обратить внимание на одну важную деталь: *функция* `shuffle()` *не возвращает список*. Вместо этого она изменяет содержимое переданного ей списка напрямую (поскольку получает ссылку на список). Вот почему мы используем вызов `random.shuffle(spam)`, а не инструкцию `spam = random.shuffle(spam)`.

Случайное перемешивание строки

Вернемся к программе *transpositionTest.py*. Чтобы перемешать символы строки, мы должны предварительно преобразовать строку в список, используя функцию `list()`.

```
15.         # Преобразование строки в список для перемешивания
16.         message = list(message)
17.         random.shuffle(message)
18.         message = ''.join(message)  # обратное преобразование в строку
```

Функция `list()` возвращает список односимвольных строк, соответствующих символам переданной ей строки. Поэтому в строке 16 мы назначаем новую переменную `message` для хранения списка символов. Далее функция `shuffle()` рандомизирует порядок следования элементов списка `message`. Затем программа преобразует список обратно в строку с помощью строкового метода `join()`. Благодаря такому перемешиванию мы можем тестировать множество различных сообщений.

Тестирование каждого сообщения

Создав случайное сообщение, программа тестирует с его помощью функций шифрования и дешифрования. В качестве обратной связи с про-

граммой мы выводим определенную информацию, чтобы можно было наблюдать за ходом тестирования:

```
20.         print('Test #%s: "%s..."' % (i + 1, message[:50]))
```

Строка 20 содержит вызов функции `print()`, которая отображает номер выполняемого теста (мы прибавляем 1 к `i`, поскольку `i` начинается с 0, а номера тестов — с 1). Так как строка `message` может быть слишком длинной, мы используем срез строки, чтобы отображать только первые 50 символов сообщения.

В строке 20 применяется строковое форматирование. Значение `i + 1` заменяет первую пару символов `%s` в строке, а значение `message[:50]` — вторую пару символов `%s`. Используя строковое форматирование, следите за тем, чтобы количество символов `%s` в строке совпадало с количеством аргументов, указанных после нее в круглых скобках.

Далее мы тестируем все возможные ключи. Если ключ для шифра Цезаря мог иметь целочисленные значения в пределах от 0 до 65 (длина символьного набора), то значения ключа для перестановочного шифра могут находиться в пределах от 1 до половины длины сообщения. В цикле `for`, который начинается в строке 23, выполняются соответствующие проверки.

```
22.         # Проверяем все возможные ключи для каждого сообщения
23.         for key in range(1, int(len(message)/2)):
24.             encrypted =
                    transpositionEncrypt.encryptMessage(key, message)
25.             decrypted =
                    transpositionDecrypt.decryptMessage(key, encrypted)
```

В строке 24 строка `message` шифруется с помощью функции `encryptMessage()`. Так как эта функция содержится в файле *transpositionEncrypt.py*, мы должны снабдить ее префиксом `transpositionEncrypt.` (с точкой в конце).

Затем зашифрованная строка, возвращенная функцией `encryptMessage()`, передается функции `decryptMessage()`. В обоих вызовах функции мы должны использовать один и тот же ключ. Значение, возвращенное функцией `decryptMessage()`, сохраняется в переменной `decrypted`. Если функции работают корректно, то строки, содержащиеся в переменных `message` и `decrypted`, должны совпадать. Рассмотрим, как программа выполняет эту проверку.

Проверка корректности результата и завершение программы

После того как мы зашифровали и дешифровали сообщение, нужно проверить, действительно ли обе функции отработали корректно. Для этого достаточно проверить, что дешифрованное сообщение совпадает с исходным.

```
27.             # Если дешифрованный текст не совпадает с оригинальным,
28.             # вывести сообщение об ошибке и завершиться
29.             if message != decrypted:
30.                 print('Mismatch with key %s and message %s.' %
                          (key, message))
31.                 print('Decrypted as: ' + decrypted)
32.                 sys.exit()
33.
34.     print('Transposition cipher test passed.')
```

В строке 29 проверяется, равны ли переменные `message` и `decrypted`. Если это не так, программа отобразит соответствующую информацию. В строках 30 и 31 выводятся значения `key`, `message` и `decrypted` в качестве обратной связи, чтобы можно было определить, где именно возникла проблема. Затем программа завершается.

Обычно выход из программы осуществляется по достижении конца файла, когда больше не остается строк кода, подлежащих выполнению. Но как только встречается вызов `sys.exit()`, работа программы сразу же завершается, и новые сообщения не тестируются. (Программа нуждается в исправлении, если хотя бы один из тестов завершился ошибкой!)

Но если значения переменных `message` и `decrypted` равны, то программа игнорирует блок кода `if`, в том числе и вызов `sys.exit()`. Программа продолжит выполнение цикла до тех пор, пока не исчерпает все тесты. По завершении цикла выполняется строка 34, которая, как вы уже знаете, находится вне цикла, запускаемого в строке 9, поскольку имеет другой отступ. В строке 34 выводится текст `'Transposition cipher test passed.'`, указывающий на успешное завершение тестирования.

Вызов функции `main()`

Как и в случае других программ, мы проверяем, импортируется ли файл как модуль или выполняется как основная программа.

```
37. # Если файл transpositionTest.py выполняется как программа
38. # (а не импортируется как модуль), вызвать функцию main()
39. if __name__ == '__main__':
40.     main()
```

В строках 39 и 40 проверяется, содержит ли специальная переменная `__name__` значение `'__main__'`, и если это так, то вызывается функция `main()`.

Тестирование программы-тестера

Мы написали программу, которая тестирует функции шифрования и дешифрования перестановочного шифра, но как узнать, что она работает правильно? Что, если в программе-тестере имеются логические ошибки и она лишь сообщает нам, что функции шифрования/дешифрования работают, хотя на самом деле это не так?

Мы можем протестировать саму программу-тестер, намеренно введя ошибки в функции шифрования/дешифрования. Тогда, если программа-тестер не обнаружит проблему, мы будем знать, что она работает не так, как ожидается.

Чтобы внести в программу логическую ошибку, откройте файл *transpositionEncrypt.py* и добавьте `+ 1` в строке 36.

transpositionEncrypt.py

```
35.             # Увеличить значение currentIndex
36.             currentIndex += key + 1
```

Теперь, если запустить программу-тестер, она должна вывести следующее сообщение об ошибке.

```
Test #1: "JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQ..."
Mismatch with key 1 and message
JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQTRGJHDVCZECRESZJARAVIPFOBWZ
XXTBFOFHVSIGBWIBBHGKUWHEUUDYONYTZVKNVVTYZPDDMIDKBHTYJAHBNDVJUZDCEMFMLUXEONCZX
WAWGXZSFTMJNLJOKKIJXLWAPCQNYCIQOFTEAUHRJODKLGRIZSJBXQPBMQPPFGMVUZHKFWPGNMRYXR
OMSCEEXLUSCFHNELYPYKCNYTOUQGBFSRDDMVIGXNYPHVPQISTATKVKM.
Decrypted as:
JQDKZACYCPTRLHBQEWLWGQRIITGHVZCEZAAIFBZXBOHSGWBHKWEUYNTVNVYPDIKHYABDJZCMMUENZ
WWXSTJLOKJLACNCQFEUROKGISBQBQPGVZKWGMYRMCELSFNLPKNTUGFRDVGNPVQSAKK
```

После того как мы намеренно ввели ошибку, программа споткнулась на первом же сообщении, как и ожидалось!

Резюме

Приобретенные к этому времени навыки программирования пригодятся вам не только для того, чтобы писать программы шифрования и дешифрования. Теперь вы умеете тестировать программы и можете убедиться в

том, что они корректно работают для различных входных данных. Написание тестирующего кода — распространенная задача в программировании.

В этой главе вы узнали о том, как применять функцию `random.randint()` для получения псевдослучайных чисел, а функцию `random.seed()` — для переустановки затравочного значения. Несмотря на то что псевдослучайные числа не подходят для использования в криптографических программах, они вполне годятся для программы-тестера, которая рассматривалась в этой главе.

Кроме того, вы узнали о различиях между списками и ссылками на них и о том, как с помощью функции `copy.deepcopy()` создавать копии списков. Дополнительно вы научились перемешивать порядок следования элементов списка с помощью функции `random.shuffle()`.

Все программы, которые мы создали до сих пор, шифруют лишь короткие сообщения. В главе 10 вы узнаете о том, как шифровать и дешифровывать целые файлы.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Если вы запустили приведенную ниже программу и она вывела на экран число 8, то что будет выведено на экран, когда вы запустите программу в следующий раз?

```
import random
random.seed(9)
print(random.randint(1, 10))
```

2. Что выведет на экран следующая программа?

```
spam = [1, 2, 3]
eggs = spam
ham = eggs
ham[0] = 99
print(ham == spam)
```

3. В каком модуле содержится функция `deepcopy()`?
4. Что выведет на экран следующая программа?

```
import copy
spam = [1, 2, 3]
eggs = copy.deepcopy(spam)
ham = copy.deepcopy(eggs)
ham[0] = 99
print(ham == spam)
```

10

ШИФРОВАНИЕ И ДЕШИФРОВАНИЕ ФАЙЛОВ

"Зачем службе государственной безопасности
хватать людей и подвергать их пыткам?
Чтобы получить от них информацию.
Но жесткие диски не сопротивляются пыткам.
Вы должны наделить их способностью к сопротивлению.
В этом и заключается суть криптографии".

Патрик Болл, Группа анализа данных
по правам человека

В предыдущих главах мы работали только с небольшими сообщениями, которые вводились непосредственно в код в строковом виде. Программа шифрования, которую мы создадим в этой главе, позволит шифровать и дешифровать целые файлы, размеры которых могут исчисляться миллионами символов.

В этой главе...

- Функция `open()`
- Чтение и запись файлов
- Методы `write()`, `close()` и `read()` файлового объекта
- Функция `os.path.exists()`
- Строковые методы `upper()`, `lower()` и `title()`
- Строковые методы `startswith()` и `endswith()`
- Модуль `time` и функция `time.time()`

Простые текстовые файлы

Программа для работы с перестановочным шифром шифрует и дешифрует простые (неформатированные) текстовые файлы. Файлы этого типа содержат только текст и обычно имеют расширение *.txt*. Создавать подобные файлы можно с помощью таких программ, как Блокнот в Windows, TextEdit в macOS и `gedit` в Linux. (Текстовые процессоры тоже позволяют создавать простые текстовые файлы, но учитывайте, что информация о шрифтах, цвете и других элементах форматирования текста будет утеряна.) Для этой цели подойдет даже файловый редактор IDLE, если сохранять файлы с расширением *.txt* вместо привычного *.py*.

Несколько текстовых файлов доступно на сайте издательства (см. введение). Эти образцы текста взяты из книг, которые в настоящее время находятся в свободном доступе, а потому доступны для загрузки и использования на законных основаниях. Например, текстовый файл классического романа Мэри Шелли *Франкенштейн* содержит 78 000 слов! Если бы текст такой книги пришлось вводить вручную, то это заняло бы у нас массу времени, но благодаря использованию готового файла программа справится с его шифрованием буквально за пару секунд.

Исходный код программы `Transposition File Cipher`

Как и в случае программы-тестера для перестановочного шифра, программа шифрования файлов импортирует файлы *transpositionEncrypt.py* и *transpositionDecrypt.py*, чтобы можно было использовать функции `encryptMessage()` и `decryptMessage()`. Благодаря этому вам не придется заново набирать код функций в новой программе.

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите приведенный ниже код и сохраните его в файле *transpositionFileCipher.py*. После этого скопируйте файл *frankenstein.txt* из папки с примерами книги и поместите в тот же каталог, в котором находится файл *transpositionFileCipher.py*. Запустите программу, нажав клавишу <F5>.

transpositionFileCipher.py

```
# Шифрование/дешифрование файлов с помощью перестановочного шифра
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import time, os, sys, transpositionEncrypt, transpositionDecrypt

def main():
    inputFilename = 'frankenstein.txt'
    # БУДЬТЕ ВНИМАТЕЛЬНЫ: если файл с именем outputFilename
    # уже существует, программа перезапишет его
    outputFilename = 'frankenstein.encrypted.txt'
    myKey = 10
    myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'

    # Если входного файла не существует, программа завершается
    if not os.path.exists(inputFilename):
        print('The file %s does not exist. Quitting...' %
              (inputFilename))
        sys.exit()

    # Если выходной файл существует, задать пользователю вопрос
    if os.path.exists(outputFilename):
        print('This will overwrite the file %s. (C)ontinue or
              (Q)uit?' %(outputFilename))
        response = input('> ')
        if not response.lower().startswith('c'):
            sys.exit()

    # Прочитать сообщение из входного файла
    fileObj = open(inputFilename)
    content = fileObj.read()
    fileObj.close()

    print('%sing...' % (myMode.title()))

    # Измерить, как долго длится шифрование/дешифрование
    startTime = time.time()
    if myMode == 'encrypt':
        translated = transpositionEncrypt.encryptMessage(myKey, content)
    elif myMode == 'decrypt':
        translated = transpositionDecrypt.decryptMessage(myKey, content)
```

```
39.     totalTime = round(time.time() - startTime, 2)
40.     print('%sion time: %s seconds' % (myMode.title(), totalTime))
41.
42.     # Записать транслированное сообщение в выходной файл
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
46.
47.     print('Done %sing %s (%s characters).' % (myMode, inputFilename,
            len(content)))
48.     print('%sed file is %s.' % (myMode.title(), outputFilename))
49.
50.
51. # Если файл transpositionCipherFile.py выполняется как программа
52. # (а не импортируется как модуль), вызвать функцию main()
53. if __name__ == '__main__':
54.     main()
```

Пример выполнения программы `Transposition File Cipher`

Выполнив программу *transpositionFileCipher.py*, вы должны получить следующие результаты.

```
Encrypting...
Encryption time: 1.21 seconds
Done encrypting frankenstein.txt (441034 characters).
Encrypted file is frankenstein.encrypted.txt.
```

Новый файл *frankenstein.encrypted.txt* создается в той же папке, что и файл *transpositionFileCipher.py*. Если вы откроете этот файл с помощью редактора IDLE, то увидите зашифрованное содержимое файла *frankenstein.py*. Оно должно выглядеть примерно так.

```
PtFiyedleo a arnvmt eneeGLchongnes Mmuyedlsu0#uiSHTGA r sy,n t ys
s nuaoGeL
sc7s,
--опущено--
```

Располагая зашифрованным текстом, вы сможете переслать его тому, кто его расшифрует. Для этого у получателя сообщения тоже должна быть программа *transpositionFileCipher.py*.

Чтобы дешифровать текст, внесите следующие изменения в исходный код (выделены полужирным шрифтом) и вновь запустите программу.

```
 7.     inputFilename = 'frankenstein.encrypted.txt'
 8.     # БУДЬТЕ ВНИМАТЕЛЬНЫ: если файл с именем outputFilename
 9.     # уже существует, программа перезапишет его
10.     outputFilename = 'frankenstein.decrypted.txt'
11.     myKey = 10
12.     myMode = 'decrypt'  # задать 'encrypt' или 'decrypt'
```

Теперь, когда вы запустите программу, в папке появится новый файл *frankenstein.decrypted.txt*, содержимое которого совпадает с содержимым исходного файла *frankenstein.txt*.

Работа с файлами

Прежде чем перейти к изучению кода программы *transpositionFileCipher.py*, необходимо узнать о том, как Python работает с файлами. Для чтения содержимого файла потребуется выполнить следующие три действия: открыть файл, загрузить его содержимое в переменную и закрыть его. Аналогичным образом, чтобы записать новое содержимое в файл, необходимо открыть (или создать) файл, записать в него новое содержимое и закрыть файл.

Открытие файлов

В Python можно открыть файл для чтения или записи с помощью функции `open()`. Ее первым аргументом является имя открываемого файла. Если файл находится в той же папке, что и программа, достаточно указать только имя файла, например `'thetimemachine.txt'`. В этом случае команда для открытия файла *thetimemachine.txt* будет выглядеть так:

```
fileObj = open('thetimemachine.txt')
```

Файловый объект сохраняется в переменной `fileObj`, которая будет использоваться в операциях чтения/записи.

Можно также указать *абсолютный путь* к файлу, включающий имя папки, в которой находится данный файл, и имена всех родительских папок. Например, `'C:\\Users\\Al\\frankenstein.txt'` (Windows) и `'/Users/Al/frankenstein.txt'` (macOS и Linux) — примеры абсолютных путей. Не забывайте о том, что в Windows символу обратной косой черты (`\`) должен предшествовать такой же экранирующий символ.

Если, к примеру, вы хотите открыть файл *frankenstein.txt*, то в качестве первого аргумента функции `open()` необходимо указать путь к файлу в виде строки (формат строки зависит от операционной системы).

```
fileObj = open('C:\\Users\\Al\\frankenstein.txt')
```

Файловый объект поддерживает несколько методов, предназначенных для записи, чтения и закрытия файла.

Запись и закрытие файлов

В программе шифрования нам нужно будет записывать зашифрованное (или дешифрованное) содержимое в новый файл, а для этого потребуется использовать метод `write()`.

Предварительно следует открыть файл в режиме записи, передав функции `open()` строку `'w'` в качестве второго аргумента (этот аргумент является необязательным, т.к. по умолчанию файл открывается для чтения). Например, введите в интерактивной оболочке следующую инструкцию:

```
>>> fileObj = open('spam.txt', 'w')
```

Эта команда создает файл *spam.txt* и открывает его в режиме записи, тем самым разрешая редактировать его содержимое. Если файл с таким именем уже существует в той папке, в которой функция `open()` создает новый файл, то он будет заменен, поэтому будьте внимательны, когда используете функцию `open()` в режиме записи.

Теперь, когда файл *spam.txt* открыт в режиме записи, можно выполнить запись, вызвав метод `write()`. У этого метода один аргумент — строка текста, подлежащего записи в файл. Запишите строку `'Hello, world!'` в файл *spam.txt*, введя в интерактивной оболочке следующую инструкцию:

```
>>> fileObj.write('Hello, world!')
13
```

В результате передачи строки `'Hello, world!'` методу `write()` она записывается в файл *spam.txt*, вслед за чем Python выводит на экран число `13` — количество символов, записанных в файл.

Если файл больше не нужен, сообщите Python о том, что работа с файлом завершена, вызвав метод `close()` файлового объекта:

```
>>> fileObj.close()
```

Поддерживается также режим присоединения, который напоминает режим записи, за исключением того, что новое содержимое не заменяет содержимое, уже имеющееся в файле, а добавляется в его конец. И хотя в нашей программе этот режим не используется, вам будет полезно знать,

что для его установки необходимо передать строку 'a' в качестве второго аргумента методу open().

Если при попытке вызова метода write() вы получаете сообщение об ошибке io.UnsupportedOperation: not readable, то это может означать, что файл не был открыт в режиме записи. Когда функция open() вызывается без указания необязательного второго параметра, она автоматически открывает файл в режиме чтения ('r'), что позволяет применять к данному файловому объекту только метод read().

Чтение из файла

Метод read() возвращает строку, содержащую весь текст, сохраненный в файле. Чтобы убедиться в этом, мы прочитаем файл *spam.txt*, который создали перед этим с помощью метода write(). Введите в интерактивной оболочке следующий код.

```
>>> fileObj = open('spam.txt', 'r')
>>> content = fileObj.read()
>>> print(content)
Hello world!
>>> fileObj.close()
```

В результате открытия файла создается файловый объект, который сохраняется в переменной fileObj. Далее можно прочитать содержимое файла с помощью метода read() и сохранить его в переменной content, которая затем выводится на экран. Когда работа с файловым объектом будет завершена, закройте его с помощью метода close().

Если вы получите сообщение об ошибке IOError: [Errno 2] No such file or directory, убедитесь в том, что файл действительно находится там, где вы предполагаете, и проверьте, что имена файла и папки указаны правильно. (Термины *папка* и *каталог* — синонимы.)

Мы будем использовать методы open(), read(), write() и close() для шифрования и дешифрования текстовых файлов в программе *transpositionFileCipher.py*.

Функция main()

Первая часть программы *transpositionFileCipher.py* должна быть вам знакома. В строке 4 импортируются программы *transpositionEncrypt.py* и *transpositionDecrypt.py*, а также модули Python time, os и sys. Далее следует определение функции main(), в которой устанавливаются переменные, используемые в программе.

```
 1. # Шифрование/дешифрование файлов с помощью перестановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import time, os, sys, transpositionEncrypt, transpositionDecrypt
 5.
 6. def main():
 7.     inputFilename = 'frankenstein.txt'
 8.     # БУДЬТЕ ВНИМАТЕЛЬНЫ: если файл с именем outputFilename
 9.     # уже существует, программа перезапишет его
10.     outputFilename = 'frankenstein.encrypted.txt'
11.     myKey = 10
12.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

Переменная `inputFilename` предназначена для хранения имени файла, подлежащего чтению, а зашифрованный (или дешифрованный) текст записывается в файл, имя которого хранится в переменной `outputFilename`. Целочисленное значение ключа, используемое в перестановочном шифре, хранится в переменной `myKey`. Программа ожидает, что в качестве значения переменной `myMode` будет указана строка `'encrypt'` или `'decrypt'`, задающая соответственно шифрование или дешифрование файла. Но прежде чем пытаться прочитать файл `inputFilename`, мы должны убедиться в том, что он существует, используя функцию `os.path.exists()`.

Проверка существования файла

Чтение файла — безопасная операция, тогда как в отношении записи в файл следует проявлять осторожность. В случае вызова функции `open()` в режиме записи для уже существующего файла его исходное содержимое будет заменено новым текстом. С помощью функции `os.path.exists()` можно проверить, существует ли указанный файл.

Функция `os.path.exists()`

Функция `os.path.exists()` имеет единственный строковый аргумент, задающий имя файла или путь к файлу, и возвращает значение `True`, если указанный файл существует, в противном случае возвращается значение `False`. Функция `os.path.exists()` находится в модуле `path`, который, в свою очередь, находится в модуле `os`, поэтому, когда мы импортируем модуль `os`, одновременно с ним импортируется также модуль `path`.

Введите в интерактивной оболочке следующие инструкции.

```
>>> import os
❶ >>> os.path.exists('spam.txt')
False
>>> os.path.exists('C:\\Windows\\System32\\calc.exe')  # Windows
True
>>> os.path.exists('/usr/local/bin/idle3')  # macOS
False
>>> os.path.exists('/usr/bin/idle3')  # Linux
False
```

В этом примере функция `os.path.exists()` подтверждает, что файл *calc.exe* существует в Windows. Разумеется, вы получите показанные результаты только в том случае, если Python запущен в Windows. Не забудьте экранировать символ обратной косой черты в обозначении пути к файлу в Windows, поместив перед ним еще один такой же символ. Если полный путь к файлу не задан (❶), Python будет искать его в текущем каталоге. В случае интерактивной оболочки IDLE это будет папка, в которой установлен Python.

Проверка существования файла с помощью функции `os.path.exists()`

Мы используем функцию `os.path.exists()` для того, чтобы проверить, существует ли файл, имя которого содержится в переменной `inputFilename`, иначе нам просто нечего будет шифровать или дешифровать. Это делается в строках 14–17.

```
14.     # Если входного файла не существует, программа завершается
15.     if not os.path.exists(inputFilename):
16.         print('The file %s does not exist. Quitting...' %
                (inputFilename))
17.         sys.exit()
```

Если указанного файла не существует, выводится сообщение для пользователя, и программа завершает свою работу.

Строковые методы, используемые для повышения гибкости пользовательского ввода

Далее программа проверяет, существует ли файл, имя которого содержится в переменной `outputFilename`, и, если это так, предлагает пользователю ввести `'C'` для продолжения или `'Q'` для выхода из программы.

Поскольку пользователь может ввести 'c', 'C' или даже 'Continue', мы хотим гарантировать, что программа воспримет любой из подобных вариантов ответа. Для этого нам потребуются дополнительные строковые методы.

Строковые методы upper(), lower() и title()

Строковые методы upper() и lower() возвращают строку, для которой они вызываются, соответственно в верхнем или нижнем регистре. Чтобы увидеть результаты их применения к одной и той же строке, введите в интерактивной оболочке следующие инструкции.

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.lower()
'hello'
```

Родственный метод title() возвращает строку, в которой первая буква каждого слова — прописная, а все остальные — строчные. Введите в интерактивной оболочке следующие инструкции.

```
>>> 'hello'.title()
'Hello'
>>> 'HELLO'.title()
'Hello'
>>> 'extra! extra! man bites shark!'.title()
'Extra! Extra! Man Bites Shark!'
```

Мы используем метод title() при форматировании сообщений, отображаемых на экране.

Строковые методы startswith() и endswith()

Метод startswith() возвращает значение True, если его строковый аргумент располагается в начале строки. Введите в интерактивной оболочке следующие инструкции.

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.startswith('H')
False
>>> spam = 'Albert'
❶ >>> spam.startswith('Al')
True
```

Строковый метод `startswith()` чувствителен к регистру символов и поддерживает аргументы, содержащие более одного символа (❶).

Строковый метод `endswith()` используется для проверки того, что одно строковое значение содержится в конце другого. Введите в интерактивной оболочке следующие инструкции.

```
>>> 'Hello world!'.endswith('world!')
True
❷ >>> 'Hello world!'.endswith('world')
False
```

Строковые значения должны в точности совпадать. Обратите внимание на то, что отсутствие восклицательного знака в строке `'world'` (❷) приводит к тому, что метод `endswith()` возвращает значение `False`.

Использование строковых методов в программе

Как уже подчеркивалось, мы хотим, чтобы программа принимала любой ввод, начинающийся с буквы `'C'`, независимо от ее регистра. То есть мы хотим, чтобы содержимое файла заменялось, если пользователь вводит `'c'`, `'continue'`, `'C'` или любую другую строку, начинающуюся с буквы `'C'`. Для повышения гибкости программы мы используем методы `lower()` и `startswith()`.

```
19.     # Если выходной файл существует, задать пользователю вопрос
20.     if os.path.exists(outputFilename):
21.         print('This will overwrite the file %s. (C)ontinue or
                  (Q)uit?' %(outputFilename))
22.         response = input('> ')
23.         if not response.lower().startswith('c'):
24.             sys.exit()
```

В строке 23 мы получаем первую букву строки и с помощью метода `startswith()` проверяем, является ли она буквой `'C'`. Этот метод чувствителен к регистру символов и проверяет совпадение с буквой `'c'` в нижнем регистре, поэтому мы используем метод `lower()` для изменения регистра строки. Если пользователь введет ответ, не начинающийся с буквы `'C'`, то метод `startswith()` вернет значение `False`, в результате чего условие в инструкции `if` станет равно `True` (благодаря оператору `not`), и вызов функции `sys.exit()` приведет к завершению работы программы. Фактически пользователь даже не обязан вводить `'Q'` для выхода из программы: любая строка, не начинающаяся с буквы `'C'`, приведет к вызову функции `sys.exit()` и выходу из программы.

Чтение входного файла

В строке 27 мы начинаем применять методы файлового объекта, которые обсуждались в начале главы.

```
26.     # Прочитать сообщение из входного файла
27.     fileObj = open(inputFilename)
28.     content = fileObj.read()
29.     fileObj.close()
30.
31.     print('%sing...' % (myMode.title()))
```

В строках 27–29 файл, имя которого хранится в переменной `inputFilename`, открывается, его содержимое читается в переменную `content`, а затем файл закрывается. Когда содержимое файла прочитано, в строке 31 выводится сообщение, извещающее пользователя о начале процесса шифрования или дешифрования. Поскольку переменная `myMode` содержит строку `'encrypt'` или `'decrypt'`, вызов строкового метода `title()` делает первую букву этой строки прописной. Сама строка вставляется в выражение `'%sing...'`, которое будет отображаться либо как `'Encrypting...'`, либо как `'Decrypting...'`.

Измерение затрат времени на шифрование и дешифрование

Шифрование или дешифрование целого файла может занять намного больше времени по сравнению с короткой строкой. Пользователю полезно знать, как долго длится этот процесс. Для измерения длительности операций можно использовать функции модуля `time`.

Модуль `time` и функция `time.time()`

Функция `time.time()` возвращает текущее время в виде вещественного числа, которое выражает количество секунд, прошедших с полуночи 1 января 1970 года. Этот момент называют *эпохой Unix*. Чтобы увидеть, как работает данная функция, введите в интерактивной оболочке следующие инструкции.

```
>>> import time
>>> time.time()
1540944000.7197928
>>> time.time()
1540944003.4817972
```

Поскольку функция `time.time()` возвращает вещественное значение, оно может выводиться с точностью до миллисекунды (1/1000 секунды). Разумеется, числа, отображаемые функцией `time.time()`, зависят от того, в какой момент времени она вызывается, а их непосредственная интерпретация несколько затруднительна. С первого взгляда вовсе не очевидно, что значению `1540944000.7197928` соответствует примерное время 15:00, приходящееся на вторник, 30 октября 2018 года. В то же время функция `time.time()` очень удобна для определения количества секунд, прошедших между двумя ее вызовами. Мы используем эту функцию для определения того, как долго выполняется программа.

Например, если вычесть приведенные в предыдущем листинге вещественные значения, то можно узнать, сколько времени ушло на ввод второй инструкции в интерактивной оболочке.

```
>>> 1540944003.4817972 - 1540944000.7197928
2.7620043754577637
```

Если вам нужно написать код, работающий со значениями даты и времени, ознакомьтесь с описанием модуля `datetime`, доступным по следующему адресу:

`https://automatetheboringstuff.com/chapter15/`

Использование функции `time.time()` в программе

В строке 34 функция `time.time()` возвращает текущее время, которое сохраняется в переменной `startTime`. В строках 35–38 вызывается функция `encryptMessage()` или `decryptMessage()`, в зависимости от того, какая строка сохранена в переменной `myMode`: `'encrypt'` или `'decrypt'`.

```
33.     # Измерить, как долго длится шифрование/дешифрование
34.     startTime = time.time()
35.     if myMode == 'encrypt':
36.         translated = transpositionEncrypt.encryptMessage(myKey, content)
37.     elif myMode == 'decrypt':
38.         translated = transpositionDecrypt.decryptMessage(myKey, content)
39.     totalTime = round(time.time() - startTime, 2)
40.     print('%sion time: %s seconds' % (myMode.title(), totalTime))
```

В строке 39 функция `time.time()` вызывается вновь, и время, сохраненное в переменной `startTime`, вычитается из текущего времени. Результат представляет количество секунд, прошедших между двумя вызовами функции `time.time()`. Выражение `time.time() - startTime` передается функции `round()`, которая округляет его до двух десятичных знаков после

запятой, поскольку нашей программе не нужна миллисекундная точность. Полученный результат сохраняется в переменной `totalTime`. В строке 40 выводится режим работы программы и сообщается количество времени, затраченное программой на шифрование или дешифрование файла.

Запись в выходной файл

На данном этапе зашифрованное (или дешифрованное) содержимое файла хранится в переменной `translated`. Однако по завершении работы программы эта переменная будет утеряна, поэтому необходимо сохранить содержащуюся в ней строку в файле. В строках 43–45 открывается новый файл (функции `open()` передается аргумент `'w'`), а затем вызывается метод `write()` файлового объекта.

```
42.     # Записать транслированное сообщение в выходной файл
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
```

В строках 47 и 48 выводятся завершающие сообщения и указывается имя файла, в котором был сохранен результирующий текст.

```
47.     print('Done %sing %s (%s characters).' % (myMode, inputFilename,
              len(content)))
48.     print('%sed file is %s.' % (myMode.title(), outputFilename))
```

Строка 48 — последняя в функции `main()`.

Вызов функции `main()`

В строках 53 и 54 (которые выполняются после инструкции `def`) вызывается функция `main()`, если программа выполняется, а не импортируется в виде модуля.

```
51. # Если программа transpositionCipherFile.py выполняется
52. # (а не импортируется как модуль), вызвать функцию main()
53. if __name__ == '__main__':
54.     main()
```

Об этом подробно рассказывалось в главе 7.

Резюме

В программе *transpositionFileCipher.py* использовались методы `open()`, `read()`, `write()` и `close()`, которые позволили нам шифровать большие текстовые файлы, хранящиеся на жестком диске. Вы узнали о том, как проверить существование файла с помощью функции `os.path.exists()`. Самое главное, что у нас появилась возможность расширить сферу применения наших программ, импортируя их функции в другие программы.

Вы также изучили ряд полезных строковых методов, упрощающих ввод данных пользователем, и узнали о том, как использовать модуль `time` для измерения быстродействия программы.

В отличие от шифра Цезаря, количество возможных ключей в случае перестановочного шифра слишком велико для того, чтобы можно было организовать атаку методом грубой силы. Но если написать программу, распознающую текст на естественном языке (в отличие от строк, содержащих бессмысленный текст), то компьютер сможет исследовать результат тысяч попыток дешифрования и определить, какой ключ позволяет успешно расшифровать сообщение. Об этом мы поговорим в главе 11.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Как правильно: `os.exists()` или `os.path.exists()`?
2. Когда началась эпоха Unix?
3. Каким будет результат вычисления следующих выражений?

```
'Foobar'.startswith('Foo')
'Foo'.startswith('Foobar')
'Foobar'.startswith('foo')
'bar'.endswith('Foobar')
'Foobar'.endswith('bar')
'The quick brown fox jumped over the yellow lazy dog.'.title()
```

11

ПРОГРАММНОЕ РАСПОЗНАВАНИЕ АНГЛИЙСКИХ СЛОВ

"Дед произносит что-то еще более длинное и сложное. И тут Уотерхауз (вот она – стихия криптоаналитика, когда нужно найти смысл среди кажущейся бессвязности, а нейронная сеть в голове чутко улавливает избыточность сигнала) понимает, что дед говорит по-английски с сильным местным акцентом".

Нил Стивенсон, "Криптономикон"

В предыдущей главе мы использовали перестановочный шифр для шифрования/дешифрования файлов, но пока не пробовали взломать его методом грубой силы. Сообщения, зашифрованные с помощью такого шифра, могут иметь тысячи возможных ключей. Компьютер, конечно, справится с ними методом грубой силы, но нам придется просматривать тысячи вариантов дешифровки, чтобы найти среди них исходный текст. Понятно, что это достаточно затруднительно, но решение все же существует.

Если компьютер дешифрует сообщение, используя неверный ключ, то результирующий текст представляет собой "мусор", а не текст на английском языке. Можно написать такую программу, чтобы она распознавала слова естественного языка. Тогда, если программа обнаружит, что был использован неверный ключ, она сможет автоматически перейти к следующему ключу. Как только будет найден ключ, результат применения которого содержит слова на английском языке, программа сообщит об этом

пользователю, что избавит вас от необходимости просматривать тысячи некорректно дешифрованных сообщений.

В этой главе...

- Словарный тип данных
- Метод `split()`
- Значение `None`
- Ошибки деления на нуль
- Функции `float()`, `int()` и `str()` и целочисленное деление
- Списковый метод `append()`
- Аргументы по умолчанию
- Расчет процентов

Может ли компьютер понимать английский язык?

Компьютер не способен понимать английский язык, во всяком случае, он понимает его не так, как мы, люди. Компьютеры не разбираются в математике, шахматах или политике, так же, как часы не знают, когда у нас обед. Компьютеры всего лишь выполняют инструкции одну за другой. А вот сами эти инструкции способны имитировать сложное поведение, позволяющее решать математические задачи, выигрывать шахматные партии и предсказывать результаты выборов.

В идеале нам нужна функция (назовем ее `isEnglish()`), которой можно передать строку и получить в ответ `True`, если строка представляет собой текст на английском языке, или `False`, если строка содержит текстовый "мусор". Рассмотрим в качестве примера предложение на английском языке и "мусорный" текст и попытаемся выявить в них какие-то закономерности.

```
Robots are your friends. Except for RX-686. She will try to eat you.
ai-pey  e. xrx ne augur iirl6 Rtiyt fhubE6d hrSei t8..ow eo.telyoosEs   t
```

Обратите внимание на то, что английский текст состоит из слов, которые можно найти в словаре, чего нельзя сказать о "мусоре". Поскольку слова обычно разделяются пробелами, один из способов проверки может заключаться в том, чтобы разбить сообщение по пробелам на строки и проверить, встречаются ли эти строки в словаре. Для разбивки строк на подстроки предназначен строковый метод `split()`, который способен определять границы слов по пробелам. Далее можно сравнить каждую под-

строку с каждым словом словаря, используя длинную инструкцию if, как показано ниже.

```
if word == 'aardvark' or word == 'abacus' or word == 'abandon' or
word == 'abandoned' or word ==' abbreviate' or word == 'abbreviation' or
word == 'abdomen' or ...
```

Подобное решение, конечно, допустимо, но навряд ли мы будем так делать, поскольку это слишком утомительно. К счастью, можно воспользоваться *файлом английского словаря*, который представляет собой текстовый файл, содержащий почти все слова английского языка. Этот файл имеется в материалах книги, так что нам остается только написать функцию isEnglish(), проверяющую вхождение подстрок сообщения в словарь.

Учитывайте, что это не официальный орфографический словарь. В нем содержатся далеко не все слова. Кроме того, в сообщении могут использоваться разного рода обозначения, такие как *RX-686*. Также исходный текст может быть составлен на другом языке, но мы ограничимся только английским.

Таким образом, функция isEnglish() не является абсолютно надежной, но если *большинство* слов в строковом аргументе — английские, то существует высокая вероятность того, что он представляет собой текст на английском языке. Вероятность дешифровать шифротекст неверным ключом и при этом получить текст на английском языке очень невелика.

Файл словаря (содержащий более 45 000 слов) доступен для загрузки на сайте издательства (см. введение). В нем слова, записанные прописными буквами, располагаются по одному в строке. Открыв его, вы увидите следующее.

```
AARHUS
AARON
ABABA
ABACK
ABAFT
ABANDON
ABANDONED
ABANDONING
ABANDONMENT
ABANDONS
--опущено--
```

Наша функция isEnglish() будет разбивать дешифрованную строку на подстроки и проверять, встречается ли каждая из них в файле словаря. Если определенное число подстрок являются английскими словами, мы

будем идентифицировать сообщение как текст на английском языке. А раз так, то существует большая вероятность того, что мы успешно дешифровали шифротекст с корректным ключом.

Исходный код модуля `Detect English`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите приведенный ниже код и сохраните его в файле *detectEnglish.py*. Убедитесь, что файл *dictionary.txt* находится в том же самом каталоге, иначе программа не будет работать. Запустите программу, нажав клавишу <F5>.

detectEnglish.py

```
 1. # Модуль распознавания английского языка
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. # Чтобы использовать данный модуль, введите следующий код:
 5. #   import detectEnglish
 6. #   detectEnglish.isEnglish(someString)  # возвращает True или False
 7. # В каталоге программы должен существовать файл dictionary.txt,
 8. # содержащий по одному слову в строке. Файл доступен для загрузки
 9. # на сайте издательства (см. введение).
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower() + ' \t\n'
12.
13. def loadDictionary():
14.     dictionaryFile = open('dictionary.txt')
15.     englishWords = {}
16.     for word in dictionaryFile.read().split('\n'):
17.         englishWords[word] = None
18.     dictionaryFile.close()
19.     return englishWords
20.
21. ENGLISH_WORDS = loadDictionary()
22.
23.
24. def getEnglishCount(message):
25.     message = message.upper()
26.     message = removeNonLetters(message)
27.     possibleWords = message.split()
28.
29.     if possibleWords == []:
30.         return 0.0  # слова отсутствуют, поэтому возвращаем 0.0
31.
32.     matches = 0
33.     for word in possibleWords:
```

```
34.         if word in ENGLISH_WORDS:
35.             matches += 1
36.     return float(matches) / len(possibleWords)
37.
38.
39. def removeNonLetters(message):
40.     lettersOnly = []
41.     for symbol in message:
42.         if symbol in LETTERS_AND_SPACE:
43.             lettersOnly.append(symbol)
44.     return ''.join(lettersOnly)
45.
46.
47. def isEnglish(message, wordPercentage=20, letterPercentage=85):
48.     # По умолчанию 20% слов должны быть в файле словаря,
49.     # а 85% символов сообщения должны быть буквами или
50.     # пробелами (а не знаками препинания или числами)
51.     wordsMatch = getEnglishCount(message) * 100 >= wordPercentage
52.     numLetters = len(removeNonLetters(message))
53.     messageLettersPercentage = float(numLetters) / len(message) * 100
54.     lettersMatch = messageLettersPercentage >= letterPercentage
55.     return wordsMatch and lettersMatch
```

Применение модуля `Detect English`

Программа *detectEnglish.py* не будет выполняться сама по себе. Вместо этого другие программы шифрования будут импортировать ее, чтобы можно было вызвать функцию `detectEnglish.isEnglish()`, которая возвращает `True`, если строка определяется как текст на английском языке. Вот почему в файле *detectEnglish.py* нет функции `main()`. Другие функции, включенные в файл *detectEnglish.py*, являются вспомогательными и вызываются функцией `isEnglish()`. То, что мы сделаем в этой главе, позволит любой программе импортировать модуль `detectEnglish` с помощью инструкции `import` и использовать содержащиеся в нем функции.

Вы также сможете использовать этот модуль в интерактивной оболочке, для того чтобы проверить, представляет ли собой отдельная строка текст на английском языке.

```
>>> import detectEnglish
>>> detectEnglish.isEnglish('Is this sentence English text?')
True
```

В этом примере функция определила, что строка `'Is this sentence English text?'` действительно представляет текст на английском языке, и поэтому вернула значение `True`.

Указания по использованию модуля и установка констант

Обратимся к первой части программы *detectEnglish.py*.

```
 1. # Модуль распознавания английского языка
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. # Чтобы использовать данный модуль, введите следующий код:
 5. #   import detectEnglish
 6. #   detectEnglish.isEnglish(someString)  # возвращает True или False
 7. # В каталоге программы должен существовать файл dictionary.txt,
 8. # содержащий по одному слову в строке. Файл доступен для загрузки
 9. # на сайте издательства (см. введение).
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower() + ' \t\n'
```

Первые девять строк кода — комментарии с описанием того, как использовать данный модуль. Они напоминают, что модуль будет работать только в том случае, если файл *dictionary.txt* находится в том же каталоге, что и файл *detectEnglish.py*.

В строках 10 и 11 устанавливается несколько констант, имена которых записаны в верхнем регистре. Как вам уже известно из главы 5, константы — это переменные, значения которых не изменяются после того, как были установлены. `UPPERLETTERS` — это константа, содержащая 26 букв алфавита в верхнем регистре. Она создана для удобства. С ее помощью устанавливается константа `LETTERS_AND_SPACE`, которая содержит все буквы алфавита в верхнем и нижнем регистре, а также символы пробела, табуляции и новой строки. Вместо того чтобы вводить все символы в верхнем и нижнем регистре, мы конкатенируем строку `UPPERLETTERS` с самой собой, вызывая строковый метод `UPPERLETTERS.lower()`. Символы табуляции и новой строки представляются экранированными символами `\t` и `\n`.

Словарный тип данных

Прежде чем продолжить рассмотрение остальной части программы *detectEnglish.py*, необходимо познакомиться со словарным типом данных, чтобы понять, как преобразовать текст, содержащийся в файле, в строку. *Словарный тип данных* (не следует путать с файлом словаря) может хранить множество значений, как и список. Но в списках мы извлекаем элементы с помощью целочисленных индексов, например `spam[42]`. А в словаре доступ к каждому элементу осуществляется по ключу. Если индексами списка

могут служить только целые числа, то ключом словаря может быть как целое число, так и строка, например `spam['hello']` или `spam[42]`. Словари позволяют организовать более гибкое хранение данных и не хранить элементы строго по порядку. Вместо квадратных скобок, используемых в списках, словари создаются с помощью фигурных скобок. Пустой словарь обозначается как `{}`.

Примечание

Имейте в виду, что файлы словарей и словарные значения – совершенно разные понятия, хоть и называются похожим образом. Словарь Python может содержать множество значений, а файл словаря – это текстовый файл, содержащий английские слова.

Элементы словаря представляются парами "ключ: значение", в которых ключи и значения разделены двоеточиями. Пары "ключ: значение" разделяются запятыми. Чтобы извлечь значение из словаря, необходимо указать ключ в квадратных скобках, как при индексации списка. Введите в интерактивной оболочке следующие инструкции.

```
>>> spam = {'key1': 'This is a value', 'key2': 42}
>>> spam['key1']
'This is a value'
```

Сначала мы создаем словарь `spam` с двумя парами "ключ: значение", а затем обращаемся к значению, связанному со строковым ключом `'key1'`. В словарях, как и в списках, можно хранить любые типы данных.

Обратите внимание на то, что, как и в случае списков, в переменных хранятся не сами словари, а ссылки на них. В следующем примере используются две переменные, содержащие ссылки на один и тот же словарь.

```
>>> spam = {'hello': 42}
>>> eggs = spam
>>> eggs['hello'] = 99
>>> eggs
{'hello': 99}
>>> spam
{'hello': 99}
```

В первой строке создается словарь `spam`, содержащий только одну пару "ключ: значение". В нем хранится целочисленное значение `42`, ассоциированное со строковым ключом `'hello'`. Во второй строке словарь присваивается другой переменной, `eggs`. После этого можно использовать переменную `eggs` для того, чтобы изменить значение в

словаре, ассоциированное со строковым ключом 'hello', сделав его равным 99. Теперь обе переменные, eggs и spam, должны возвращать одну и ту же словарную пару с обновленным значением.

Различие между словарями и списками

Между словарями и списками немало общего, но существуют и важные различия.

- Элементы словаря не упорядочены. Не существует первого или последнего элемента словаря, как это имеет место в списках.
- Словари нельзя конкатенировать с помощью оператора +. Если необходимо добавить новый элемент, используйте индекс с новым ключом, например foo['a newkey'] = 'a string'.
- В списках могут использоваться только целочисленные индексы, значения которых изменяются от нуля до длины списка минус один, тогда как в словарях можно использовать любой ключ. Если словарь хранится в переменной spam, то можно сохранить значение в элементе [3] даже в отсутствие элементов spam[0], spam[1] и spam[2].

Добавление и изменение элементов словаря

Можно добавлять и изменять значения в словаре, используя ключ словаря в качестве индекса. Чтобы увидеть, как это работает, введите в интерактивной оболочке следующие инструкции.

```
>>> spam = {42: 'hello'}
>>> print(spam[42])
hello
>>> spam[42] = 'goodbye'
>>> print(spam[42])
goodbye
```

В этом примере в словаре хранится строковое значение 'hello', связанное с ключом 42. Мы можем присвоить новое строковое значение 'goodbye' этому же ключу, используя инструкцию spam[42] = 'goodbye'. Присваивание нового значения существующему ключу словаря приводит к замене исходного значения, ассоциированного с этим ключом. Например, если мы обратимся к словарю с помощью ключа 42, то получим связанное с ним новое значение.

И точно так же, как списки могут содержать другие списки, словари тоже могут содержать другие словари (или списки). Чтобы проверить это, введите в интерактивной оболочке следующие инструкции.

```
>>> foo = {'fizz': {'name': 'Al', 'age': 144}, 'moo': ['a', 'brown',
'cow']}
>>> foo['fizz']
{'age': 144, 'name': 'Al'}
>>> foo['fizz']['name']
'Al'
>>> foo['moo']
['a', 'brown', 'cow']
>>> foo['moo'][1]
'brown'
```

Здесь создается словарь `foo`, содержащий два ключа: `'fizz'` и `'moo'`. Ключу `'fizz'` соответствует другой словарь, а ключу `'moo'` — список. (Не забывайте о том, что значения в словарях не хранятся в определенном порядке. Именно по этой причине пары "ключ: значение" при вводе **`foo['fizz']`** отображаются не в том порядке, в каком вы их вводили.) Чтобы извлечь значение из словаря, вложенного в другой словарь, сначала необходимо указать в квадратных скобках ключ большего набора данных, т.е. ключ `'fizz'` в данном примере. Затем следует вновь использовать квадратные скобки и ввести ключ `'name'`, соответствующий вложенному строковому элементу `'Al'`, который нужно извлечь.

Применение функции `len()` к словарям

Функция `len()` позволяет определить количество элементов в списке или количество символов в строке. Она же позволяет узнать количество элементов в словаре. Введите в интерактивной оболочке следующие инструкции.

```
>>> spam = {}
>>> len(spam)
0
>>> spam['name'] = 'Al'
>>> spam['pet'] = 'Zophie the cat'
>>> spam['age'] = 89
>>> len(spam)
3
```

В первой строке создается пустой словарь `spam`. Функция `len()` корректно сообщает длину этого словаря, равную нулю. Но после того как мы включаем в словарь три значения, `'Al'`, `'Zophie the cat'` и `89`, она возвращает `3`, соответствующее трем парам "ключ: значение", которые были только что занесены в словарь.

Применение оператора `in` к словарям

С помощью оператора `in` можно проверить существование определенного ключа в словаре. Важно помнить о том, что оператор `in` проверяет ключи, а не значения. Введите в интерактивной оболочке следующие инструкции.

```
>>> eggs = {'foo': 'milk', 'bar': 'bread'}
>>> 'foo' in eggs
True
>>> 'milk' in eggs
False
>>> 'blah blah blah' in eggs
False
>>> 'blah blah blah' not in eggs
True
```

Мы создаем словарь `eggs` с несколькими парами "ключ: значение", а затем с помощью оператора `in` проверяем, какие ключи существуют в словаре. Строка `'foo'` является ключом в словаре `eggs`, поэтому возвращается `True`. В двух последующих случаях мы получаем `False`, поскольку строка `'milk'` является значением, а не ключом, а строки `'blah blah blah'` вообще нет в словаре. Оператор `not in` тоже поддерживается, что подтверждает последняя команда.

Поиск элементов в словарях выполняется быстрее, чем в списках

Представьте, что вы создали в интерактивной оболочке следующие список и словарь.

```
>>> listVal = ['spam', 'eggs', 'bacon']
>>> dictionaryVal = {'spam': 0, 'eggs': 0, 'bacon': 0}
```

Python может найти строку `'bacon'` в словаре `dictionaryVal` немного быстрее, чем в списке `listVal`. Так происходит потому, что в случае списка Python должен начинать поиск с самого начала, поочередно просматривая элементы до тех пор, пока не найдет нужный. Если список очень большой, придется просмотреть множество элементов, что отнимет немало времени.

Но словарь, называемый также *хеш-таблицей*, сразу же сообщает, где именно в компьютерной памяти хранится значение для пары "ключ: значение", чем и объясняется тот факт, что элементы словаря не упорядочены. Каким бы ни был размер словаря, для нахождения любого элемента требуется одно и то же количество времени.

При поиске в коротких списках и словарях различие в скорости едва заметно. Однако в нашем модуле `detectEnglish` количество элементов будет исчисляться десятками тысяч, и выражение `word in ENGLISH_WORDS` будет многократно вычисляться при вызовах функции `isEnglish()`. В случае большого количества элементов использование словаря существенно ускоряет работу программы.

Использование циклов `for` со словарями

Цикл `for` может применяться для итерирования по ключам словаря аналогично тому, как это происходит в списках. Введите в интерактивной оболочке следующий код.

```
>>> spam = {'name': 'Al', 'age': 99}
>>> for k in spam:
...     print(k, spam[k])
...
Age 99
name Al
```

Цикл начинается с ключевого слова `for`. Далее задается переменная цикла `k`, а с помощью ключевого слова `in` мы указываем, что хотим итерировать по ключам словаря `spam`. Инструкция завершается двоеточием, за которым следует тело цикла. Функция `print(k, spam[k])` отображает текущий ключ словаря вместе с соответствующим значением.

Реализация файла словаря

Вернемся к программе *detectEnglish.py* и настроим файл словаря. Файл словаря хранится на жестком диске, и нам необходимо загрузить его в строковом виде. С этой целью мы создадим вспомогательную функцию `loadDictionary()`.

```
13. def loadDictionary():
14.     dictionaryFile = open('dictionary.txt')
15.     englishWords = {}
```

Прежде всего мы получаем объект файла словаря, вызывая функцию `open()` и передавая ей строку `'dictionary.txt'` в качестве имени файла. Затем создается переменная `englishWords` в виде пустого словаря.

Слова, хранящиеся в файле английского словаря, будут скопированы в словарь Python. Для хранения строк можно было бы использовать список,

но мы выбрали именно словарь, поскольку оператор `in` работает со словарями быстрее, чем со списками.

Сейчас мы познакомимся со строковым методом `split()`, который служит для разбивки файла словаря на отдельные подстроки.

Метод `split()`

Строковый метод `split()` разбивает переданную ему строку по пробелам, возвращая список из нескольких строк. Чтобы увидеть, как это работает, введите в интерактивной оболочке следующую инструкцию.

```
>>> 'My very energetic mother    just served us Nutella.'.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'Nutella.']
```

Результатом будет список, состоящий из восьми строк, — по одной строке для каждого из слов в исходной строке. Пробелы, даже если их несколько, опускаются. Методу `split()` можно передать необязательный аргумент, позволяющий задать непробельную строку, по которой следует осуществлять разбивку. Введите в интерактивной оболочке следующую инструкцию.

```
>>> 'helloXXXworldXXXhowXXXareXXyou?'.split('XXX')
['hello', 'world', 'how', 'areXXyou?']
```

Обратите внимание на то, что данная строка вообще не содержит пробелов. Вызов `split('XXX')` разбивает исходную строку на подстроки в тех местах, где встречается группа символов `'XXX'`, в результате чего мы получаем список из четырех строк. Последняя часть строки, `'areXXyou?'`, не разбивается, поскольку `'XX'` — не то же самое, что `'XXX'`.

Разбивка файла словаря на отдельные слова

Вернемся к файлу *detectEnglish.py* и рассмотрим, как осуществляется разбивка строки, хранящейся в файле словаря, с сохранением каждого слова в виде ключа.

```
16.     for word in dictionaryFile.read().split('\n'):
17.         englishWords[word] = None
```

Проанализируем строку 16. Переменная `dictionaryFile` хранит объект открытого файла. Метод `dictionaryFile.read()` осуществляет чтение всего файла и возвращает его в виде одной длинной строки. Затем мы вызываем метод `split()` для разбивки этой длинной строки по символам

новой строки. Поскольку в файле словаря слова содержатся по одному в строке, в результате разбивки по символам новой строки возвращается список, состоящий из всех слов словаря.

Создаваемый в строке 16 цикл `for` проходит по словам и сохраняет каждое из них в ключе словаря. А поскольку значения, ассоциируемые с ключами (мы используем словарный тип данных) нас не интересуют, мы сохраняем для каждого ключа значение `None`.

`None` — это служебное значение типа `NoneType`, которое присваивается переменной для того, чтобы показать отсутствие реального значения. В то время как булев тип данных имеет только два значения (`True` и `False`), тип `NoneType` имеет всего одно значение: `None`. Оно всегда записывается без кавычек и начинается с прописной буквы.

Предположим, например, что у вас имеется переменная `quizAnswer`, предназначенная для хранения ответов на вопросы викторины, которые пользователь дает в форме "да — нет". Если пользователь пропускает вопрос и не отвечает на него, то имеет смысл назначить переменной `quizAnswer` значение `None` в качестве значения по умолчанию, а не значение `True` или `False`. В противном случае может сложиться впечатление, что пользователь ответил на вопрос правильно или неправильно. Аналогичным образом при выходе из функции по достижении ее конца, а не в результате выполнения инструкции `return`, автоматически возвращается значение `None`, поскольку сама функция ничего не возвращает.

В строке 17 переменная `word` служит текущим ключом в словаре `englishWord`, а значением ключа становится `None`.

Возврат данных в виде словаря

По завершении цикла `for` словарь `englishWords` будет содержать десятки тысяч слов. На данном этапе мы закрываем объект файла, поскольку его содержимое уже прочитано, и возвращаем словарь `englishWords`.

```
18.     dictionaryFile.close()
19.     return englishWords
```

После этого вызывается функция `loadDictionary()` и возвращаемый ею словарь сохраняется в переменной `ENGLISH_WORDS`.

```
21. ENGLISH_WORDS = loadDictionary()
```

Функция `loadDictionary()` вызывается еще до того, как начнет выполняться остальной код модуля `detectEnglish`, но Python должен выполнить инструкцию `def loadDictionary()` до того, как мы сможем ее

вызвать. Именно по этой причине инструкция присваивания располагается после определения функции `loadDictionary()`.

Подсчет количества английских слов в сообщении

В строках 24–27 программы создается функция `getEnglishCount()`, которая имеет строковый аргумент и возвращает вещественное значение, определяющее долю распознанных английских слов по отношению к общему количеству слов в сообщении. Мы представим это отношение в виде значения в интервале от 0,0 до 1,0. Значению 0,0 соответствует отсутствие английских слов в сообщении, тогда как значение 1,0 означает, что все слова в сообщении — английские. На основании этого значения функция `isEnglish()` будет возвращать значение `True` или `False`.

```
24. def getEnglishCount(message):
25.     message = message.upper()
26.     message = removeNonLetters(message)
27.     possibleWords = message.split()
```

В функции сначала создается список слов, содержащихся в строке сообщения. В строке 25 символы строки преобразуются в верхний регистр. В строке 26 с помощью функции `removeNonLetters()` из строки удаляются все небуквенные символы, такие как числа и знаки препинания (о том, как работает эта функция, вы узнаете далее). Наконец, в строке 27 метод `split()` разбивает строку на слова и сохраняет их в переменной `possibleWords`.

Например, если функции `getEnglishCount()` была передана строка `'Hello there. How are you?'`, то после выполнения строк 25–27 список `possibleWords` будет содержать `['HELLO', 'THERE', 'HOW', 'ARE', 'YOU']`.

Если в строке сообщения содержатся цифры, например `'12345'`, то функция `removeNonLetters()` вернет пустую строку, и метод `split()` сформирует пустой список. В данной программе пустой список эквивалентен отсутствию слов на английском языке, т.е. случаю, когда их количество равно нулю, что может повлечь за собой ошибку деления на нуль.

Ошибка деления на нуль

Чтобы вернуть вещественное значение в интервале от 0,0 до 1,0, мы делим количество слов в переменной `possibleWords`, распознанных как английские, на общее количество слов, сохраненных в этой переменной. Несмотря на простоту данной операции, следует убедиться в том, что

список, хранящийся в переменной possibleWords, не является пустым. Если он пуст, то общее количество слов в списке possibleWords равно нулю.

Поскольку в арифметике операция деления на нуль не имеет смысла, попытка деления на нуль в Python приводит к возникновению ошибки. Чтобы проверить это, введите в интерактивной оболочке следующую инструкцию.

```
>>> 42 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    42 / 0
ZeroDivisionError: division by zero
```

Как видите, при делении 42 на 0 возникает ошибка ZeroDivisionError и выводится сообщение о том, что произошло. Во избежание проблем следует убедиться в том, что список possibleWords не является пустым.

В строке 29 проверяется, является ли список possibleWords пустым, а в строке 30 возвращается значение 0.0, если список не содержит ни одного слова.

```
29.     if possibleWords == []:
30.         return 0.0  # слова отсутствуют, поэтому возвращаем 0.0
```

Эта проверка необходима для того, чтобы избежать деления на нуль.

Считаем английские слова

Чтобы получить отношение количества английских слов к общему числу слов, мы будем делить количество слов в переменной possibleWords, распознанных как английские, на общее число слов, хранящихся в этой переменной. Таким образом, нужно подсчитать, сколько в сообщении английских слов. В строке 32 для счетчика matches устанавливается нулевое значение. В строке 33 организуется цикл for, в котором мы проходим по всем словам в списке possibleWords и проверяем существование каждого из них в словаре ENGLISH_WORDS. Если слово содержится в словаре, значение счетчика matches инкрементируется в строке 35.

```
32.     matches = 0
33.     for word in possibleWords:
34.         if word in ENGLISH_WORDS:
35.             matches += 1
```

По завершении цикла `for` найденное количество английских слов сохраняется в переменной `matches`. Не забывайте о том, что качество работы модуля `detectEnglish` зависит от точности и полноты файла словаря. Если какое-то слово отсутствует в файле словаря, то оно не будет учитываться как английское, даже если является таковым. И наоборот, если в словаре допущена орфографическая ошибка, то слова, не являющиеся английскими, могут быть учтены как таковые.

На данном этапе количество слов в списке `possibleWords`, распознанных как английские, и общее количество слов, хранящихся в этом списке, представлены целыми числами. Чтобы получить вещественное число в интервале от 0,0 до 1,0 при делении этих двух чисел, мы должны сделать одно из них вещественным.

Функции `float()`, `int()` и `str()` и целочисленное деление

Рассмотрим, как превратить целое число в вещественное, поскольку для нахождения искомого отношения мы должны разделить одно число на другое, а они оба являются целыми числами. В Python 3 деление всегда выполняется одним и тем же способом независимо от типов значений, тогда как в Python 2 в тех случаях, когда оба значения являются целыми, выполняется целочисленное деление. Поскольку пользователи могут импортировать модуль *detectEnglish.py*, пользуясь версией Python 2, мы должны передать хотя бы одно из чисел функции `float()`, чтобы получить результат деления в виде вещественного числа. Это будет гарантировать выполнение обычного деления независимо от используемой версии Python. Так обеспечивается *обратная совместимость* с предыдущими версиями.

Рассмотрим ряд других функций преобразования, хоть они и не будут использоваться в программе. Функция `int()` возвращает целочисленную версию своего аргумента, а функция `str()` преобразует аргумент строку. Чтобы увидеть, как они работают, введите в интерактивной оболочке следующие инструкции.

```
>>> float(42)
42.0
>>> int(42.0)
42
>>> int(42.7)
42
>>> int('42')
42
>>> str(42)
'42'
>>> str(42.7)
'42.7'
```

Как видите, функция `float()` превращает целое число `42` в вещественное. Функция `int()` превращает вещественные числа `42.0` и `42.7` в целые, отбрасывая их дробные части, и может превратить строку `'42'` в целое число. Функция `str()` преобразует числовые значения в строковые. Эти функции полезны, если нужно получить значение, эквивалентное другому типу данных.

Нахождение доли английских слов в сообщении

Чтобы найти отношение количества английских слов к общему количеству слов, мы делим полученное значение счетчика `matches` на общее количество слов, хранящихся в переменной `possibleWords`. В строке 36 для деления этих двух чисел используется оператор `/`.

```
36.     return float(matches) / len(possibleWords)
```

Когда мы передаем функции `float()` целочисленное значение счетчика `matches`, она возвращает вещественный эквивалент этого числа, который делится на длину списка `possibleWords`.

Инструкция `return float(matches) / len(possibleWords)` может стать причиной возникновения ошибки деления на нуль в том случае, когда вызов `len(possibleWords)` возвращает нуль. В свою очередь, это возможно только тогда, когда список `possibleWords` оказывается пустым. Однако в строках 29 и 30 этот случай специально проверяется, и если список пуст, то возвращается значение `0.0`. Если переменная `possibleWords` пустая, то выполнение программы никогда не идет далее строки 30, поэтому мы можем быть уверены в том, что строка 36 не станет причиной возникновения ошибки `ZeroDivisionError`.

Удаление небуквенных символов

Некоторые символы, например числа или знаки препинания, будут препятствовать нормальной работе нашей программы, поскольку слова не будут выглядеть в точности так, как они сохранены в файле словаря. Например, если последним словом в сообщении является `'you.'` и мы не удалим точку в конце слова, то оно не будет воспринято как английское, поскольку `'you'` записано без точки в файле словаря. Чтобы избежать подобного рода случаев неправильной интерпретации, числа и знаки препинания необходимо удалять.

Чтобы удалить из строки любые числа и знаки препинания, функция `getEnglishCount()`, которую мы только что рассмотрели, вызывает для нее функцию `removeNonLetters()`.

```
39. def removeNonLetters(message):
40.     lettersOnly = []
41.     for symbol in message:
42.         if symbol in LETTERS_AND_SPACE:
43.             lettersOnly.append(symbol)
```

В строке 40 создается пустой список, а в строке 41 начинается цикл `for`, в котором перебираются все символы в аргументе `message`. В цикле проверяется, встречается ли текущий символ в строке `LETTERS_AND_SPACE`. Если символ является числом или знаком препинания, значит, он отсутствует в строке `LETTERS_AND_SPACE` и не будет добавлен в список. Если же символ встречается в этой строке, то он добавляется в список с помощью метода `append()`, который мы сейчас рассмотрим.

Метод append() списка

Когда мы добавляем значение в конец списка, мы говорим, что оно *присоединяется* к списку. Эта операция встречается в Python настолько часто, что для нее предусмотрен метод `append()`, который имеет единственный аргумент, присоединяемый к концу списка. Введите в интерактивной оболочке следующие инструкции.

```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
```

Создав пустой список `eggs`, мы можем вызвать метод `eggs.append('hovercraft')`, чтобы добавить в список строку `'hovercraft'`. В результате возвращается единственное значение, сохраненное в этом списке, т.е. `'hovercraft'`. Если мы вновь используем метод `append()` для добавления строки `'eels'` в конец списка, то возвращается строка `'hovercraft'`, за которой следует строка `'eels'`. Точно так же метод `append()` можно использовать для добавления элементов в список `lettersOnly`, который мы создали ранее. Именно это и делает вызов `lettersOnly.append(symbol)` в строке 43 цикла `for`.

Создание строки, объединяющей буквы

По завершении цикла `for` список `lettersOnly` будет содержать список всех буквенных и пробельных символов из оригинальной строки `message`. Поскольку список, состоящий из односимвольных строк, не очень полезен для обнаружения английских слов, в строке 44 строки символов, хранящиеся в списке `lettersOnly`, объединяются в одну возвращаемую строку.

```
44.     return ''.join(lettersOnly)
```

Чтобы конкатенировать список элементов `lettersOnly` в одну большую строку, мы вызываем строковый метод `join()` для пустой строки `''`. В результате будет получена объединенная строка без разделителей. После этого результирующее строковое значение возвращается функцией `removeNonLetters()`.

Распознавание английских слов

Попытки дешифровать сообщение с использованием неверного ключа будут часто приводить к получению намного большего количества небуквенных и непробельных символов, чем в типичном сообщении на английском языке. Кроме того, получаемые слова часто будут представлять собой случайные сочетания символов, отсутствующие в словаре английских слов. Оба этих момента проверяются в функции `isEnglish()`.

```
47. def isEnglish(message, wordPercentage=20, letterPercentage=85):
48.     # По умолчанию 20% слов должны быть в файле словаря,
49.     # а 85% символов сообщения должны быть буквами или
50.     # пробелами (а не знаками препинания или числами)
```

В строке 47 создается функция `isEnglish()`, которая получает строковый аргумент и возвращает `True`, если строка представляет собой текст на английском языке, и `False` в противном случае. Функция имеет три параметра: `message`, `wordPercentage=20` и `letterPercentage=85`. Первый параметр содержит строку, подлежащую проверке, тогда как второй и третий параметры задают значения по умолчанию для процентных долей слов и букв, которые должна содержать строка, чтобы считаться текстом на английском языке. (*Процент* — это число в интервале от 0 до 100, которое характеризует долю определенных элементов по отношению к общему количеству элементов.) Использование аргументов по умолчанию и расчет процентов обсуждаются в следующих разделах.

Использование аргументов по умолчанию

Иногда оказывается, что функции почти всегда передается одно и то же значение. Вместо того чтобы указывать это значение при каждом вызове, можно задать аргумент по умолчанию в инструкции `def`.

В строке 47 определение функции включает три параметра, причем для параметров `wordPercentage` и `letterPercentage` заданы значения по умолчанию: `20` и `85` соответственно. Функция `isEnglish()` может быть вызвана с указанием от одного до трех аргументов. Если параметры `wordPercentage` и `letterPercentage` не предоставлены, то значениями этих параметров будут их аргументы по умолчанию.

Аргументы по умолчанию определяют, какую процентную долю в строке `message` должны составлять реальные английские слова, чтобы функция `isEnglish()` определила данное сообщение как состоящее из английских слов, и какую процентную долю должны составлять буквы и пробелы, а не числа и знаки препинания. Например, если функция `isEnglish()` вызывается только с одним аргументом, то будут использованы значения по умолчанию для параметров `wordPercentage` (целое число `20`) и `letterPercentage` (целое число `85`), т.е. 20% строк должны быть представлены английскими словами и 85% строк — буквами и пробелами. Эти значения подходят для распознавания текстов на английском языке в большинстве случаев, но, возможно, вы захотите испытать и другие комбинации аргументов, когда функция `isEnglish()` нуждается в использовании более слабых или более строгих пороговых значений. В подобных ситуациях нужно лишь передать значения для параметров `wordPercentage` и `letterPercentage`, а не использовать значения по умолчанию. В табл. 11.1 представлены различные варианты вызова функции `isEnglish()` и их эквиваленты.

Таблица 11.1. Вызов функции с указанием и без указания аргументов по умолчанию

Вызов функции	Эквивалент
`isEnglish('Hello')`	`isEnglish('Hello', 20, 85)`
`isEnglish('Hello', 50)`	`isEnglish('Hello', 50, 85)`
`isEnglish('Hello', 50, 60)`	`isEnglish('Hello', 50, 60)`
`isEnglish('Hello', letterPercentage=60)`	`isEnglish('Hello', 20, 60)`

Как показывает третий из приведенных в таблице примеров, если функция вызывается с указанием второго и третьего параметров, то программа будет использовать эти значения, а не аргументы, заданные по умолчанию.

Вычисление процентной доли

Как только программе становятся известны процентные доли, которые следует использовать в качестве критериев, она должна вычислить реальные процентные доли для строки `message`. Например, строка `'Hello cat MOOSE fsdkl ewpin'` содержит пять "слов", но только три из них являются реальными английскими словами. Чтобы вычислить процентную долю английских слов в этой строке, мы делим их количество на общее количество слов и умножаем результат на 100. Процентная доля английских слов в данной строке равна 3 / 5 * 100, что составляет 60%. Другие примеры вычисления процентных долей приведены в табл. 11.2.

Таблица 11.2. Вычисление процентной доли английских слов

Количество английских слов	Общее количество слов	Доля английских слов	* 100	=	Процентная доля
3	5	0,6	* 100	=	60
6	10	0,6	* 100	=	60
300	500	0,6	* 100	=	60
32	87	0,3678	* 100	=	36,78
87	87	1,0	* 100	=	100
0	10	0	* 100	=	0

Значение процентной доли всегда будет заключено в пределах от 0 (отсутствие английских слов) до 100 (все слова — английские). Наша функция `isEnglish()` будет рассматривать строку как состоящую из английских слов, если по крайней мере 20% слов включены в словарь и 85% символов являются буквами или пробелами. Это означает, что сообщение все равно будет считаться текстом, состоящим из английских слов, даже если файл словаря не совершенен или некоторые слова в сообщении имеют другое написание, чем то, которое зафиксировано в словаре.

В строке 51 процентная доля распознанных английских слов вычисляется путем передачи аргумента `message` функции `getEnglishCount()`, которая возвращает вещественное значение в интервале от `0.0` до `1.0`.

```
51.     wordsMatch = getEnglishCount(message) * 100 >= wordPercentage
```

Процентная доля получается умножением этого вещественного числа на `100`. Если результат равен или превышает значение параметра `wordPercentage`, то в переменной `wordsMatch` сохраняется значение

True. (Вспомните, что оператор сравнения >= вычисляет булево выражение.) В противном случае переменная wordsMatch будет равна False.

Процентная доля буквенных символов в строке message вычисляется в строках 52–54 путем деления количества буквенных символов на общее количество символов в сообщении.

```
52.     numLetters = len(removeNonLetters(message))
53.     messageLettersPercentage = float(numLetters) / len(message) * 100
54.     lettersMatch = messageLettersPercentage >= letterPercentage
```

Ранее в программе мы создали функцию removeNonLetters(), предназначенную для нахождения всех буквенных и пробельных символов в строке, поэтому нам достаточно вызвать ее. В строке 52 функция removeNonLetters(message) возвращает строку, содержащую только буквенные и пробельные символы из аргумента message. Передача этой строки функции len() позволяет получить общее количество буквенных и пробельных символов в строке message, которое мы сохраняем в виде целого числа в переменной numLetters.

Процентная доля букв определяется в строке 53 путем деления вещественной версии целого числа, сохраненного в переменной numLetters, на результат вызова len(message). Значение, возвращаемое функцией len(message), будет равно общему количеству символов в строке message. Как уже обсуждалось, функция float() нужна для того, чтобы гарантировать, что в строке 53 выполняется обычное, а не целочисленное деление, если модуль detectEnglish подключается к программе, которая выполняется под управлением Python 2.

В строке 54 проверяется, превышает ли процентная доля, хранящаяся в переменной messageLettersPercentage, значение параметра letterPercentage. Результатом вычисления этого выражения будет булево значение, которое сохраняется в переменной lettersMatch.

Мы хотим, чтобы функция isEnglish() возвращала значение True только в том случае, если обе переменные, wordsMatch и lettersMatch, равны True. В строке 55 эти значения комбинируются с помощью оператора and.

```
55.     return wordsMatch and lettersMatch
```

Если обе переменные равны True, то функция isEnglish() признает сообщение как текст на английском языке и возвращает True. В противном случае возвращается значение False.

Резюме

Программа шифрования/дешифрования файлов с помощью перестановочного шифра является улучшенным решением по сравнению с аналогичной программой для работы с шифром Цезаря, поскольку она может иметь сотни и даже тысячи возможных ключей, а не 26. Несмотря на то что для компьютера не составляет труда расшифровать сообщение с тысячами возможных ключей, нам нужно написать код, который определяет, является ли строка текстом на английском языке, что может свидетельствовать о правильном дешифровании сообщения.

В этой главе мы написали программу для распознавания английских слов на основе текстового файла словаря. Словарь полезен по той причине, что может содержать множество значений, как и список. Однако, в отличие от списка, в качестве индексов в словарях можно использовать строковые значения ключей, а не только целые числа. Большинство операций, которые можно выполнять в отношении списка, такие как определение длины с помощью функции `len()` или использование операторов `in` и `not in`, поддерживается также и для словарей. Однако в случае большого количества элементов оператор `in` выполняется для словаря гораздо быстрее, чем для списка. Для нас это особенно полезно, поскольку наш словарь содержит тысячи значений, которые нужно быстро просматривать.

В этой главе также были рассмотрены метод `split()`, позволяющий разбивать строку на список строк, и тип данных `NoneType`, имеющий только одно значение: `None`. Это значение позволяет показать факт отсутствия значения.

Вы узнали о том, как избежать ошибок деления на нуль при использовании оператора `/`, как преобразовать значение в другой тип данных с помощью функций `int()`, `float()` и `str()` и как использовать метод `append()` для добавления значения в конец списка.

Создавая определение функции, можно предоставить для некоторых параметров аргументы по умолчанию. Если при вызове функции этим параметрам не передаются аргументы, то программа использует значения аргументов по умолчанию.

В главе 12 вы узнаете о том, как применить распознавание английских слов для взлома перестановочного шифра.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Что выведет следующий код?

```
spam = {'name': 'Al'}
print(spam['name'])
```

2. Что выведет следующий код?

```
spam = {'eggs': 'bacon'}
print('bacon' in spam)
```

3. Напишите цикл `for` для вывода значений, содержащихся в следующем словаре.

```
spam = {'name': 'Zophie', 'species': 'cat', 'age': 8}
```

4. Что выведет следующий код?

```
print('Hello, world!'.split())
```

5. Что выведет следующий код?

```
def spam(eggs=42):
    print(eggs)
spam()
spam('Hello')
```

6. Каков процент слов английского языка в следующем предложении?

```
"Whether it's flobulllar in the mind to quarfalog the slings
and arrows of outrageous guuuuuuuuur."
```

12

ВЗЛОМ ПЕРЕСТАНОВОЧНОГО ШИФРА

"Рон Ривест, один из тех, кто придумал RSA, полагает, что ограничение криптографии окажется безрассудством. По его мнению, плохо без разбора запрещать технологии только потому, что некоторые преступники могут использовать их в своих целях".

Саймон Сингх, "Книга шифров"

В этой главе мы применим метод грубой силы для взлома перестановочного шифра. Из тысяч потенциальных ключей корректным может быть только тот, который приводит к получению понятного текста (в книге предполагается текст на английском языке). Используя модуль *detectEnglish.py* (см. главу 11), программа взлома перестановочного шифра поможет нам найти корректный ключ.

В этой главе...

- Многострочный текст с тройными кавычками
- Строковый метод `strip()`

Исходный код программы Transposition Hacker

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨New File. Введите в этом окне приведенный ниже код и сохраните его в файле *transpositionHacker.py*. Как всегда, убедитесь в том, что модули *pyperclip.py* и *transpositionDecrypt.py* (см. главу 8), а также модуль *detectEnglish.py* и файл *dictionary.txt* (см. главу 11) находятся в том же каталоге, что и файл *transpositionHacker.py*. Запустите программу, нажав клавишу <F5>.

transpositionHacker.py

```
# Программа взлома перестановочного шифра
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import pyperclip, detectEnglish, transpositionDecrypt

def main():
    # Приведенный ниже текст можно скопировать
    # из файла примера (см. введение)
    myMessage = """AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e  enlh
      na  indeit n uhoretrm au ieu v er Ne2 gmanw,forwnlbsya apor tE.no
      euarisfatt  e mealefedhsppmgAnlnoe(c -or)alat r lw o eb  nglom,Ain
      one dtes ilhetcdba. t tg eturmudg,tfl1e1 v  nitiaicynhrCsaemie-sp
      ncgHt nie cetrgmnoa yc r,ieaa  toesa- e a0m82e1w shcnth  ekh
      gaecnpeutaaieetgn iodhso d ro hAe snrsfcegrt NCsLc b17m8aEheideikfr
      aBercaeu thllnrshicwsg etriebruaisss  d iorr."""

    hackedMessage = hackTransposition(myMessage)

    if hackedMessage == None:
        print('Failed to hack encryption.')
    else:
        print('Copying hacked message to clipboard:')
        print(hackedMessage)
        pyperclip.copy(hackedMessage)


def hackTransposition(message):
    print('Hacking...')

    # Выполнение программы на Python можно в любой момент прервать,
    # нажав <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux)
    print('(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux)
           to quit at any time.)')

    # Перебор всех возможных ключей методом грубой силы
    for key in range(1, len(message)):
        print('Trying key #%s...' % (key))

```

```
32.         decryptedText = transpositionDecrypt.decryptMessage(key, message)
33.
34.         if detectEnglish.isEnglish(decryptedText):
35.             # Запросить у пользователя подтверждение корректности ключа
36.             print()
37.             print('Possible encryption hack:')
38.             print('Key %s: %s' % (key, decryptedText[:100]))
39.             print()
40.             print('Enter D if done, anything else to continue hacking:')
41.             response = input('> ')
42.
43.             if response.strip().upper().startswith('D'):
44.                 return decryptedText
45.
46.     return None
47.
48. if __name__ == '__main__':
49.     main()
```

Пример выполнения программы `Transposition Hacker`

Выполнив программу *transpositionHacker.py*, вы должны получить следующие результаты.

```
Hacking...
(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux) to quit at any time.)
Trying key #1...
Trying key #2...
Trying key #3...
Trying key #4...
Trying key #5...
Trying key #6...

Possible encryption hack:
Key 6: Augusta Ada King-Noel, Countess of Lovelace (10 December 1815 - 27
November 1852) was an English mat

Enter D if done, anything else to continue hacking:
> D
Copying hacked message to clipboard:
Augusta Ada King-Noel, Countess of Lovelace (10 December 1815 - 27 November
1852) was an English mathematician and writer, chiefly known for her work
on Charles Babbage's early mechanical general-purpose computer, the Analytical
Engine. Her notes on the engine include what is recognised as the first
algorithm intended to be carried out by a machine. As a result, she is often
regarded as the first computer programmer.
```

После проверки ключа #6 программа отображает отрывок дешифрованного сообщения, чтобы получить от пользователя подтверждение правильности найденного ключа. В данном примере расшифрованная версия выглядит многообещающей. Если пользователь подтверждает вводом **D**, что представленный отрывок дешифрован правильно, то программа возвращает полную версию сообщения. Данное сообщение представляет собой фрагмент биографии Ады Лавлейс (разработав в 1842–1843 годах алгоритм вычисления чисел Бернулли, она стала первым компьютерным программистом в истории человечества). Если расшифрованный вариант оказывается ложным, пользователь может нажать любую другую клавишу, и программа продолжит проверять другие ключи.

Вновь запустите программу, но на этот раз пропустите корректный ключ, нажав любую другую клавишу, кроме **D**. Программа предположит, что корректный ключ все еще не найден, и продолжит перебор всех возможных ключей методом грубой силы.

```
--опущено--
Trying key #417...
Trying key #418...
Trying key #419...
Failed to hack encryption.
```

В конечном счете программа переберет все возможные ключи и проинформирует пользователя о том, что взлом шифротекста оказался невозможным.

Давайте теперь проанализируем исходный код, чтобы понять, как работает программа.

Импорт модулей

Первые несколько строк кода информируют пользователя о возможностях программы. В строке 4 импортируются несколько модулей, которые были написаны нами или уже встречались в предыдущих главах: *pyperclip.py*, *detectEnglish.py* и *transpositionDecrypt.py*.

```
1. # Программа взлома перестановочного шифра
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, detectEnglish, transpositionDecrypt
```

Программа оказалась довольно короткой (менее 50 строк кода), потому что значительная ее часть реализована в других программах, которые мы используем в качестве модулей.

Создание многострочного текста с помощью тройных кавычек

Переменная myMessage содержит шифротекст, который мы пытаемся взломать. В строке 9 хранится строка, которая начинается и заканчивается тройными кавычками. Заметьте, насколько это длинная строка.

```
6. def main():
7.     # Приведенный ниже текст можно скопировать
8.     # из файла примера (см. введение)
9.     myMessage = """AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e  enlh
         na  indeit n uhoretrm au ieu v er Ne2 gmanw,forwnlbsya apor tE.no
         euarisfatt  e mealefedhsppmgAnlnoe(c -or)alat r lw o eb  nglom,Ain
         one dtes ilhetcdba. t tg eturmudg,tfl1e1 v  nitiaicynhrCsaemie-sp
         ncgHt nie cetrgmnoa yc r,ieaa  toesa- e a0m82e1w shcnth  ekh
         gaecnpeutaaieetgn iodhso d ro hAe snrsfcegrt NCsLc b17m8aEheideikfr
         aBercaeu  thllnrshicwsg etriebruaisss  d iorr."""
```

Текст, заключенный в тройные кавычки, называется *многострочным*, поскольку он охватывает несколько строк и может содержать символы разрыва строки. Многострочные тексты удобны по той причине, что их можно использовать для включения длинных строк в исходный код программы, а также тем, что они не требуют экранирования встречающихся в них одинарных и двойных кавычек. В качестве примера многострочного текста введите в интерактивной оболочке следующие инструкции.

```
>>> spam = """Dear Alice,
Why did you dress up my hamster in doll clothing?
I look at Mr. Fuzz and think, "I know this was Alice's doing."
Sincerely,
Brienne"""
>>> print(spam)
Dear Alice,
Why did you dress up my hamster in doll clothing?
I look at Mr. Fuzz and think, "I know this was Alice's doing."
Sincerely,
Brienne
```

Обратите внимание на то, что этот текст занимает несколько строк. Все, что следует за открывающими тройными кавычками, будет интерпретироваться как часть строки до тех пор, пока программа снова не встретит тройные кавычки. При создании многострочного текста допускается использование либо трех двойных, либо трех одинарных кавычек.

Отображение результатов взлома сообщения

Код взлома шифротекста помещен в функцию `hackTransposition()`, которая вызывается в строке 11 и будет определена в строке 21. Эта функция имеет один аргумент: зашифрованное сообщение, которое мы пытаемся взломать. Если функции удалось взломать шифротекст, она возвращает строку дешифрованного текста, в противном случае возвращается значение `None`.

```
11.     hackedMessage = hackTransposition(myMessage)
12.
13.     if hackedMessage == None:
14.         print('Failed to hack encryption.')
15.     else:
16.         print('Copying hacked message to clipboard:')
17.         print(hackedMessage)
18.         pyperclip.copy(hackedMessage)
```

В строке 11 вызывается функция `hackTransposition()`, которая возвращает взломанное сообщение, если попытка оказалась удачной, или значение `None`. Результат сохраняется в переменной `hackedMessage`.

В строках 13 и 14 программа проверяет значение переменной `hackedMessage` и, если она равна `None`, сообщает пользователю о том, что взломать шифротекст не удалось.

Следующие четыре строки определяют, что будет делать программа, если функции удалось взломать шифротекст. В строке 17 выводится дешифрованное сообщение, а в строке 18 оно копируется в буфер обмена. Но для того, чтобы этот код заработал, мы должны определить функцию `hackTransposition()`, которая рассматривается ниже.

Получение взломанного сообщения

Функция `hackTransposition()` начинается парой инструкций `print()`.

```
21. def hackTransposition(message):
22.     print('Hacking...')
23.
24.     # Выполнение программы на Python можно в любой момент прервать,
25.     # нажав <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux):
26.     print('(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux)
              to quit at any time.)')
```

Поскольку программа проверяет множество ключей, она выводит сообщение о том, что предпринимается попытка дешифровать сообщение.

Функция `print()` в строке 26 сообщает пользователю о том, что он может в любой момент выйти из программы, нажав комбинацию клавиш <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux). Данная комбинация позволяет завершить работу любой программы на языке Python.

В следующих строках определяется диапазон возможных ключей, которые программа должна перебрать, пытаясь взломать перестановочный шифр.

```
28.     # Перебор всех возможных ключей методом грубой силы
29.     for key in range(1, len(message)):
30.         print('Trying key #%s...' % (key))
```

Возможные ключи берутся из интервала от 1 до длины сообщения. Все они перебираются в цикле `for`, который начинается в строке 29. В строке 30 программа отображает текущий ключ.

В строке 32 вызывается функция `decryptMessage()` из созданной нами ранее программы *transpositionDecrypt.py*, которая получает дешифрованное сообщение, соответствующее текущему тестируемому ключу, и сохраняет его в переменной `decryptedText`.

```
32.         decryptedText = transpositionDecrypt.decryptMessage(key, message)
```

Дешифрованный текст, содержащийся в переменной `decryptedText`, будет представлять собой осмысленный текст на английском языке только в случае использования корректного ключа. В противном случае мы получим "мусор".

Далее строка, содержащаяся в переменной `decryptedText`, передается функции `detectEnglish.isEnglish()`, которую мы создали в главе 11. Если функция определяет текст как английский, то программа выводит фрагмент текста из переменной `decryptedText` и выдает запрос пользователю.

```
34.         if detectEnglish.isEnglish(decryptedText):
35.             # Запросить у пользователя подтверждение корректности ключа
36.             print()
37.             print('Possible encryption hack:')
38.             print('Key %s: %s' % (key, decryptedText[:100]))
39.             print()
40.             print('Enter D if done, anything else to continue hacking:')
41.             response = input('> ')
```

Тот факт, что функция `detectEnglish.isEnglish()` вернула значение `True`, еще не означает, что программе удалось найти корректный ключ. Это

может быть всего лишь ложноположительный результат в том смысле, что программа распознала в качества английского текст, который в действительности является "мусором". Чтобы пользователь мог это проверить, программа выводит в строке 38 фрагмент текста. Для вывода первых 100 символов строки, сохраненной в переменной `decryptedText`, используется срез `decryptedText[:100]`.

В строке 41 программа приостанавливает работу, ожидая, пока пользователь не введет букву **D** или что-нибудь еще. Полученная строка сохраняется в переменной `response`.

Строковый метод `strip()`

Когда пользователь не придерживается строго инструкций, выводимых для него программой, могут возникать ошибки. Если программа *transpositionHacker.py* предлагает пользователю ввести **D** в качестве подтверждения, то это означает, что никакой другой ответ не будет принят. Рассмотрим, как можно использовать строковый метод `strip()` для того, чтобы заставить программу принимать другие варианты ответа при условии, что они близки к `D`.

Метод `strip()` возвращает версию строки, из которой удалены любые пробельные символы в начале и конце. К пробельным относятся символы пробела, табуляции и новой строки. Чтобы увидеть, как это работает, введите в интерактивной оболочке следующие инструкции.

```
>>> '      Hello'.strip()
'Hello'
>>> 'Hello       '.strip()
'Hello'
>>> '     Hello World      '.strip()
'Hello World'
```

В этом примере метод `strip()` удаляет пробельные символы, находящиеся в начале и в конце первых двух строк. Если строка наподобие `'     Hello World      '` содержит пробельные символы и в начале, и в конце, то метод удалит их по обе стороны строки, но не удалит пробелы между другими символами.

Метод `strip()` может также иметь строковый аргумент, определяющий, какие символы, в отличие от пробельных, подлежат удалению из начала и конца строки. В качестве примера введите в интерактивной оболочке следующие инструкции.

```
>>> 'aaaaaHELLOaa'.strip('a')
'HELLO'
>>> 'ababaHELLObaba'.strip('ab')
'HELLO'
>>> 'abccabcbacbXYZabcXYZacccab'.strip('abc')
'XYZabcXYZ'
```

В первых двух случаях передача строковых аргументов `'a'` и `'ab'` приводит к удалению этих символов в начале и в конце строки. В то же время метод `strip()` не удаляет символы, находящиеся посередине строки. Как показано в третьем примере, строка `'abc'` остается в строке `'XYZabcXYZ'`.

Применение строкового метода `strip()`

Вернемся к исходному коду программы. В строке 43 с помощью инструкции `if` задается условие, позволяющее пользователю вводить разные варианты ответа.

```
43.             if response.strip().upper().startswith('D'):
44.                 return decryptedText
```

Если бы это условие было сформулировано в виде `response == 'D'`, то пользователь был бы строго обязан ввести именно **`D`** для завершения работы программы. Например, если пользователь введет `'d'`, `' D'` или `'Done'`, то условие не сработает, и программа продолжит проверку других ключей вместо того, чтобы вернуть взломанное сообщение.

Чтобы обойти эту проблему, из начала и конца строки, сохраненной в переменной `response`, с помощью метода `strip()` удаляются пробелы. Затем для строки, возвращаемой методом `response.strip()`, вызывается метод `upper()`. Независимо от того, введет ли пользователь `'d'` или `'D'`, строка, возвращаемая методом `upper()`, всегда будет начинаться с прописной буквы `'D'`. Это немного упрощает использование программы.

Чтобы пользователь мог ввести слово, которое начинается с буквы `'D'`, мы используем метод `startswith()`, позволяющий проверить только первую букву слова. Например, если пользователь введет `'  done'` в качестве ответа, то пробелы будут удалены, а результирующая строка `'done'` будет передана методу `upper()`. После того как метод `upper()` переведет все символы строки в верхний регистр, результирующая строка `'DONE'` будет передана методу `startswith()`, который вернет `True`, поскольку строка начинается с символа `'D'`.

Если пользователь подтверждает, что дешифрованная строка корректна, то функция `hackTransposition()` в строке 44 возвращает дешифрованный текст.

Невозможность взлома сообщения

Строка 46 выполняется по завершении цикла `for`, запущенного в строке 29:

```
46.     return None
```

Если программа достигла данной точки, значит, не была выполнена инструкция `return` в строке 44, а это может произойти только в том случае, если перебор всех ключей не позволил дешифровать сообщение. Тогда в строке 46 возвращается значение `None`, указывающее на невозможность взлома.

Вызов функции `main()`

В строках 48 и 49 вызывается функция `main()`, если программа была запущена на выполнение, а не импортирована в виде модуля другой программой, использующей функцию `hackTransposition()`.

```
48. if __name__ == '__main__':
49.     main()
```

Не забывайте о том, что значение переменной `__name__` устанавливается интерпретатором Python. Функция `main()` не будет вызвана, если программа *transpositionHacker.py* импортируется в виде модуля.

Резюме

Как и глава 6, эта глава оказалась короткой, потому что значительная часть рассмотренного в ней кода уже реализована в написанных ранее программах. Наша программа взлома может использовать функции из других программ, импортируя их в виде модулей.

Вы узнали о том, как применять тройные кавычки для включения текстовых блоков, простирающихся на несколько строк в исходном коде. Вы также узнали, насколько полезен может быть строковый метод `strip()` для удаления пробелов или других символов из начала или конца строки.

Наличие программы *detectEnglish.py* помогло нам сэкономить немало времени, которое пришлось бы потратить на самостоятельную проверку того,

содержит ли каждый дешифрованный вариант сообщения осмысленный текст на английском языке. В результате мы смогли применить метод грубой силы для взлома шифротекста путем перебора тысяч возможных ключей шифрования.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Какой результат вернет приведенное ниже выражение?

```
'   Hello world'.strip()
```

2. Какие символы являются пробельными?
3. Почему вызов `'Hello world'.strip('o')` возвращает строку, которая по-прежнему содержит букву `'o'`?
4. Почему вызов `'xxxHelloxxx'.strip('X')` возвращает строку, которая по-прежнему содержит букву `'x'`?

13

АФФИННОЕ ШИФРОВАНИЕ С ПОМОЩЬЮ МОДУЛЬНОЙ АРИФМЕТИКИ

"Люди веками пытались добиться приватности – шепотом, в темноте, за закрытыми дверями, с помощью тайных рукопожатий и посланников с запечатанными конвертами. Действуя по старинке, мы не могли надежно хранить свои тайны, а с помощью электронных технологий – можем".

Эрик Хьюз, "Манифест шифропанка" (1993)

В этой главе рассматриваются мультипликативный и аффинный шифры. Мультипликативный шифр напоминает шифр Цезаря, только в качестве операции шифрования используется не сложение, а умножение. Аффинный шифр представляет собой сочетание мультипликативного шифра и шифра Цезаря, что повышает его стойкость и надежность.

Но сначала вы узнаете о модульной арифметике и наибольшем общем делителе – двух математических концепциях, в которых следует разобраться для реализации аффинного шифра. На их основе мы создадим модуль для обработки "завертываний" и поиска подходящих ключей для аффинного шифра. Этот модуль будет задействован при разработке программы аффинного шифрования в главе 14.

В этой главе...

- Модульная арифметика
- Оператор деления с остатком (%)
- Наибольший общий делитель (НОД)
- Групповое присваивание
- Алгоритм Евклида для нахождения наибольшего общего делителя
- Мультипликативный и аффинный шифры
- Расширенный алгоритм Евклида для нахождения модульных обращений

Модульная арифметика

Модульная арифметика — раздел математики, в котором целые числа сравниваются по модулю некоторого натурального числа. Вычисляемые остатки "вращаются" вокруг этого числа, подобно стрелке часов. Мы будем использовать модульную арифметику для обработки "завертываний" в аффинном шифре. Рассмотрим, как это работает.

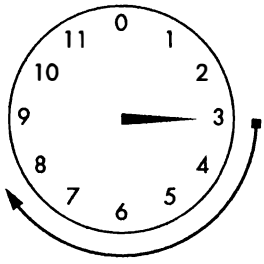


Рис. 13.1. *3 часа + 5 часов = 8 часов*

Представьте циферблат, снабженный только часовой стрелкой, на котором вместо 12 часов указано 0. (Если бы часы проектировались программистами, клянусь, так и было бы.) Если сейчас 3 часа, то что покажет стрелка спустя 5 часов? Ответить на этот вопрос не составит труда: 3 + 5 = 8, т.е. стрелка преодолеет 5 отметок и покажет 8 часов (рис. 13.1).

А если сейчас 10 часов, то что покажет стрелка через 5 часов? Сложив два числа, получим 5 + 10 = 15, но на часах нет такой отметки, там всего 12 часовых делений. Чтобы определить время, мы выполняем вычитание 15 – 12 = 3, что дает 3 часа. (Обычно мы различаем время до и после полудня, но для модульной арифметики это не имеет значения.)

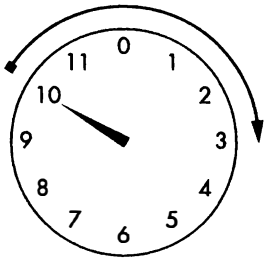


Рис. 13.2. *10 часов + 5 часов = 3 часа*

Чтобы проверить расчеты, вращайте часовую стрелку на 5 часов вперед, начиная с 10 часов. Она окажется на делении, которое соответствует 3 часам (рис. 13.2).

Если сейчас 10 часов, то какое время будет через 200 часов? Результат сложения 200 + 10 = 210, что намного больше 12. Поскольку один полный оборот

возвращает часовую стрелку в ее исходную позицию, задачу можно решить, последовательно вычитая 12 (что соответствует одному полному обороту) до тех пор, пока не получим число, меньшее 12. Выполнив вычитание 210 – 12, мы получаем число 198, которое все еще больше, чем 12, поэтому продолжаем вычитать 12, пока разность не станет меньше 12. В данном случае окончательным ответом будет 6. Таким образом, если сейчас 10 часов, то спустя 200 часов часовая стрелка покажет 6 часов (рис. 13.3).

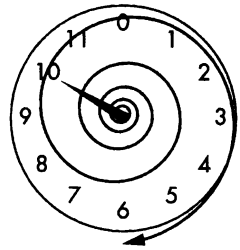


Рис. 13.3. *10 часов + 200 часов = 6 часов*

Если хотите проверить расчеты, вращайте часовую стрелку на 200 часов вперед, и вы убедитесь в том, что она остановится на отметке 6 часов. Но нам гораздо проще заставить компьютер выполнить соответствующие вычисления с помощью оператора деления с остатком.

Оператор деления с остатком

Для записи модульных выражений используют *оператор деления с остатком*[1] (modulo operator), сокращенно — *mod*. В Python оператор *mod* обозначается символом процента (`%`). Этот оператор возвращает остаток от деления целых чисел. Например, 21 ÷ 5 = 4 с остатком 1, и 21 % 5 = 1. Точно так же 15 % 12 равно 3, аналогично тому, как 15 часов — это также 3 часа. Чтобы увидеть, как работает оператор *mod*, введите в интерактивной оболочке следующие выражения.

```
>>> 21 % 5
1
>>> (10 + 200) % 12
6
>>> 10 % 10
0
>>> 20 % 10
0
```

Мы уже знаем, что если к времени 10 часов прибавить 200 часов, то стрелка на 12-часовом циферблате покажет 6 часов. Соответственно, результатом выражения (10 + 200) % 12 будет 6. Обратите внимание на то, что применение оператора *mod* к двум числам, которые делятся без остатка, дает 0, как, например, в случае операций 10 % 10 или 20 % 10.

[1] Также существует неформальный термин *деление по модулю*, однако его использование не рекомендуется (`https://ru.wikipedia.org/wiki/Деление_с_остатком`). — *Примеч. ред.*

Позже мы задействуем оператор *mod* для обработки "завертываний" в аффинном шифре. Он также применяется в алгоритме нахождения наибольшего общего делителя двух целых чисел, что позволяет искать допустимые ключи для аффинного шифра.

Нахождение множителей для вычисления наибольшего общего делителя

Множители — это числа, произведение которых равно заданному числу. Рассмотрим операцию $4 \cdot 6 = 24$. Здесь 4 и 6 — множители, на которые разбивается число 24. Поскольку представленное в таком виде число делится на любой из своих множителей нацело (без остатка), их также называют *делителями* данного числа.

Число 24 можно разбить на множители разными способами:

$$8 \cdot 3 = 24,$$
$$12 \cdot 2 = 24,$$
$$24 \cdot 1 = 24.$$

Поэтому множителями числа 24 являются числа 1, 2, 3, 4, 6, 8, 12 и 24.

Рассмотрим множители числа 30:

$$1 \cdot 30 = 30,$$
$$2 \cdot 15 = 30,$$
$$3 \cdot 10 = 30,$$
$$5 \cdot 6 = 30.$$

Таким образом, множителями числа 30 являются числа 1, 2, 3, 5, 6, 10, 15 и 30. Имейте в виду, что любое число всегда будет иметь в качестве множителей 1 и само себя, поскольку результат умножения 1 на число равен данному числу. Также обратите внимание на то, что списки множителей для чисел 24 и 30 содержат общие множители 1, 2, 3 и 6. Из этих множителей наибольшим является число 6, поэтому 6 называется *наибольшим общим множителем*, или, что встречается чаще, — *наибольшим общим делителем* (НОД), чисел 24 и 30.

Самый простой способ нахождения НОД двух чисел — визуализация их множителей. Для этого мы воспользуемся *счетными палочками Кюизенера* — брусочками, составленными из ряда квадратиков, количество которых равно представляемому числу. Счетные палочки облегчают наглядную демонстрацию арифметических операций. Использование счетных палочек Кюизенера для визуализации операций $3 + 2 = 5$ и $5 \cdot 3 = 15$ проиллюстрировано на рис. 13.4.

Длина составленных вместе палочек 3 и 2 равна длине палочки 5. С помощью счетных палочек можно находить даже результат умножения двух чисел, строя прямоугольник, сторонами которого являются палочки, соответствующие умножаемым числам. Количество квадратиков в таком прямоугольнике дает искомый результат.

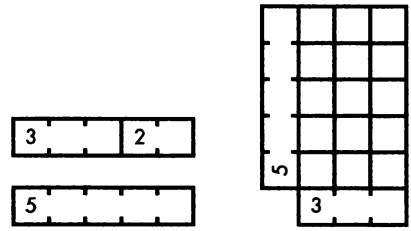


Рис. 13.4. *Представление операций сложения и умножения с помощью счетных палочек Кюизенера*

Пусть палочка длиной 20 квадратиков представляет число 20. Тогда число является делителем 20, если некоторое количество соответствующих ему палочек укладывается вдоль палочки 20 вровень с ней. Как показано на рис. 13.5, числа 4 и 10 являются делителями 20, поскольку путем составления соответствующих палочек в ряд можно получить палочку длиной 20 квадратиков.

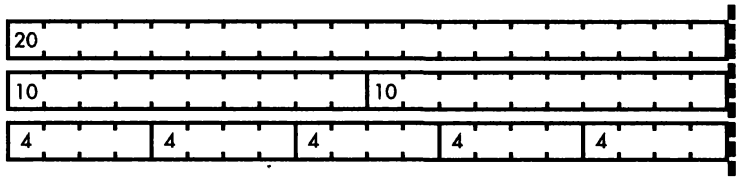


Рис. 13.5. *Использование счетных палочек Кюизенера для иллюстрации того факта, что числа 4 и 10 являются делителями числа 20*

В то же время числа 6 и 7 не являются делителями числа 20, поскольку укладыванием соответствующих палочек в ряд не удается добиться того, чтобы края результирующей палочки и палочки, состоящей из 20 квадратиков, совпадали (рис. 13.6).

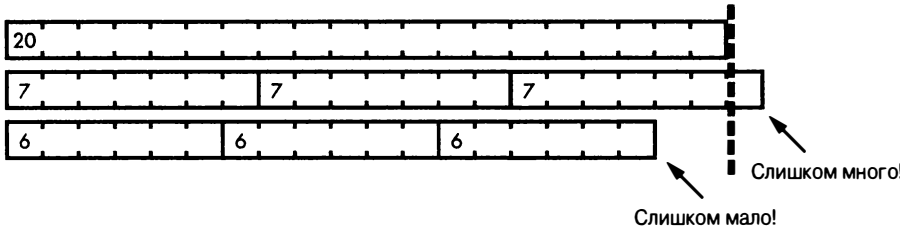


Рис. 13.6. *Использование счетных палочек Кюизенера для иллюстрации того факта, что числа 6 и 7 не являются делителями числа 20*

Наибольшим общим делителем (НОД) двух палочек, т.е. двух представляемых ими чисел, является самая длинная из палочек, составлением которых можно уравнять длины обоих рядов (рис. 13.7).

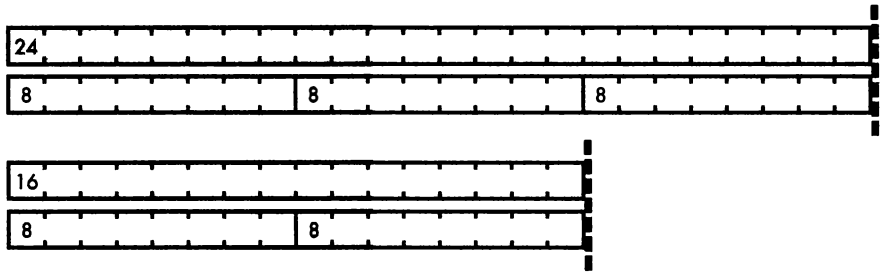


Рис. 13.7. *Использование счетных палочек Кюизенера для демонстрации НОД чисел 16 и 24*

В этом примере ряд из палочек длиной 8 квадратиков можно выложить вровень как с палочкой 24, так и с палочкой 16. Поэтому искомым НОД является число 8.

Теперь можно написать функцию для нахождения НОД двух чисел.

Групповое присваивание

Функция `gcd()`, которую мы собираемся написать, будет возвращать НОД двух чисел. Но сначала рассмотрим полезный трюк, который называется *групповым присваиванием*. С помощью этого трюка в одной инструкции присваивания можно задать значения сразу нескольким переменным. Введите в интерактивной оболочке следующие выражения.

```
>>> spam, eggs = 42, 'Hello'
>>> spam
42
>>> eggs
'Hello'
>>> a, b, c, d = ['Alice', 'Brienne', 'Carol', 'Danielle']
>>> a
'Alice'
>>> d
'Danielle'
```

Переменные слева и справа от оператора присваивания = разделяются запятыми. Количество элементов в списке должно совпадать с количеством переменных, указанных слева от оператора присваивания. В противном случае Python выдаст сообщение об ошибке, указывающее на то, что необходимо предоставить больше значений или же количество значений слишком велико.

Одно из основных применений группового присваивания — перестановка значений двух переменных. Введите в интерактивной оболочке следующие инструкции.

```
>>> spam = 'hello'
>>> eggs = 'goodbye'
>>> spam, eggs = eggs, spam
>>> spam
'goodbye'
>>> eggs
'hello'
```

Присвоив значение `'hello'` переменной `spam` и значение `'goodbye'` переменной `eggs`, мы затем меняем значения местами с помощью группового присваивания. Этот трюк с перестановкой значений будет применен для реализации алгоритма Евклида, с помощью которого ищется НОД.

Алгоритм Евклида для нахождения НОД

На первый взгляд, задача нахождения НОД кажется простой: для этого нужно определить все множители обоих чисел, а затем взять наибольший из них. Но в случае больших чисел все оказывается вовсе не таким легким.

Евклид, греческий математик, живший около 2000 лет тому назад, разработал короткий алгоритм нахождения НОД двух чисел с помощью модульной арифметики. Ниже приведен код функции `gcd()`, которая реализует этот алгоритм на языке Python и возвращает НОД двух чисел, `a` и `b`.

```
def gcd(a, b):
    while a != 0:
        a, b = b % a, a
    return b
```

Функция `gcd()` получает два числа, `a` и `b`, а затем выполняет групповое присваивание в цикле для нахождения НОД. Работа функции `gcd()` по нахождению НОД чисел 24 и 32 представлена на рис. 13.8.

Детальное рассмотрение того, как работает алгоритм Евклида, выходит за рамки книги, но можете быть уверены в том, что эта функция возвращает НОД двух чисел, которые ей передаются в качестве аргументов. Если вызвать ее в интерактивной оболочке, передав ей числа `24` и `32` в качестве параметров `a` и `b`, то она вернет число `8`.

```
>>> gcd(24, 32)
8
```

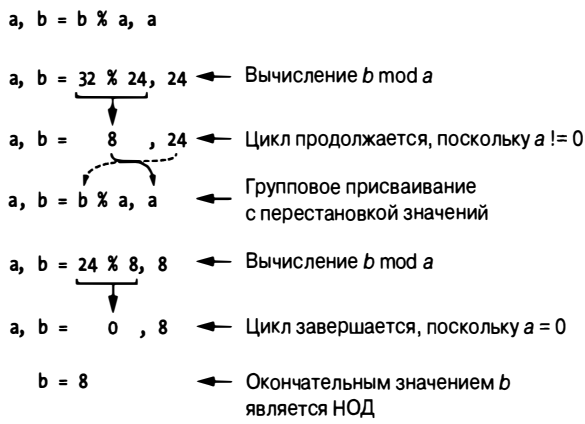


Рис. 13.8. *Схема работы функции* `gcd()`

Огромным достоинством функции `gcd()` является то, что она легко справляется с большими числами.

```
>>> gcd(409119243, 87780243)
6837
```

Функция `gcd()` пригодится нам при выборе допустимых ключей для мультипликативного и аффинного шифров, о чем мы поговорим далее.

Как работают мультипликативный и аффинный шифры

В шифре Цезаря шифрование и дешифрование символов реализуется путем их преобразования в числа, добавления или вычитания ключа и последующего обратного преобразования новых чисел в символы.

В случае шифрования с помощью *мультипликативного шифра* мы умножаем индекс на ключ. Например, если мы шифруем букву 'E' с помощью ключа 3, то нужно найти ее индекс (4) и умножить его на ключ (3), чтобы получить индекс зашифрованной буквы ($4 \cdot 3 = 12$), которому соответствует буква 'M'.

Если результат умножения превышает общее количество букв, то мультипликативный шифр сталкивается с той же проблемой "завертывания", что и шифр Цезаря, но теперь мы можем использовать для решения проблемы оператор *mod*. Например, в шифре Цезаря переменная `SYMBOLS` содержала строку `'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'`. Ниже приведена таблица, включающая несколько первых и последних символов строки `SYMBOLS` вместе с их индексами.

0	1	2	3	4	5	6		59	60	61	62	63	64	65
A	B	C	D	E	F	G		8	9	0		I	?	

Рассмотрим, во что превращаются символы в результате шифрования с помощью ключа 17. Чтобы зашифровать символ 'F' с помощью этого ключа, мы умножаем его индекс 5 на 17 и находим остаток от деления полученного результата на 66, чтобы учесть "завертывание" 66-символьного набора. Результат операции $(5 \cdot 17) \bmod 66$ равен 19, а числу 19 соответствует символ 'T'. Поэтому в результате шифрования буквы 'F' с помощью мультипликативного шифра и ключа 17 мы получаем букву 'T'. Ниже приведены две строки, представляющие все символы используемого алфавита и их зашифрованные версии. Символу в верхней строке соответствует символ в нижней строке, имеющий тот же индекс.

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'ARizCTk2EVm4GXo6IZq8Kbs0Mdu!Ofw.QhyBSj1DUl3FWn5HYp7Jar9Lct Nev?Pgx'
```

Сравните этот результат с тем, который мы получили бы с помощью шифра Цезаря, где символы шифруются путем простого сдвига:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'RSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.ABCDEFGHIJKLMNOPQ'
```

Как нетрудно заметить, мультипликативный шифр с ключом 17 обеспечивает более непредсказуемое распределение символов в зашифрованной строке, поэтому его труднее взломать. Однако выбор ключей для мультипликативного шифра требует внимательности, о чем мы поговорим далее.

Выбор допустимых мультипликативных ключей

Не всякое число годится на роль ключа мультипликативного шифра. Например, выбрав 11 в качестве ключа, вы получите следующий результат:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4'
```

Такой ключ нам не подходит, поскольку все символы 'A', 'G' и 'M' шифруются одной и той же буквой: 'A'. Встретив в зашифрованном тексте символ 'A', вы не будете знать, в какой символ он должен быть дешифрован. Аналогичная проблема возникает при шифровании букв 'S', 'Y', 'e', 'k' и др.

В мультипликативном шифре ключ и размер символьного набора должны быть взаимно простыми числами. Два числа называются *взаимно простыми*, если их НОД равен 1. Иными словами, они не должны иметь никаких других общих множителей, кроме 1. Например, числа `num1` и `num2`

взаимно просты, если `gcd(num1, num2) == 1`, где `num1` — ключ, а `num2` — размер символьного набора. Поскольку в предыдущем примере 11 (ключ) и 66 (размер символьного набора) имеют НОД, не равный 1, то они не являются взаимно простыми, откуда следует, что число 11 не может быть использовано в качестве ключа мультипликативного шифра. Отметим, что свойство взаимной простоты не требует, чтобы каждое из двух чисел было простым.

В случае использования мультипликативного шифра очень важно знать, как правильно применять модульную арифметику и функцию `gcd()`. С помощью функции `gcd()` мы выясняем, являются ли два числа взаимно простыми, так как это необходимо для определения того, пригодно ли число для использования в качестве ключа мультипликативного шифра.

Для набора, состоящего из 66 символов, мультипликативный шифр может иметь только 20 различных ключей, т.е. даже меньше, чем в случае шифра Цезаря! Но можно скомбинировать мультипликативный шифр с шифром Цезаря и получить более мощный аффинный шифр, к обсуждению которого мы сейчас перейдем.

Шифрование с помощью аффинного шифра

Одним из недостатков мультипликативного шифра является то, что буква 'A' всегда транслируется в букву 'A'. Причина заключается в том, что буква 'A' имеет нулевой индекс, а умножение нуля на любое число всегда дает нуль. От этой проблемы можно избавиться, добавив второй ключ для шифрования с помощью шифра Цезаря после того, как будет выполнено мультипликативное шифрование. Этот дополнительный шаг превращает мультипликативный шифр в аффинный.

Аффинный шифр имеет два ключа: A и B. Ключ A — это целое число, которое умножается на индекс буквы. К полученному произведению прибавляется ключ B, после чего вычисляется остаток от деления полученной суммы на 66, как это делается в оригинальном шифре Цезаря. Это означает, что аффинный ключ имеет в 66 раз больше возможных ключей, чем мультипликативный. Это также гарантирует, что буква 'A' не будет всегда транслироваться в саму себя в шифротексте.

Процесс дешифрования аффинного шифра — это зеркальное отражение процесса шифрования (рис. 13.9).

Для дешифрования аффинного шифра мы используем операции, обратные по отношению к использованным при шифровании. Рассмотрим процесс дешифрования и способ вычисления модульных обращений более подробно.

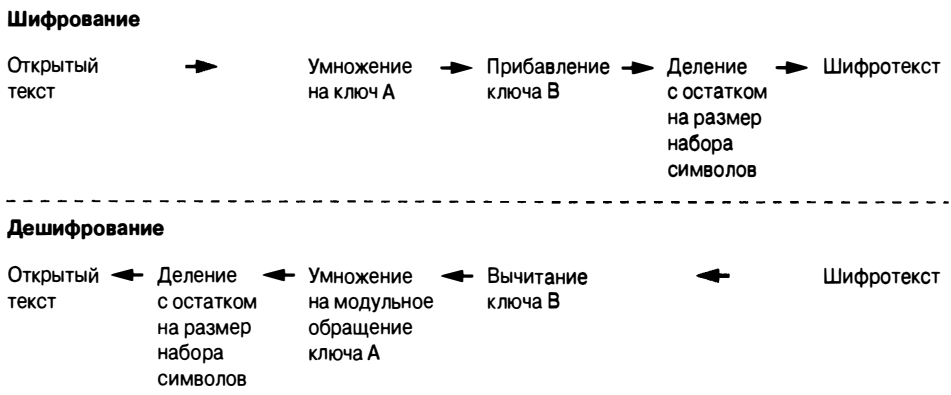


Рис. 13.9. *Процессы шифрования и дешифрования с использованием аффинного шифра*

Дешифрование с помощью аффинного шифра

В шифре Цезаря для шифрования используется операция сложения, а для дешифрования — операция вычитания. В аффинном шифре для шифрования используется операция умножения. Логично предположить, что для дешифрования аффинного шифра потребуется операция деления. Но если вы попытаетесь это сделать, то увидите, что такой подход не работает. Для дешифрования аффинного шифра необходимо выполнить умножение на модульное обращение ключа. Эта операция является обращением операции *mod*, которая применялась в процессе шифрования.

Модульное обращение (modular inverse) двух чисел представляется выражением $(a \cdot i) \% m = 1$, где i — модульное обращение, а a и m — два числа. Например, модульным обращением 5 mod 7 будет некоторое число i, такое, что $(5 \cdot i) \% 7 = 1$. Можно попытаться найти это число методом грубой силы, выполнив следующие вычисления:

1 не является модульным обращением 5 mod 7, поскольку $(5 \cdot 1) \% 7 = 5$;

2 не является модульным обращением 5 mod 7, поскольку $(5 \cdot 2) \% 7 = 3$;

3 является модульным обращением 5 mod 7, поскольку $(5 \cdot 3) \% 7 = 1$.

Несмотря на то что ключи шифрования и дешифрования для шифра Цезаря, используемого в аффинном шифре, совпадают, ключи шифрования и дешифрования для мультипликативного шифра — это два разных числа. В качестве ключа шифрования можно выбрать любое число, лишь бы оно было взаимно простым с размером символьного набора, который в данном случае равен 66. Если для шифрования с помощью аффинного шифра выбран ключ 53, то ключом дешифрования должно стать модульное обращение 53 mod 66.

1 не является модульным обращением 53 mod 66, поскольку (53 · 1) % 66 = 53.

2 не является модульным обращением 53 mod 66, поскольку (53 · 2) % 66 = 40.

3 не является модульным обращением 53 mod 66, поскольку (53 · 3) % 66 = 27.

4 не является модульным обращением 53 mod 66, поскольку (53 · 4) % 66 = 14.

5 является модульным обращением 53 mod 66, поскольку (53 · 5) % 66 = 1.

Поскольку модульным обращением для чисел 53 и 66 является число 5, именно его следует использовать в качестве ключа дешифрования для аффинного шифра. Чтобы расшифровать букву шифротекста, умножьте индекс этой буквы на 5, а затем найдите остаток от деления полученного результата на 66 с помощью оператора *mod*. Вы получите число, являющееся индексом незашифрованной буквы.

Используя 66-символьный набор, зашифруем слово "Cat" с помощью ключа 53. Букве 'C' соответствует индекс 2, а 2 · 53 = 106, что больше размера символьного набора, поэтому делим 106 на 66 и получаем остаток 40. Символом с индексом 40 в символьном наборе является 'o', таким образом, символ 'C' становится в шифротексте символом 'o'.

Выполним те же действия для следующей буквы, 'a'. В символьном наборе букве 'a' соответствует индекс 26, а 26 · 53 % 66 = 58, что соответствует индексу символа '7'. Поэтому символ 'a' в результате шифрования превращается в символ '7'. Букве 't' соответствует индекс 45, а 45 · 53 % 66 = 9, что дает индекс буквы 'J'. Таким образом, слову "Cat" в шифротексте соответствует слово "o7J".

В процессе дешифрования мы выполняем умножение на модульное обращение 53 % 66, т.е. 5. Символу 'o' соответствует индекс 40, а 40 · 5 % 66 = 2 (индекс буквы 'C'). Символу '7' соответствует индекс 58, а 58 · 5 % 66 = 26 (индекс буквы 'a'). Символу 'J' соответствует индекс 9, а 9 · 5 % 66 = 45 (индекс строки 't'). В итоге шифротекст "o7J" корректно дешифруется в слово "Cat".

Вычисление модульных обращений

Модульное обращение можно вычислить методом грубой силы, начав с целых чисел 1, 2, 3 и т.д. Но в случае больших ключей, таких как 8 953 851, это займет слишком много времени.

К счастью, для нахождения модульного обращения можно использовать расширенный алгоритм Евклида, пример реализации которого приведен ниже.

```
def findModInverse(a, m):
    if gcd(a, m) != 1:
        return None  # модульное обращение отсутствует, если a и m
                     # не являются взаимно простыми числами
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3  # здесь // - это оператор целочисленного деления
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2),
          (u3 - q * v3), v1, v2, v3
    return u1 % m
```

Чтобы использовать функцию `findModInverse()`, вам не обязательно знать, как работает расширенный алгоритм Евклида. Если два аргумента, которые вы передаете функции, являются взаимно простыми числами, то она вернет модульное обращение ключа `a`.

Более подробную информацию о том, как работает расширенный алгоритм Евклида, можно найти в Википедии:

`https://ru.wikipedia.org/wiki/Алгоритм_Евклида`

Оператор целочисленного деления

Возможно, вы обратили внимание на оператор `//`, который используется в функции `findModInverse()`, приведенной в предыдущем разделе. Это оператор целочисленного деления. Он выполняет деление двух целых чисел и округляет результат вниз до ближайшего целого. Чтобы увидеть, как работает оператор `//`, введите в интерактивной оболочке следующие выражения.

```
>>> 41 / 7
5.857142857142857
>>> 41 // 7
5
>>> 10 // 5
2
```

В то время как результатом выражения `41 / 7` является число `5,857142857142857`, результат выражения `41 // 7` равен `5`. В тех случаях, когда числа не делятся нацело, оператор `//` полезен для получения целочисленной части ответа (ее называют *частным*), тогда как оператор `%`

возвращает остаток от деления. Результатом вычисления выражения, содержащего оператор `//`, всегда будет целое, а не вещественное число. Как видите, результатом выражения `10 // 5` является `2`, а не `2.0`.

Исходный код модуля `Cryptomath`

Поскольку функции `gcd()` и `findModInverse()` будут использоваться в последующих главах, целесообразно поместить их в отдельный модуль. Откройте в редакторе файлов новое окно, введите в нем приведенный ниже код и сохраните его в файле *cryptomath.py*.

cryptomath.py

```
 1. # Модуль Cryptomath
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. def gcd(a, b):
 5.     # Возвращает НОД чисел a и b, используя алгоритм Евклида
 6.     while a != 0:
 7.         a, b = b % a, a
 8.     return b
 9.
10.
11. def findModInverse(a, m):
12.     # Возвращает модульное обращение a % m,
13.     # т.е. такое число x, что a * x % m = 1
14.
15.     if gcd(a, m) != 1:
16.         return None    # модульное обращение отсутствует, если a и m
                             не являются взаимно простыми числами
17.
18.     # Используем расширенный алгоритм Евклида
19.     u1, u2, u3 = 1, 0, a
20.     v1, v2, v3 = 0, 1, m
21.     while v3 != 0:
22.         q = u3 // v3  # здесь // - это оператор целочисленного деления
23.         v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2),
              (u3 - q * v3), v1, v2, v3
24.     return u1 % m
```

Модуль содержит описанную ранее функцию `gcd()`, а также функцию `findModInverse()`, которая реализует расширенный алгоритм Евклида.

Импортировав модуль *cryptomath.py*, вы сможете поэкспериментировать с ним в интерактивной оболочке.

```
>>> import cryptomath
>>> cryptomath.gcd(24, 32)
8
>>> cryptomath.gcd(37, 41)
1
>>> cryptomath.findModInverse(7, 26)
15
>>> cryptomath.findModInverse(8953851, 26)
17
```

Итак, теперь у нас есть возможность вызывать функции `gcd()` и `findModInverse()` для нахождения НОД и модульного обращения двух чисел.

Резюме

В этой главе было рассмотрено несколько важных математических понятий. Оператор `%` возвращает остаток от деления одного числа на другое. Функция `gcd()` возвращает наибольшее число, на которое нацело делятся оба ее аргумента. Если НОД двух чисел равен 1, то они взаимно простые. Наиболее полезным алгоритмом для нахождения НОД двух чисел является алгоритм Евклида.

В отличие от шифра Цезаря аффинный шифр использует для шифрования букв не просто сложение, а умножение и сложение. Однако не все числа годятся на роль ключей для аффинного шифра. Ключ и размер символьного набора должны быть взаимно простыми числами.

При дешифровании аффинного шифра вы умножаете индекс символа шифротекста на модульное обращение ключа. Модульным обращением $a \% m$ является такое число i, что $(a \cdot i) \% m = 1$. Для вычисления модульных обращений можно использовать расширенный алгоритм Евклида. В главе 23 модульные обращения будут использоваться в шифровании с открытым ключом.

В следующей главе мы напишем программу, реализующую аффинное шифрование. Поскольку мультипликативный шифр — это то же самое, что и аффинный шифр, в котором ключ B равен нулю, в написании отдельной программы для мультипликативного шифра нет никакой необходимости. А с учетом того, что мультипликативный шифр — всего лишь менее надежная версия аффинного шифра, его в любом случае не следует использовать.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Каковы результаты вычисления приведенных ниже выражений?

```
17 % 1000
5 % 5
```

2. Чему равен НОД чисел 10 и 15?
3. Что будет содержать переменная `spam` после выполнения инструкции `spam, eggs = 'hello', 'world'`?
4. НОД чисел 17 и 31 равен 1. Означает ли это, что числа 17 и 31 являются взаимно простыми?
5. Почему числа 6 и 8 не являются взаимно простыми?
6. Напишите формулу для модульного обращения выражения A mod C.

14

ПРОГРАММИРОВАНИЕ АФФИННОГО ШИФРА

"Я хочу иметь возможность шепнуть вам что-то на ухо, даже если нас разделяют тысячи миль, но правительство с этим не согласно".

Филипп Циммерманн, создатель PGP

В главе 13 вы узнали о том, что аффинный шифр — это комбинация мультипликативного шифра и шифра Цезаря (см. главу 5), причем мультипликативный шифр аналогичен шифру Цезаря, за исключением того, что для шифрования сообщений в нем применяется не операция сложения, а операция умножения. В этой главе мы напишем программу, реализующую аффинный шифр. Поскольку процесс аффинного шифрования включает два различных шифра, нам понадобятся два ключа: один — для мультипликативного шифра, другой — для шифра Цезаря. С этой целью мы разобьем одно целое число на два ключа.

В этой главе...

- Кортежи
- Сколько различных ключей может иметь аффинный ключ
- Генерирование случайных ключей

Исходный код программы `Affine Cipher`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *affineCipher.py*. Убедитесь в том, что модуль *pyperclip.py* и модуль *cryptomath.py*, который мы создали в главе 13, находятся в той же папке, что и файл *affineCipher.py*.

affineCipher.py

```
 1. # Программа аффинного шифрования
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import sys, pyperclip, cryptomath, random
 5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
 6.
 7.
 8. def main():
 9.     myMessage = """A computer would deserve to be called intelligent
          if it could deceive a human into believing that it was human."
          -Alan Turing"""
10.     myKey = 2894
11.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
12.
13.     if myMode == 'encrypt':
14.         translated = encryptMessage(myKey, myMessage)
15.     elif myMode == 'decrypt':
16.         translated = decryptMessage(myKey, myMessage)
17.     print('Key: %s' % (myKey))
18.     print('%sed text:' % (myMode.title()))
19.     print(translated)
20.     pyperclip.copy(translated)
21.     print('Full %sed text copied to clipboard.' % (myMode))
22.
23.
24. def getKeyParts(key):
25.     keyA = key // len(SYMBOLS)
```

```
    keyB = key % len(SYMBOLS)
    return (keyA, keyB)


def checkKeys(keyA, keyB, mode):
    if keyA == 1 and mode == 'encrypt':
        sys.exit('Cipher is weak if key A is 1. Choose a different key.')
    if keyB == 0 and mode == 'encrypt':
        sys.exit('Cipher is weak if key B is 0. Choose a different key.')
    if keyA < 0 or keyB < 0 or keyB > len(SYMBOLS) - 1:
        sys.exit('Key A must be greater than 0 and Key B must be
          between 0 and %s.' % (len(SYMBOLS) - 1))
    if cryptomath.gcd(keyA, len(SYMBOLS)) != 1:
        sys.exit('Key A (%s) and the symbol set size (%s) are not
          relatively prime. Choose a different key.' % (keyA,
          len(SYMBOLS)))


def encryptMessage(key, message):
    keyA, keyB = getKeyParts(key)
    checkKeys(keyA, keyB, 'encrypt')
    ciphertext = ''
    for symbol in message:
        if symbol in SYMBOLS:
            # Зашифровать символ
            symbolIndex = SYMBOLS.find(symbol)
            ciphertext += SYMBOLS[(symbolIndex * keyA + keyB) %
              len(SYMBOLS)]
        else:
            ciphertext += symbol  # присоединить символ без шифрования
    return ciphertext


def decryptMessage(key, message):
    keyA, keyB = getKeyParts(key)
    checkKeys(keyA, keyB, 'decrypt')
    plaintext = ''
    modInverseOfKeyA = cryptomath.findModInverse(keyA, len(SYMBOLS))

    for symbol in message:
        if symbol in SYMBOLS:
            # Дешифровать символ
            symbolIndex = SYMBOLS.find(symbol)
            plaintext += SYMBOLS[(symbolIndex - keyB) *
              modInverseOfKeyA % len(SYMBOLS)]
        else:
            plaintext += symbol  # присоединить символ без дешифрования
    return plaintext

```

```
70.
71. def getRandomKey():
72.     while True:
73.         keyA = random.randint(2, len(SYMBOLS))
74.         keyB = random.randint(2, len(SYMBOLS))
75.         if cryptomath.gcd(keyA, len(SYMBOLS)) == 1:
76.             return keyA * len(SYMBOLS) + keyB .
77.
78.
79. # Если файл affineCipher.py выполняется как программа
80. # (а не импортируется как модуль), вызвать функцию main()
81. if __name__ == '__main__':
82.     main()
```

Пример выполнения программы `Affine Cipher`

Запустите программу, нажав клавишу <F5>. Вы должны получить следующий результат.

```
Ключ: 2894
Encrypted text:
"5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!AuaRLQADQALQG93!xQxaGaAfaQ1QX3o1R
QARL9Qda!AafARuQLX1LQALQI1iQX3o1RN"Q-5!1RQP36ARu
Full encrypted text copied to clipboard.
```

В программе аффинного шифрования сообщение `"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human." -Alan Turing` было преобразовано в шифротекст с помощью ключа 2894. Чтобы дешифровать этот шифротекст, скопируйте его в переменную `myMessage` в строке 9 и присвойте переменной `myMode` в строке 11 кода значение `'decrypt'`.

Импорт модулей, настройка констант и функция `main()`

В строках 1 и 2 программы содержатся комментарии, описывающие ее назначение. Далее следует инструкция импорта необходимых модулей.

```
1. # Программа аффинного шифрования
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import sys, pyperclip, cryptomath, random
```

Программа импортирует четыре модуля, предоставляющих следующие функции:

- модуль `sys` — функция `exit()`;
- модуль `pyperclip` — функция `copy()` для копирования сообщения в буфер обмена;
- модуль `cryptomath` (см. главу 13) — функции `gcd()` и `findModInverse()`;
- модуль `random` — функция `randint()` для генерирования случайных ключей.

Строка, хранящаяся в переменной `SYMBOLS`, — это символьный набор, содержащий список всех символов, которые могут быть зашифрованы.

```
 5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
```

Если в сообщении встречается какой-либо символ, не входящий в набор `SYMBOLS`, то он остается незашифрованным. В частности, в приведенном выше примере кавычки и дефис остались незашифрованными, поскольку они не входят в определенный нами символьный набор.

В строке 8 начинается определение функции `main()`, которое оказывается почти таким же, как и в программах для работы с перестановочным шифром. В строках 9–11 присваиваются значения переменным `myMessage`, `myKey` и `myMode`.

```
 8. def main():
 9.     myMessage = """A computer would deserve to be called intelligent
          if it could deceive a human into believing that it was human."
          -Alan Turing"""
10.     myKey = 2894
11.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

Значение, сохраненное в переменной `myMode`, определяет, шифруется или дешифруется сообщение.

```
13.     if myMode == 'encrypt':
14.         translated = encryptMessage(myKey, myMessage)
15.     elif myMode == 'decrypt':
16.         translated = decryptMessage(myKey, myMessage)
```

Если переменная `myMode` равна `'encrypt'`, то выполняется строка 14, и значение, возвращаемое функцией `encryptMessage()`, сохраняется в переменной `translated`. Если же переменная `myMode` равна `'decrypt'`,

то выполняется строка 16, и в переменной translated сохраняется значение, возвращаемое функцией decryptMessage(). Работу этих функций мы рассмотрим позже.

В итоге в переменной translated будет содержаться зашифрованная или дешифрованная версия сообщения, хранящегося в переменной myMessage.

В строке 17 отображается ключ шифрования (подставляется с помощью заполнителя %s), а в строке 18 пользователю сообщается о том, что выводится на экран: зашифрованный или дешифрованный текст.

```
17.     print('Key: %s' % (myKey))
18.     print('%sed text:' % (myMode.title()))
19.     print(translated)
20.     pyperclip.copy(translated)
21.     print('Full %sed text copied to clipboard.' % (myMode))
```

В строке 19 на экране отображается содержимое переменной translated, представляющее собой зашифрованную или дешифрованную версию текста, хранящегося в переменной myMessage, а в строке 20 это содержимое копируется в буфер обмена. В строке 21 программа информирует пользователя о том, что текст скопирован в буфер обмена.

Вычисление и проверка ключей

В отличие от шифра Цезаря, в котором используется один ключ и операция сложения, в аффинном шифре используются два ключа, которые мы назовем ключами A и B, и две операции: сложение и умножение. Поскольку запомнить один ключ легче, чем два, мы воспользуемся математическим трюком, позволяющим преобразовать два ключа в один.

В строке 24 начинается определение функции getKeyParts(), которая разбивает целочисленный ключ на два целых числа, которые будут служить нам ключами A и B.

```
24. def getKeyParts(key):
25.     keyA = key // len(SYMBOLS)
26.     keyB = key % len(SYMBOLS)
27.     return (keyA, keyB)
```

Подлежащий разбиению ключ передается в качестве параметра key. В строке 25 ключ A вычисляется путем целочисленного деления ключа key на размер символьного набора, len(SYMBOLS). Оператор целочисленного деления (//) возвращает частное от деления без остатка. В строке 26

оператор *mod* (`%`) возвращает остаток, который мы будем использовать в качестве ключа B.

Например, если выбран ключ 2894, а строка `SYMBOLS` содержит 66 символов, то ключ A будет равен `2894 // 66 = 43`, а ключ B — `2894 % 66 = 56`.

Чтобы объединить ключи A и B в один ключ, умножьте ключ A на размер символьного набора и прибавьте к полученному произведению ключ B: `(43 * 66) + 56`. В результате вы получите число 2894, являющееся исходным целочисленным ключом.

Примечание

Имейте в виду, что в соответствии с максимой Шеннона ("Враг знает систему!") мы должны предполагать, что хакерам известно все об алгоритме шифрования, включая набор символов и его размер. Мы считаем, что единственным элементом, неизвестным хакеру, является использованный ключ. Безопасность программы шифрования должна зависеть только от секретности ключа, а не от секретности символьного набора или исходного кода программы.

Кортежи

Строка 27 выглядит так, будто возвращает список значений, только вместо квадратных скобок использованы круглые. Так обозначаются значения типа *кортеж* (tuple).

```
27.     return (keyA, keyB)
```

Кортеж аналогичен списку в том смысле, что хранит другие значения, к которым можно получить доступ через индекс или срез. Разница лишь в том, что содержимое кортежа не может быть изменено. Метод `append()` для кортежей не поддерживается.

Поскольку в программе *affineCipher.py* нет необходимости изменять значение, возвращаемое функцией `getKeyParts()`, в данном случае уместнее использовать кортеж, а не список.

Выявление слабых ключей

Аффинное шифрование подразумевает умножение индекса символа в наборе `SYMBOLS` на ключ A и прибавление ключа B. Но если ключ A равен 1, то шифр оказывается очень слабым, поскольку в результате умножения индекса на 1 мы получаем тот же индекс. Точно так же, если ключ B равен нулю, шифр оказывается слабым, ведь добавление нуля к индексу не изме-

няет его. Если ключ A равен 1 и при этом ключ B равен 0, то "зашифрованный" текст будет совпадать с исходным сообщением. Другими словами, текст вообще окажется незашифрованным!

Мы выясняем стойкость ключей с помощью функции `checkKeys()`. Инструкции `if` в строках 31 и 33 проверяют, не равен ли параметр `keyA` единице, а параметр `keyB` — нулю.

```
30. def checkKeys(keyA, keyB, mode):
31.     if keyA == 1 and mode == 'encrypt':
32.         sys.exit('Cipher is weak if key A is 1. Choose a different key.')
33.     if keyB == 0 and mode == 'encrypt':
34.         sys.exit('Cipher is weak if key B is 0. Choose a different key.')
```

Если выполняется хотя бы одно из этих условий, то выполнение программы прекращается и выводится диагностическое сообщение. В строках 32 и 34 вызывается функция `sys.exit()`, которой передается текст сообщения об ошибке. Это необязательный параметр функции, позволяющий отобразить на экране строку, прежде чем работа программы будет завершена.

Проверка, выполняемая в строках 31 и 33, предотвращает шифрование слабыми ключами, но если переменная `mode` равна `'decrypt'`, то такая проверка опускается.

В строке 35 проверяется, не является ли значение `keyA` отрицательным числом, а также не является ли значение `keyB` отрицательным числом и не выходит ли оно за верхнюю границу допустимого диапазона (размер символьного набора минус 1).

```
35.     if keyA < 0 or keyB < 0 or keyB > len(SYMBOLS) - 1:
36.         sys.exit('Key A must be greater than 0 and Key B must be
              between 0 and %s.' % (len(SYMBOLS) - 1))
```

Причины, по которым на значения ключей наложены такие ограничения, будут описаны в следующем разделе. Если проверка хотя бы одного из указанных условий возвращает `True`, то ключи считаются недействительными и программа завершает работу.

Дополнительное условие заключается в том, что ключ A и размер символьного набора должны быть взаимно простыми числами. Это означает, что наибольший общий делитель значений `keyA` и `len(SYMBOLS)` должен быть равен 1. Данное условие проверяется в строке 37, тогда как в строке 38 осуществляется выход из программы, если указанные значения не являются взаимно простыми числами.

```
37.     if cryptomath.gcd(keyA, len(SYMBOLS)) != 1:
38.         sys.exit('Key A (%s) and the symbol set size (%s) are not
              relatively prime. Choose a different key.' % (keyA,
              len(SYMBOLS)))
```

Если все условия в функции `checkKeys()` возвращают значение `False`, то это означает, что никаких проблем с ключами нет. В таком случае выполнение программы продолжается со строки, следующей за той, из которой была вызвана функция `checkKeys()`.

Сколько ключей может иметь аффинный шифр

Попробуем вычислить количество возможных ключей аффинного шифра. Ключ B ограничен размером символьного набора, равным `len(SYMBOLS)`, т.е. 66. На первый взгляд кажется, будто ключ A может быть сколь угодно большим, лишь бы он и размер символьного набора были взаимно простыми числами. В результате складывается впечатление, будто аффинный шифр может иметь бесконечное количество ключей, а значит, его невозможно взломать методом грубой силы.

Однако это не так. Вспомните о том, что большие ключи в шифре Цезаря оказывались равными меньшим ключам благодаря эффекту "завертывания". В случае символьного набора размером 66 ключ 67 в шифре Цезаря создаст тот же зашифрованный текст, что и ключ 1. Аффинному шифру также свойственно подобное "завертывание".

Поскольку ключ B аффинного шифра совпадает с ключом шифра Цезаря, его значения могут изменяться в пределах от 1 до размера символьного набора. Чтобы проверить, не обнаружатся ли какие-то ограничения для ключа A, мы напишем небольшую программу, которая проверит целочисленные значения ключа A в определенном диапазоне, и посмотрим, как будет выглядеть соответствующий шифротекст.

Откройте в редакторе файлов новое окно, введите приведенный ниже код и сохраните его в файле *affineKeyTest.py* в той же папке, где находятся файлы *affineCipher.py* и *cryptomath.py*. Запустите программу, нажав клавишу <F5>.

affineKeyTest.py

```
1. # Эта программа доказывает, что размерность пространства ключей
2. # аффинного шифра ограничена значением len(SYMBOLS) ^ 2
3.
4. import affineCipher, cryptomath
5.
6. message = 'Make things as simple as possible, but not simpler.'
7. for keyA in range(2, 80):
```

```
 8.     key = keyA * len(affineCipher.SYMBOLS) + 1
 9.
10.     if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) == 1:
11.         print(keyA, affineCipher.encryptMessage(key, message))
```

Эта программа импортирует модуль `affineCipher`, чтобы получить функцию `encryptMessage()`, и модуль `cryptomath`, чтобы получить функцию `gcd()`. Мы шифруем строку, содержащуюся в переменной `message`. Переменная цикла `keyA` пробегает значения в диапазоне от 2 до 80, поскольку значения 0 и 1 являются недопустимыми для ключа A, о чем было сказано выше.

На каждой итерации цикла ключ определяется в строке 8 на основе текущего значения `keyA`, а в качестве ключа B всегда используется значение 1, чем и объясняется прибавление 1 в конце. Учитывайте, что в допустимом ключе ключ A и размер символьного набора должны быть взаимно простыми числами. Для этого НОД ключа A и размера символьного набора должен быть равен 1, что проверяется в строке 10. Если данное условие не соблюдается, то вызов функции `encryptMessage()` в строке 11 будет пропущен.

Программа выводит одно и то же сообщение, зашифрованное с помощью различных целочисленных значений ключа A. Результаты работы программы выглядят примерно так.

```
5 0.xTvcin?dXv.XvXn8I3Tv.XvIDXXnE3T,vEhcv?DcvXn8I3TS
7 Tz4Nn1ipKbtnztntpDY NnztnYRttp7 N,n781nKR1ntpDY Nm9
13 ZJH0P7ivuVtPJtPtvhGU0PJtPG8ttvWU0,PWF7Pu87PtvhGU0g3
17 HvTx.oizERX.vX.Xz2mkx.vX.mVXXz?kx,.?6o.EVo.Xz2mkxGy
--опущено--
67 Nblf!uijoht!bt!tjnqmf!bt!qpttjcmf,!cvu!opu!tjnqmfsA
71 0.xTvcin?dXv.XvXn8I3Tv.XvIDXXnE3T,vEhcv?DcvXn8I3TS
73 Tz4Nn1ipKbtnztntpDY NnztnYRttp7 N,n781nKR1ntpDY Nm9
79 ZJH0P7ivuVtPJtPtvhGU0PJtPG8ttvWU0,PWF7Pu87PtvhGU0g3
```

Внимательно проанализировав результаты, можно заметить, что шифротексты, полученные с помощью ключей 5 и 71, совпадают! Более того, совпадение наблюдается еще и для ключей 7 и 73, а также ключей 13 и 79!

Обратите внимание на то, что, вычитая 5 из 71, мы получим 66, т.е. размер символьного набора. Вот почему ключ 71 приводит к тому же результату, что и ключ 5: результат повторяется через каждые 66 ключей. Как видите, аффинный шифр характеризуется тем же эффектом "завертывания" в отношении ключа A, что и в отношении ключа B. Резюмируя, можно утверждать, что значения ключа A также ограничены размером символьного набора.

Если умножить 66 возможных значений ключа A на 66 возможных значений ключа B, то получим 4356 возможных комбинаций. Если исключить все целые числа, которые не могут быть использованы в качестве ключа A, поскольку они не являются взаимно простыми числами с числом 66, то общее количество возможных комбинаций ключей для аффинного шифра уменьшится до 1320.

Написание функции шифрования

Чтобы зашифровать сообщение с помощью программы *affineCipher.py*, прежде всего нужны ключ и само сообщение, которые передаются в функцию `encryptMessage()` в качестве аргументов.

```
41. def encryptMessage(key, message):
42.     keyA, keyB = getKeyParts(key)
43.     checkKeys(keyA, keyB, 'encrypt')
```

В строке 42 мы передаем ключ в функцию `getKeyParts()`, чтобы получить от нее целочисленные значения ключей A и B. Далее мы проверяем, являются ли эти значения допустимыми ключами, передавая их в функцию `checkKeys()`. Если функция `checkKeys()` не завершила работу программы, значит, ключи являются допустимыми, и выполняется остальная часть функции `encryptMessage()` после строки 43.

В строке 44 переменная `ciphertext` инициализируется пустой строкой. Впоследствии в нее будет записано зашифрованное сообщение. В цикле `for`, который начинается в строке 45, мы проходим по всем символам сообщения, добавляя их в переменную `ciphertext`.

```
44.     ciphertext = ''
45.     for symbol in message:
```

К моменту, когда цикл завершит свою работу, переменная `ciphertext` будет содержать всю строку зашифрованного сообщения.

На каждой итерации цикла переменной `symbol` присваивается очередной символ сообщения. Если этот символ содержится в строке `SYMBOLS`, которая представляет наш символьный набор, то мы находим индекс символа и присваиваем его переменной `symbolIndex` в строке 48.

```
46.         if symbol in SYMBOLS:
47.             # Зашифровать символ
48.             symbolIndex = SYMBOLS.find(symbol)
49.             ciphertext += SYMBOLS[(symbolIndex * keyA + keyB) %
```

```
                len(SYMBOLS)]
50.         else:
51.             ciphertext += symbol  # присоединить символ без шифрования
```

Чтобы зашифровать текст, необходимо вычислить индекс зашифрованной буквы. В строке 49 значение `symbolIndex` умножается на `keyA` и к полученному произведению прибавляется `keyB`. Далее вычисляется остаток от деления полученного результата на размер символьного набора, т.е. `len(SYMBOLS)`. Это гарантирует, что вычисленный индекс всегда будет в диапазоне от 0 и `len(SYMBOLS)`, исключая граничное значение. Результирующее число становится индексом зашифрованного символа в наборе `SYMBOLS`. Полученный символ присоединяется в конец строки, хранящейся в переменной `ciphertext`. И все это выполняется в одной строке 49!

Если символ не входит в символьный набор, то он конкатенируется со строкой `ciphertext` в строке 51. Например, кавычки и дефис не входят в символьный набор, поэтому они просто присоединяются.

После того как в цикле будут перебраны все символы исходного сообщения, переменная `ciphertext` будет содержать всю зашифрованную строку. Эта строка возвращается функцией `encryptMessage()` в строке 52.

```
52.     return ciphertext
```

Написание функции дешифрования

Функция `decryptMessage()`, предназначенная для дешифрования текста, во многом совпадает с функцией `encryptMessage()`. Строки 56–58 эквивалентны строкам 42–44.

```
55. def decryptMessage(key, message):
56.     keyA, keyB = getKeyParts(key)
57.     checkKeys(keyA, keyB, 'decrypt')
58.     plaintext = ''
59.     modInverseOfKeyA = cryptomath.findModInverse(keyA, len(SYMBOLS))
```

Однако в процессе дешифрования вместо умножения на ключ A выполняется умножение на его модульное обращение, которое вычисляется путем вызова функции `cryptomath.findModInverse()` (см. главу 13).

Строки 61–68 почти идентичны строкам 45–52 функции `encryptMessage()`, отличается лишь строка 65.

```
61.     for symbol in message:
62.         if symbol in SYMBOLS:
63.             # Дешифровать символ
64.             symbolIndex = SYMBOLS.find(symbol)
65.             plaintext += SYMBOLS[(symbolIndex - keyB) *
                  modInverseOfKeyA % len(SYMBOLS)]
66.         else:
67.             plaintext += symbol  # присоединить символ без дешифрования
68.     return plaintext
```

В функции `encryptMessage()` индекс символа умножался на ключ A и к полученному результату прибавлялся ключ B. В строке 65 функции `decryptMessage()` из индекса символа сначала вычитается ключ B, после чего полученный результат умножается на модульное обращение ключа A. Далее вычисляется остаток от деления полученного числа на размер символьного набора, т.е. `len(SYMBOLS)`.

Так реализуется процесс дешифрования. Теперь рассмотрим, какие изменения необходимо внести в программу *affineCipher.py*, чтобы она выбирала допустимые ключи для аффинного шифра случайным образом.

Генерирование случайных ключей

Не так-то легко подобрать допустимый ключ для аффинного шифра. Проще вызвать функцию `getRandomKey()`, которая случайным образом генерирует допустимый ключ. Для этого достаточно изменить строку 10 таким образом, чтобы возвращаемое значение функции `getRandomKey()` сохранялось в переменной `myKey`.

```
10.     myKey = getRandomKey()
--опущено--
17.     print('Key: %s' % (myKey))
```

Теперь программа сама выбирает случайный ключ и выводит его на экран в строке 17. Разберемся в том, как работает функция `getRandomKey()`.

В строке 72 начинается цикл, условием которого является значение `True`. Этот *бесконечный цикл* будет продолжаться до тех пор, пока не поступит команда выйти из него или пользователь не прервет выполнение программы. Если программа застряла в бесконечном цикле, можно прервать ее выполнение, нажав комбинацию клавиш <Ctrl+C> (<Ctrl+D> в Linux и macOS). В нашем случае функция `getRandomKey()` выйдет из бесконечного цикла с помощью инструкции `return`.

```
71. def getRandomKey():
72.     while True:
73.         keyA = random.randint(2, len(SYMBOLS))
74.         keyB = random.randint(2, len(SYMBOLS))
```

В строках 73 и 74 определяются случайные значения для ключей `keyA` и `keyB` в диапазоне от 2 до размера символьного набора. Это гарантирует, что ключ A или B с недопустимым значением 0 или 1 не будет выбран.

Инструкция `if` в строке 75 проверяет, что являются ли ключ A и размер символьного набора взаимно простыми числами, вызывая функцию `gcd()` из модуля `cryptomath`.

```
75.         if cryptomath.gcd(keyA, len(SYMBOLS)) == 1:
76.             return keyA * len(SYMBOLS) + keyB
```

Если это условие выполняется, то два случайно выбранных ключа объединяются в один путем умножения `keyA` на размер символьного набора и прибавления `keyB` к полученному результату. (Обратите внимание на то, что эта операция противоположна той, которую выполняет функция `getKeyParts()`, разбивающая один целочисленный ключ на два целых числа.) В строке 76 данное значение возвращается функцией `getRandomKey()`.

Если условие в строке 75 равно `False`, то функция возвращается к началу цикла `while` в строке 73 и вновь выбирает случайные числа в качестве значений `keyA` и `keyB`. Бесконечный цикл гарантирует, что программа будет продолжать поиск допустимых ключей до тех пор, пока не найдет их.

Вызов функции `main()`

В строках 81 и 82 функция `main()` вызывается в том случае, если файл был запущен как программа, а не импортирован другой программой как модуль.

```
79. # Если файл affineCipher.py выполняется как программа
80. # (а не импортируется как модуль), вызвать функцию main()
81. if __name__ == '__main__':
82.     main()
```

Это гарантирует, что функция `main()` будет вызвана только при автономном запуске программы.

Резюме

В этой главе мы написали программу *affineKeyTest.py* для тестирования нашего аффинного шифра. Благодаря ей мы узнали о том, что аффинный шифр имеет приблизительно 1320 возможных ключей, а такое количество ключей можно легко взломать методом грубой силы. Это означает, что нам придется выбросить аффинный шифр на свалку легко взламываемых слабых шифров.

Таким образом, аффинный шифр не намного надежнее тех шифров, которые мы обсуждали до него. У перестановочного шифра больше возможных ключей, но их число ограничено длиной сообщения. В случае сообщения длиной 20 символов перестановочный шифр будет иметь самое большее 18 ключей в диапазоне 2–19. Для более надежного шифрования коротких сообщений целесообразнее использовать вместо шифра Цезаря аффинный шифр, поскольку количество ключей в нем зависит от размера символьного набора.

В главе 15 мы напишем программу, которая применяет метод грубой силы для взлома сообщений, зашифрованных с помощью аффинного шифра.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Комбинацией каких двух шифров является аффинный шифр?
2. Что такое кортеж и чем он отличается от списка?
3. Почему аффинный шифр является слабым, если ключ A равен 1?
4. Почему аффинный шифр является слабым, если ключ B равен 0?

15

ВЗЛОМ АФФИННОГО ШИФРА

"Криптоанализ не мог появиться до тех пор, пока цивилизация не достигла достаточно высокого уровня в ряде наук, включая математику, статистику и лингвистику".
Саймон Сингх, "Книга шифров"

В главе 14 вы узнали о том, что аффинный шифр ограничен лишь несколькими тысячами ключей, а это означает, что его можно легко взломать методом грубой силы. В этой главе мы напишем программу, способную взломать сообщения, зашифрованные с помощью аффинного шифра.

В этой главе...

- Оператор возведения в степень (`**`)
- Инструкция `continue`

Исходный код программы Affine Hacker

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *affineHacker.py*. Поскольку ввод строки для переменной `myMessage` вручную вряд ли доставит вам удовольствие, скопируйте ее из файла, доступного на сайте издательства (см. введение), чтобы сэкономить время. Убедитесь в том, что словарь *dictionary.txt*, а также файлы *pyperclip.py*, *affineCipher.py*, *detectEnglish.py* и *cryptomath.py* находятся в той же папке, что и файл *affineHacker.py*.

affineHacker.py

```
 1. # Программа для взлома аффинного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip, affineCipher, detectEnglish, cryptomath
 5.
 6. SILENT_MODE = False
 7.
 8. def main():
 9.     # Этот текст можно скопировать из
10.     # файла примера (см. введение)
11.     myMessage =
          """5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!AuaRLQ
          ADQALQG93!xQxaGaAfaQ1QX3o1RQARL9Qda!AafARuQLX1LQALQI1
          iQX3o1RN"Q-5!1RQP36ARu"""
12.
13.     hackedMessage = hackAffine(myMessage)
14.
15.     if hackedMessage != None:
16.         # Вывод дешифрованного текста на экран;
17.         # для удобства копируем его в буфер обмена
18.         print('Copying hacked message to clipboard:')
19.         print(hackedMessage)
20.         pyperclip.copy(hackedMessage)
21.     else:
22.         print('Failed to hack encryption.')
23.
24.
25. def hackAffine(message):
26.     print('Hacking...')
27.
28.     # Работу программы на Python можно в любой момент прервать,
29.     # нажав <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux)
30.     print('(Press Ctrl-C or Ctrl-D to quit at any time.)')
31.
32.     # Перебор всех возможных ключей методом грубой силы
```

```
33.     for key in range(len(affineCipher.SYMBOLS) ** 2):
34.        keyA = affineCipher.getKeyParts(key)[0]
35.        if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) !=1:
36.             continue
37.
38.         decryptedText = affineCipher.decryptMessage(key, message)
39.         if not SILENT_MODE:
40.             print('Tried Key %s... (%s)' % (key, decryptedText[:40]))
41.
42.         if detectEnglish.isEnglish(decryptedText):
43.             # Подтвердить, что найден правильный ключ
44.             print()
45.             print('Possible encryption hack:')
46.             print('Key: %s' % (key))
47.             print('Decrypted message: ' + decryptedText[:200])
48.             print()
49.             print('Enter D if done, anything else to continue hacking:')
50.             response = input('> ')
51.
52.             if response.strip().upper().startswith('D'):
53.                 return decryptedText
54.     return None
55.
56.
57. # Если файл affineHacker.py выполняется как программа
58. # (а не импортируется как модуль), вызвать функцию main():
59. if __name__ == '__main__':
60.     main()
```

Пример выполнения программы `Affine Hacker`

Запустите программу, нажав клавишу <F5>. Вы должны получить примерно следующее.

```
Hacking...
(Press Ctrl-C or Ctrl-D to quit at any time.)
Tried Key 66... ("5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQA)
Tried Key 67... ("4PF8nk2KZ5PH82 wPwZhZ5eZPK8PcZPFz  ZwP.)
Tried Key 68... ("3OE7mj1JY4OG710vOvYgY4dYOJ7ObYOEy00YvO?)
--опущено--
Tried Key 2892... ("UQw970A.y!QC9A6xQxy?y!ByQ.9QvyQwu66yxQ3)
Tried Key 2893... ("rnFRPSXWHUnZRXOGnGHVHUYHnWRnEHnFDOOHGnL)
Tried Key 2894... ("A computer would deserve to be called i)

Possible encryption hack:
Key: 2894
Decrypted message: "A computer would deserve to be called intelligent if
it could deceive a human into believing that it was human." -Alan Turing
```

```
Enter D if done, anything else to continue hacking:
> d
Copying hacked message to clipboard:
"A computer would deserve to be called intelligent if it could deceive
a human into believing that it was human." -Alan Turing
```

Проанализируем работу программы более подробно.

Импорт модулей, настройка констант и функция `main()`

Программа для взлома аффинного шифра состоит всего лишь из 60 строк, поскольку значительная часть кода уже написана нами. В строке 4 импортируются модули, созданные в предыдущих главах.

```
1. # Программа для взлома аффинного шифра
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, affineCipher, detectEnglish, cryptomath
5.
6. SILENT_MODE = False
```

Запустив программу, вы увидите, что в процессе перебора всех возможных вариантов дешифрования она выводит очень много информации. Это сильно замедляет ее работу. Если хотите ускорить процесс, отключите вывод диагностических сообщений, установив для константы `SILENT_MODE` в строке 6 значение `True`.

Далее создается функция `main()`.

```
 8. def main():
 9.     # Этот текст можно скопировать из
10.     # файла примера (см. введение)
11.     myMessage =
          """5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!AuaRLQ
          ADQALQG93!xQxaGaAfaQ1QX3o1RQARL9Qda!AafARuQLX1LQALQI1
          iQX3o1RN"Q-5!1RQP36ARu"""
12.
13.     hackedMessage = hackAffine(myMessage)
```

Шифротекст, подлежащий взлому, хранится в строке `myMessage`, которая передается функции `hackAffine()` (ее мы рассмотрим в следующем разделе). Эта функция возвращает либо строку исходного сообщения, если шифротекст удалось взломать, или значение `None`, если взлом не удался.

В строках 15–22 проверяется, содержит ли переменная `hackedMessage` значение `None`.

```
15.     if hackedMessage != None:
16.         # Вывод дешифрованного текста на экран;
17.         # для удобства копируем его в буфер обмена
18.         print('Copying hacked message to clipboard:')
19.         print(hackedMessage)
20.         pyperclip.copy(hackedMessage)
21.     else:
22.         print('Failed to hack encryption.')
```

Если переменная `hackedMessage` не равна `None`, то сообщение выводится на экран в строке 19 и копируется в буфер обмена в строке 20. В противном случае программа информирует пользователя о том, что ей не удалось взломать шифротекст. Рассмотрим реализацию функции `hackAffine()`.

Функция взлома аффинного шифра

Функция `hackAffine()` начинается в строке 25 и содержит собственно код взлома аффинного шифра. Но сначала выводится напоминание для пользователя.

```
25. def hackAffine(message):
26.     print('Hacking...')
27.
28.     # Выполнение программы на Python можно в любой момент прервать,
29.     # нажав <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux)
30.     print('(Press Ctrl-C or Ctrl-D to quit at any time.)')
```

Процесс дешифровки может занять какое-то время, поэтому, если пользователь захочет преждевременно прервать работу программы, он сможет нажать <Ctrl+C> (Windows) или <Ctrl+D> (macOS и Linux).

Прежде чем двигаться дальше, необходимо познакомиться с оператором возведения в степень.

Оператор возведения в степень

Помимо базовых арифметических операторов `+`, `-`, `*`, `/` и `//` существует еще один полезный оператор, применяемый в программе взлома аффинного шифра, — это *оператор возведения в степень* (`**`). Он возводит заданное число в степень, определяемую вторым операндом. Например, 2^5 записывается в Python как `2 ** 5`. Это эквивалентно перемножению пяти двоек: `2 * 2 * 2 * 2 * 2`. Оба выражения, `2 ** 5` и `2 * 2 * 2 * 2 * 2`, дают один и тот же результат: число `32`.

Чтобы увидеть, как работает оператор **, введите в интерактивной оболочке следующие выражения.

```
>>> 5 ** 2
25
>>> 2 ** 5
32
>>> 123 ** 10
792594609605189126649
```

Результат вычисления выражения `5 ** 2` равен `25`, поскольку $5 \cdot 5 = 25$. Аналогичным образом выражение `2 ** 5` возвращает `32`, поскольку в результате перемножения пяти двоек мы получаем 32.

Рассмотрим, как этот оператор ** применяется в программе.

Вычисление общего количества возможных ключей

В строке 33 с помощью оператора ** вычисляется общее количество возможных ключей.

```
32.     # Перебор всех возможных ключей методом грубой силы
33.     for key in range(len(affineCipher.SYMBOLS) ** 2):
34.         keyA = affineCipher.getKeyParts(key)[0]
```

Мы знаем, что существует самое большее `len(affineCipher.SYMBOLS)` возможных целочисленных значений ключа A и столько же возможных целочисленных значений ключа B. Чтобы получить полный диапазон возможных значений ключей, мы возводим длину символьного набора в квадрат, используя оператор ** в выражении `len(affineCipher.SYMBOLS) ** 2`.

В строке 34 вызывается функция `getKeyParts()`, которую мы создали в файле *affineCipher.py* для разбивки одного целочисленного ключа на два целых числа. В данном случае мы используем ее для получения ключа A, который должен быть протестирован. Вспомните, что эта функции возвращает кортеж из двух целых чисел, одно из которых — ключ A, а второе — ключ B. В строке 34 первое число кортежа, задаваемое индексом `[0]`, сохраняется в переменной `keyA`.

Например, если функция возвращает кортеж `(42, 22)`, то выражение `(42, 22)[0]` равно `42`. Подобным образом мы вычленяем часть возвращаемого значения, соответствующую ключу A, и сохраняем ее в переменной `keyA`. Ключ B (второе значение в возвращаемом кортеже) игнорируется, поскольку он никак не учитывается при оценке допустимости ключа A. В строках 35 и 36 проверяется, является ли значение `keyA` допустимым ключом A для аффинного шифра, и если это не так, то программа пере-

ходит к следующему ключу. Чтобы понять, каким образом выполнение возвращается в начало цикла, необходимо познакомиться с инструкцией `continue`.

Инструкция `continue`

Инструкция `continue` применяется в циклах `while` и `for`. Когда она встречается в программе, выполнение немедленно переходит в начало цикла, и начинается следующая итерация. То же самое происходит, когда программа достигает конца блока цикла. Однако инструкция `continue` заставляет программу совершить этот переход досрочно.

Введите в интерактивной оболочке следующий код.

```
>>> for i in range(3):
... print(i)
... print('Hello!')
...
0
Hello!
1
Hello!
2
Hello!
```

Цикл `for` пробегает по всем значениям объекта `range` в диапазоне от `0` до `2`. На каждой итерации функция `print('Hello!')` отображает на экране строку `Hello!`.

Сравните этот цикл со следующим примером, в котором перед строкой `print('Hello!')` стоит инструкция `continue`.

```
>>> for i in range(3):
... print(i)
... continue
... print('Hello!')
...
0
1
2
```

Здесь строка `Hello!` вообще не выводится, поскольку инструкция `continue` всякий раз заставляет программу возвращаться в начало цикла `for`, и в результате функция `print('Hello!')` никогда не выполняется.

Инструкцию `continue` часто помещают в блок инструкции `if`, чтобы при определенных условиях программа возвращалась в начало цикла. Вер-

немся к нашему исходному коду и рассмотрим, как с помощью инструкции `continue` пропустить определенные участки кода в зависимости от выбранного ключа.

Использование инструкции `continue` для пропуска кода

В строке 35 программы с помощью функции `gcd()` из модуля `cryptomath` проверяется, являются ли ключ A и размер символьного набора взаимно простыми числами.

```
35.         if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) !=1:
36.             continue
```

Вспомните, что два числа называются взаимно простыми, если их наибольший общий делитель (НОД) равен 1. Если ключ A и размер символьного набора не являются взаимно простыми числами, то условие в строке 35 оказывается равно `True`, и тогда выполняется инструкция `continue` в строке 36. Это заставляет программу вернуться в начало цикла и перейти к следующей итерации. В результате программа пропускает вызов функции `decryptMessage()` в строке 38, так как ключ не является допустимым, и продолжает проверять другие ключи, пока не встретится подходящий.

Как только программа находит подходящий ключ, сообщение дешифруется путем вызова функции `decryptMessage()` с этим ключом в строке 38.

```
38.         decryptedText = affineCipher.decryptMessage(key, message)
39.         if not SILENT_MODE:
40.             print('Tried Key %s... (%s)' % (key, decryptedText[:40]))
```

Если константа `SILENT_MODE` задана равной `False`, то информация о проверяемом ключе выводится на экран, в противном случае вызов функции `print()` в строке 40 пропускается.

В строке 42 вызывается функция `isEnglish()` из модуля `detectEnglish`, чтобы проверить, содержит ли дешифрованное сообщение осмысленный текст на английском языке.

```
42.         if detectEnglish.isEnglish(decryptedText):
43.             # Подтвердить, что найден правильный ключ
44.             print()
45.             print('Possible encryption hack:')
46.             print('Key: %s' % (key))
47.             print('Decrypted message: ' + decryptedText[:200])
48.             print()
```

В случае использования неверного ключа дешифрованное сообщение будет выглядеть как набор случайных символов, и функция `isEnglish()` вернет значение `False`. Если же дешифрованное сообщение распознается как текст на английском языке (по критериям, заложенным в функции `isEnglish()`), то программа выводит его для пользователя.

Мы отображаем фрагмент дешифрованного сообщения, распознанного функцией `isEnglish()` как текст на английском языке, поскольку она могла ошибиться по тем или иным причинам. Если пользователь решит, что текст расшифрован правильно, он может ввести букву **D** и нажать клавишу <Enter>.

```
49.             print('Enter D if done, anything else to continue hacking:')
50.             response = input('> ')
51.
52.             if response.strip().upper().startswith('D'):
53.                 return decryptedText
```

В противном случае пользователю достаточно нажать клавишу <Enter>, и функция `input()` вернет пустую строку, после чего функция `hackAffine()` продолжит проверять другие ключи.

По отступу строки 54 можно понять, что она выполняется после завершения цикла, начатого в строке 33:

```
54.     return None
```

Если цикл `for` завершается и программа достигает строки 54, то это означает, что функция перебрала все возможные ключи, но подходящий так и не нашла. В этом случае функция `hackAffine()` возвращает значение `None`, указывающее на то, что взлом шифротекста не удался.

Если функции удается найти корректный ключ, то дешифрованное сообщение возвращается в строке 53, и строка 54 не будет выполнена.

Вызов функции `main()`

Если мы выполняем файл *affineHacker.py* как программу, то специальная переменная `__name__` будет содержать строку `'__main__'`, а не `'affineHacker'`. В таком случае вызывается функция `main()`.

```
57. # Если файл affineHacker.py выполняется как программа
58. # (а не импортируется как модуль), вызвать функцию main():
59. if __name__ == '__main__':
60.     main()
```

Резюме

Эта глава довольно короткая, поскольку в ней не вводятся никакие новые способы взлома. Как было показано, если количество возможных ключей исчисляется лишь несколькими тысячами, то компьютеру не составит большого труда применить метод грубой силы для перебора всех возможных ключей и определить подходящий ключ с помощью функции `isEnglish()`.

Вы узнали об операторе возведения в степень (`**`), который возводит заданное число в степень, определяемую вторым операндом. Вы также научились применять инструкцию `continue` для досрочного возврата в начало цикла.

К счастью, бо́льшая часть кода, используемого в программе взлома, уже была написана нами ранее и содержится в файлах *affineCipher.py*, *detectEnglish.py* и *cryptomath.py*. Трюк с функцией `main()` позволяет нам повторно использовать имеющиеся функции.

В главе 16 вы узнаете о простом подстановочном шифре, который компьютеры не в состоянии взломать методом грубой силы. Количество возможных ключей в нем исчисляется триллионами триллионов! Жизни не хватит, чтобы перебрать даже малую долю доступных ключей, что делает такой шифр неуязвимым для атаки методом грубой силы.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Чему равно выражение `2 ** 5`?
2. Чему равно выражение `6 ** 2`?
3. Что выводит следующий код?

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

4. Выполняется ли функция `main()`, определенная в файле *affineHacker.py*, если другая программа импортирует модуль `affineHacker`?

16

ПРОГРАММИРОВАНИЕ ПРОСТОГО ПОДСТАНОВОЧНОГО ШИФРА

"Интернет – самая свободная среда из всех, когда-либо созданных человечеством, и в то же время идеально подходящая для надзора. Как оказывается, одно не исключает другого".

Джон Перри Барлоу, основатель организации Electronic Frontier Foundation

В главе 15 вы узнали о том, что аффинный шифр имеет около тысячи возможных ключей, которые компьютер может легко взломать методом грубой силы. Нам нужен шифр с таким количеством возможных ключей, чтобы никаких компьютерных мощностей не хватило для их перебора.

Простой подстановочный шифр (другие названия – *шифр простой замены*, *моноалфавитный шифр*) является одним из тех шифров, которые способны активно противостоять атакам методом грубой силы, поскольку он характеризуется невероятно большим количеством возможных ключей. Даже если бы компьютер мог ежесекундно проверять триллион ключей, для перебора всех ключей ему понадобилось бы 12 миллионов лет! В этой главе мы напишем программу, реализующую простой подстановочный шифр, а заодно познакомимся с несколькими полезными функциями и методами Python.

В этой главе...

- Списковый метод `sort()`
- Как избавиться от повторяющихся символов в строке
- Функции-обертки
- Строковые методы `isupper()` и `islower()`

Как работает простой подстановочный шифр

Чтобы реализовать простой подстановочный шифр, мы выбираем для шифрования каждой буквы алфавита случайную букву, причем все буквы должны использоваться только один раз. В таком шифре ключом всегда является строка из 26 букв алфавита, расположенных в произвольном порядке. Всего существует 403 291 461 126 605 635 584 000 000 возможных ключей простого подстановочного шифра. Это поистине гигантское число! Оно настолько велико, что исключает любую возможность атаки методом грубой силы[1].

Сначала испытаем простой подстановочный шифр, используя карандаш и бумагу. Мы зашифруем сообщение "Attack at dawn."[2] с помощью ключа VJZBGNFEPLITMXDWKQUCRYAHSO. Прежде всего выпишем буквы алфавита и буквы ключа под ними (рис. 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

Рис. 16.1. *Шифрование букв с помощью ключа*

[1] Количество возможных ключей простого подстановочного шифра равно 26 • 25 • 24 • ... • 1, или 26! (факториал числа 26), что составляет 403 291 461 126 605 635 584 000 000. Этот результат получен на основании того факта, что ключами простого подстановочного шифра являются последовательности из 26 букв английского алфавита, расположенных в случайном порядке. В качестве первой буквы ключа можно выбрать любую из 26 букв. Для второй буквы ключа существует 25 вариантов выбора, соответствующих оставшимся буквам алфавита. Для третьей буквы ключа остается 24 варианта выбора и т.д. Отсюда следует, что количество возможных комбинаций определяется произведением чисел от 26 до 1, которое равно факториалу 26.

[2] "Атака на рассвете" – название картины французского художника-баталиста Альфонса де Невиля. – *Примеч. ред.*

Чтобы зашифровать сообщение, найдите букву открытого текста в верхнем ряду и подставьте вместо нее соответствующую букву из нижнего ряда. Таким образом, 'A' превращается в 'V', 'T' — в 'C', 'C' — в 'Z' и т.д. В результате сообщение "Attack at dawn." шифруется как "Vccvzi vc bvax.".

Чтобы расшифровать зашифрованное сообщение, найдите букву из шифротекста в нижнем ряду и замените ее соответствующей буквой из верхнего ряда. Таким образом, 'V' превращается в 'A', 'C' — в 'T', 'Z' — в 'C' и т.д.

В отличие от шифра Цезаря, в котором нижний ряд сдвинут, но сохраняет прежний алфавитный порядок, в простом подстановочном шифре нижний ряд полностью перемешан. Как следствие, количество возможных ключей резко возрастает, что дает огромные преимущества. Недостаток лишь один: запомнить ключ длиной 26 символов намного сложнее. Вероятно, ключ придется записать, но в таком случае необходимо позаботиться о том, чтобы никто не смог его прочитать!

Исходный код программы `Simple Substitution Cipher`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *simpleSubCipher.py*. Убедитесь в том, что файл *pyperclip.py* находится в той же самой папке. Запустите программу, нажав клавишу <F5>.

simpleSubCipher.py

```
 1. # Простой подстановочный шифр
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip, sys, random
 5.
 6.
 7. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 8.
 9. def main():
10.     myMessage = 'If a man is offered a fact which goes against his
          instincts, he will scrutinize it closely, and unless the evidence
          is overwhelming, he will refuse to believe it. If, on the other
          hand, he is offered something which affords a reason for acting
          in accordance to his instincts, he will accept it even on the
          slightest evidence. The origin of myths is explained in this way.
          -Bertrand Russell'
11.     myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
12.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

```
13.
14.     if not keyIsValid(myKey):
15.         sys.exit('There is an error in the key or symbol set.')
16.     if myMode == 'encrypt':
17.         translated = encryptMessage(myKey, myMessage)
18.     elif myMode == 'decrypt':
19.         translated = decryptMessage(myKey, myMessage)
20.     print('Using key %s' % (myKey))
21.     print('The %sed message is:' % (myMode))
22.     print(translated)
23.     pyperclip.copy(translated)
24.     print()
25.     print('This message has been copied to the clipboard.')
26.
27.
28. def keyIsValid(key):
29.     keyList = list(key)
30.     lettersList = list(LETTERS)
31.     keyList.sort()
32.     lettersList.sort()
33.
34.     return keyList == lettersList
35.
36.
37. def encryptMessage(key, message):
38.     return translateMessage(key, message, 'encrypt')
39.
40.
41. def decryptMessage(key, message):
42.     return translateMessage(key, message, 'decrypt')
43.
44.
45. def translateMessage(key, message, mode):
46.     translated = ''
47.     charsA = LETTERS
48.     charsB = key
49.     if mode == 'decrypt':
50.         # Для дешифрования можно использовать тот же самый код,
51.         # нужно лишь поменять местами строки key и LETTERS
52.         charsA, charsB = charsB, charsA
53.
54.     # Цикл по всем символам сообщения
55.     for symbol in message:
56.         if symbol.upper() in charsA:
57.             # Шифрование/дешифрование символа
58.             symIndex = charsA.find(symbol.upper())
59.             if symbol.isupper():
60.                 translated += charsB[symIndex].upper()
```

```
61.             else:
62.                 translated += charsB[symIndex].lower()
63.         else:
64.             # Символ отсутствует в наборе LETTERS; просто добавляем его
65.             translated += symbol
66.
67.     return translated
68.
69.
70. def getRandomKey():
71.     key = list(LETTERS)
72.     random.shuffle(key)
73.     return ''.join(key)
74.
75.
76. if __name__ == '__main__':
77.     main()
```

Пример выполнения программы `Simple Substitution Cipher`

Выполнив программу *simpleSubCipher.py*, вы получите следующий результат.

```
Using key LFWOAYUISVKMNXPBDCRJTQEGHZ
The encrypted message is:
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, ia 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

This message has been copied to the clipboard.
```

Заметьте, что буквы, записанные в исходном сообщении в нижнем регистре, остаются такими же и в шифротексте. То же самое относится и к буквам в верхнем регистре. Простой подстановочный шифр не шифрует пробелы и знаки препинания, оставляя их нетронутыми.

Чтобы дешифровать этот шифротекст, скопируйте его в переменную `myMessage` в строке 10 и поменяйте значение `myMode` на `'decrypt'`. Результат запуска программы должен выглядеть так.

```
Using key LFWOAYUISVKMNXPBDCRJTQEGHZ
The decrypted message is:
If a man is offered a fact which goes against his instincts, he will
scrutinize it closely, and unless the evidence is overwhelming, he will
refuse to believe it. If, on the other hand, he is offered something which
affords a reason for acting in accordance to his instincts, he will accept it
even on the slightest evidence. The origin of myths is explained in this way.
-Bertrand Russell

This message has been copied to the clipboard.
```

Импорт модулей, настройка констант и функция `main()`

Рассмотрим начальный фрагмент программы.

```
1. # Простой подстановочный шифр
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, sys, random
5.
6.
7. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

В строке 4 импортируются модули `pyperclip`, `sys` и `random`. Значением константы `LETTERS` становится строка, содержащая все буквы алфавита в верхнем регистре. Это символьный набор для простого подстановочного шифра.

Функция `main()` напоминает аналогичные функции программ шифрования, рассмотренных в предыдущих главах. В переменных `myMessage`, `myKey` и `myMode` задаются используемые в программе сообщение, ключ и режим соответственно.

```
 9. def main():
10.     myMessage = 'If a man is offered a fact which goes against his
          instincts, he will scrutinize it closely, and unless the evidence
          is overwhelming, he will refuse to believe it. If, on the other
          hand, he is offered something which affords a reason for acting
          in accordance to his instincts, he will accept it even on the
          slightest evidence. The origin of myths is explained in this way.
          -Bertrand Russell'
11.     myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
12.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

В ключах для простого подстановочного шифра можно легко допустить ошибку, поскольку они довольно длинные и должны содержать каждую букву алфавита. Например, можно ввести ключ, в котором какая-то буква отсутствует или встречается дважды. Функция `keyIsValid()` в строке 14 проверяет допустимость ключа, и в случае обнаружения проблемы программа завершает работу, выводя сообщение об ошибке.

```
14.     if not keyIsValid(myKey):
15.         sys.exit('There is an error in the key or symbol set.')
```

В строках 16–19 проверяется, чему равна переменная `myMode` — `'encrypt'` или `'decrypt'`, и соответственно вызывается либо функция `encryptMessage()`, либо функция `decryptMessage()`.

```
16.     if myMode == 'encrypt':
17.         translated = encryptMessage(myKey, myMessage)
18.     elif myMode == 'decrypt':
19.         translated = decryptMessage(myKey, myMessage)
```

Функции `encryptMessage()` и `decryptMessage()` возвращают строку зашифрованного или дешифрованного текста, которая сохраняется в переменной `translated`.

В строке 20 используемый ключ отображается на экране. Далее на экран выводится зашифрованное или дешифрованное сообщение, которое также копируется в буфер обмена.

```
20.     print('Using key %s' % (myKey))
21.     print('The %sed message is:' % (myMode))
22.     print(translated)
23.     pyperclip.copy(translated)
24.     print()
25.     print('This message has been copied to the clipboard.')
```

Строка 25 — последняя в функции `main()`. На этом программа завершает свою работу.

Далее мы рассмотрим, как в функции `keyIsValid()` применяется метод `sort()` для проверки допустимости ключа.

Списковый метод `sort()`

У списков есть метод `sort()`, располагающий элементы списка в определенном порядке. Возможность сортировки очень удобна, когда необходимо проверить, содержат ли два списка одни и те же элементы, пусть и расположенные иначе.

В программе *simpleSubCipher.py* строковое значение ключа допустимо лишь в том случае, если оно включает все символы из набора LETTERS и в нем отсутствуют повторяющиеся символы. Мы можем проверить, является ли строковое значение допустимым ключом, сравнив его отсортированную версию с отсортированным списком LETTERS. Но поскольку сортировать разрешается только списки, а не строки (вспомните о том, что строка — неизменяемый тип данных), мы преобразуем строки в списки, передав их функции list(). Затем, предварительно отсортировав оба списка, мы сравним их между собой. Символы в строке LETTERS уже расположены в алфавитном порядке, но мы все равно отсортируем эту строку, так как впоследствии мы добавим в нее другие символы.

```
28. def keyIsValid(key):
29.     keyList = list(key)
30.     lettersList = list(LETTERS)
31.     keyList.sort()
32.     lettersList.sort()
```

Строка, хранящаяся в переменной key, передается функции list() в строке 29. Полученный список сохраняется в переменной keyList.

В строке 30 константа LETTERS (содержащая строку 'ABCDEFGHIJKLMNOPQRSTUVWXYZ') передается функции list(), которая возвращает список в следующем формате: ['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'].

В строках 31 и 32 списки keyList и lettersList сортируются по алфавиту путем вызова для них метода sort(). Обратите внимание на то, что, как и метод append(), метод sort() изменяет сам объект списка, не возвращая никакого значения.

Отсортированные списки keyList и lettersList должны совпадать, поскольку строка keyList содержала те же символы, что и строка LETTERS, но перемешанные в случайном порядке. В строке 34 проверяется равенство значений keyList и lettersList.

```
34.     return keyList == lettersList
```

Это гарантирует, что параметр key не содержит повторяющихся символов, поскольку в строке LETTERS они отсутствуют. В таком случае функция возвращает значение True. Если же значения keyList и lettersList не совпадают, значит, ключ недопустим, и функция возвращает значение False.

Функции-обертки

В программе *simpleSubCipher.py* код шифрования и дешифрования почти совпадает. В тех случаях, когда имеются два очень похожих фрагмента кода, лучше поместить их в одну функцию и вызывать ее дважды, чем два раза вводить одинаковый код. Это не только сэкономит время, но и, что важнее, позволит избежать появления ошибок в процессе копирования и вставки кода. Дополнительным преимуществом является то, что даже если в код вкралась ошибка, вам придется исправить ее в одном месте, а не нескольких.

Функции-обертки, или *интерфейсные функции* (wrapper functions), позволяют избежать дублирования кода путем обертывания его другой функцией. Как правило, функция-обертка незначительно изменяет аргументы вызова или значение, возвращаемое обернутой функцией. В противном случае в подобном обертывании не было бы никакой необходимости, поскольку можно непосредственно вызвать саму функцию.

В нашем случае обертками являются функции `encryptMessage()` и `decryptMessage()`.

```
37. def encryptMessage(key, message):
38.     return translateMessage(key, message, 'encrypt')
39.
40.
41. def decryptMessage(key, message):
42.     return translateMessage(key, message, 'decrypt')
```

Каждая из них вызывает функцию `translateMessage()` и возвращает полученное от нее значение (данную функцию мы рассмотрим в следующем разделе). Поскольку обе функции-обертки используют одну и ту же функцию `translateMessage()`, в случае, если потребуется модифицировать процесс шифрования, нужно будет внести изменения только в одну функцию, а не в две.

Любой, кто импортирует файл *simpleSubCipher.py*, сможет вызвать функции `encryptMessage()` и `decryptMessage()` точно так же, как и в других программах шифрования, рассмотренных в книге. Функции-обертки имеют понятные имена, благодаря которым пользователи могут догадаться об их назначении, даже не просматривая код. Тем самым значительно облегчается совместное использование кода.

Другие программы смогут шифровать сообщения с использованием различных шифров, импортируя соответствующие модули и вызывая их функции `encryptMessage()`, как показано ниже.

```
import affineCipher, simpleSubCipher, transpositionCipher
--опущено--
ciphertext1 =        affineCipher.encryptMessage(encKey1, 'Hello!')
ciphertext2 = transpositionCipher.encryptMessage(encKey2, 'Hello!')
ciphertext3 =     simpleSubCipher.encryptMessage(encKey3, 'Hello!')
```

Соглашения об именах полезны тем, что, поработав с одной программой шифрования, можно легко перейти к использованию других аналогичных программ. Например, вы уже знаете, что первым аргументом функции всегда является ключ, а вторым — сообщение. Этого соглашения мы последовательно придерживаемся во всех программах шифрования, рассматриваемых в книге. Использование единой функции `translateMessage()` вместо отдельных функций `encryptMessage()` и `decryptMessage()` нарушило бы согласованность с другими программами.

Теперь перейдем к рассмотрению функции `translateMessage()`.

Функция `translateMessage()`

Функция `translateMessage()` предназначена как для шифрования, так и для дешифрования сообщений.

```
45. def translateMessage(key, message, mode):
46.     translated = ''
47.     charsA = LETTERS
48.     charsB = key
49.     if mode == 'decrypt':
50.         # Для дешифрования можно использовать тот же самый код,
51.         # нужно лишь поменять местами строки key и LETTERS
52.         charsA, charsB = charsB, charsA
```

Обратите внимание на то, что у функции есть не только параметры `key` и `message`, но и третий параметр: `mode`. При вызове из функции `encryptMessage()` он равен `'encrypt'`, а при вызове из функции `decryptMessage()` — `'decrypt'`. Именно так функция `translateMessage()` узнает, должна ли она шифровать или дешифровать переданное ей сообщение.

Сам процесс шифрования довольно прост: для каждой буквы, которая встречается в параметре `message`, функция определяет ее индекс в строке `LETTERS` и заменяет соответствующий символ буквой, которая встречается под тем же индексом в параметре `key`. В процессе дешифрования выполняется противоположное действие: находится индекс в параметре `key` и соответствующий символ заменяется буквой с тем же индексом в строке `LETTERS`.

Вместо переменных LETTERS и key в функции используются переменные charsA и charsB, что позволяет заменять буквы в строке charsA буквами из строки charsB с тем же индексом. Возможность перестановки букв в переменных charsA и charsB упрощает переключение между режимами шифрования и дешифрования. В строке 47 в переменную charsA записывается содержимое строки LETTERS, а в строке 48 в переменную charsB записывается строка key.

Процесс шифрования проиллюстрирован на рис. 16.2. В верхнем ряду показаны символы строки charsA (содержит строку LETTERS), в среднем — символы строки charsB (содержит строку key), в нижнем — целочисленные индексы, соответствующие символам. Функция translateMessage() ищет индекс символа в строке charsA и заменяет его символом из строки charsB, имеющим тот же индекс.

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
Индекс	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

Рис. 16.2. *Использование индексов для шифрования сообщения*

Когда нужно дешифровать сообщение, мы меняем переменные charsA и charsB местами с помощью инструкции charsA, charsB = charsB, charsA в строке 52. Процесс дешифрования проиллюстрирован на рис. 16.3.

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
Индекс	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

Рис. 16.3. *Использование индексов для дешифрования шифротекста*

Не забывайте о том, что функция всегда заменяет символы из строки charsA символами из строки charsB, имеющими тот же индекс. Поэтому после перестановки переменных в строке 52 функция выполняет не шифрование, а дешифрование.

В следующем фрагменте кода осуществляется поиск требуемого индекса.

```
54.     # Цикл по всем символам сообщения
55.     for symbol in message:
56.         if symbol.upper() in charsA:
57.             # Шифрование/дешифрование символа
58.             symIndex = charsA.find(symbol.upper())
```

На каждой итерации цикла for в строке 55 переменной symbol присваивается очередной символ строки сообщения. Если этот символ, преобразованный в верхний регистр, встречается в строке charsA (вспомните, что

переменные key и LETTERS содержат лишь символы в верхнем регистре), то в строке 58 определяется его позиция с помощью строкового метода find(). Найденный индекс сохраняется в переменной symIndex.

Мы знаем, что в данном случае метод find() не может вернуть -1 (это означало бы, что методу не удалось обнаружить аргумент в строке), поскольку инструкция if в строке 56 гарантирует, что результат, возвращаемый методом symbol.upper(), встречается в строке charsA. В противном случае строка 58 не будет выполнена.

Далее каждый шифруемый или дешифруемый символ добавляется в строку, которая возвращается функцией translateMessage(). Но поскольку переменные key и LETTERS содержат лишь буквы в верхнем регистре, необходимо определить регистр исходного символа и выполнить соответствующее преобразование. Для этого нужно познакомиться с двумя строковыми методами: isupper() и islower().

Строковые методы isupper() и islower()

Методы isupper() и islower() позволяют проверить, находятся ли все символы строки соответственно в верхнем или нижнем регистре.

В частности, строковый метод isupper() возвращает значение True, когда выполняются следующие два условия:

- строка содержит по крайней мере один символ в верхнем регистре;
- строка не содержит ни одного символа в нижнем регистре.

Строковый метод islower() возвращает значение True при выполнении следующих двух условий:

- строка содержит по крайней мере один символ в нижнем регистре;
- строка не содержит ни одного символа в верхнем регистре.

Наличие в строке небуквенных символов не влияет на результат, однако в тех случаях, когда строка целиком состоит из небуквенных символов, оба метода возвращают значение False. Чтобы увидеть, как работают эти методы, введите в интерактивной оболочке следующие инструкции.

```
>>> 'HELLO'.isupper()
True
❶ >>> 'HELLO WORLD 123'.isupper()
True
❷ >>> 'hello'.islower()
True
>>> '123'.isupper()
False
>>> ''.islower()
False
```

Инструкция ❶ возвращает значение `True`, поскольку строка `'HELLO WORLD 123'` содержит по крайней мере один символ в верхнем регистре и не содержит букв в нижнем регистре. Наличие чисел в этой строке никак не влияет на результат. Метод `'hello'.islower()` (❷) возвращает значение `True`, поскольку строка `'hello'` содержит по крайней мере один символ в нижнем регистре и не содержит букв в верхнем регистре. В последних двух случаях оба метода возвращают значение `False`, так как строки не содержат букв.

Вернемся к рассмотрению функции `translateMessage()`, в которой используется метод `isupper()`.

Сохранение регистра букв с помощью метода `isupper()`

В функции `translateMessage()` с помощью строкового метода `isupper()` гарантируется, что преобразованные буквы сохранят исходный регистр.

```
59.             if symbol.isupper():
60.                 translated += charsB[symIndex].upper()
61.             else:
62.                 translated += charsB[symIndex].lower()
```

В строке 59 проверяется, содержит ли переменная `symbol` букву в верхнем регистре. Если это действительно так, то в строке 60 символ, хранящийся в позиции `charsB[symIndex]`, преобразуется в верхний регистр и конкатенируется со строкой, хранящейся в переменной `translated`. Если же переменная `symbol` содержит букву в нижнем регистре, то в строке 62 символ, хранящийся в позиции `charsB[symIndex]`, преобразуется в нижний регистр и конкатенируется со строкой, хранящейся в переменной `translated`.

Метод `isupper()` возвращает `True`, только когда строка содержит хотя бы одну букву в верхнем регистре. Если в переменной `symbol` окажется небуквенный символ, такой, например, как `'5'` или `'?'`, то условие в строке 59 вернет значение `False`, и вместо строки 60 выполнится строка 62. В этом случае метод `lower()` не окажет никакого влияния на строку, так как в ней отсутствуют буквы. Метод просто вернет исходный небуквенный символ.

Таким образом, в строке 62 учитывается возможное наличие в строке `symbol` символов в нижнем регистре, *а также* небуквенных символов.

Отступ строки 63 указывает на то, что данная инструкция `else` относится к инструкции `if symbol.upper() in charsA:` в строке 56, поэтому строка 65 выполняется в том случае, если символ `symbol` не встречается в строке `LETTERS`.

```
63.         else:
64.             # Символ отсутствует в наборе LETTERS; просто добавляем его
65.             translated += symbol
```

Мы не можем зашифровать или дешифровать такой символ, поэтому просто присоединяем его к строке `translated` в неизменном виде.

В строке 67 функция `translateMessage()` возвращает значение, хранящееся в переменной `translated`, которая содержит зашифрованное или дешифрованное сообщение:

```
67.     return translated
```

Далее мы узнаем, как сгенерировать корректный ключ для простого подстановочного шифра с помощью функции `getRandomKey()`.

Генерирование случайных ключей

Вводить строку ключа, содержащую все символы алфавита, — не слишком приятное занятие. Справиться с этим помогает функция `getRandomKey()`, которая возвращает подходящий ключ. В строках 71–73 осуществляется случайное перемешивание символов в константе `LETTERS`.

```
70. def getRandomKey():
71.     key = list(LETTERS)
72.     random.shuffle(key)
73.     return ''.join(key)
```

Примечание

Если хотите понять, как осуществляется случайное перемешивание символов строки с помощью методов `list()`, `random.shuffle()` и `join()`, прочитайте раздел "Случайное перемешивание строки" в главе 9.

Чтобы задействовать функцию `getRandomKey()`, измените строку 11 следующим образом:

```
11.     myKey = getRandomKey()
```

Поскольку используемый ключ отображается на экране в строке 20, вы увидите, какой ключ вернула функция `getRandomKey()`.

Вызов функции `main()`

Функция `main()` вызывается только в том случае, если файл *simpleSubCipher.py* был запущен как программа, а не импортируется в виде модуля другой программой.

```
76. if __name__ == '__main__':
77.     main()
```

Резюме

В этой главе вы узнали о том, как использовать списковый метод `sort()` для упорядочения элементов списка и как сравнить два упорядоченных списка между собой. Вы также познакомились со строковыми методами `isupper()` и `islower()`, позволяющими проверить, содержит ли строка одни только символы верхнего или нижнего регистра. Еще вы узнали о так называемых функциях-обертках, которые служат для вызова других функций, обычно с той или иной модификацией аргументов.

Количество возможных ключей простого подстановочного шифра настолько велико, что его невозможно взломать методом грубой силы. Это делает его неуязвимым для методик взлома, которые мы применяли в отношении других шифров. Тем не менее в главе 17 вы узнаете, как взломать простой подстановочный шифр. Вместо простого перебора всех возможных ключей методом грубой силы мы используем более изощренный и сложный алгоритм.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Почему взлом простого подстановочного шифра методом грубой силы невозможен даже с помощью сверхмощного компьютера?
2. Что будет содержать переменная `spam` после выполнения следующего кода?

```
spam = [4, 6, 2, 8]
spam.sort()
```

3. Что такое функция-обертка?
4. Что возвращает метод `'hello'.islower()`?
5. Что возвращает метод `'HELLO 123'.isupper()`?
6. Что возвращает метод `'123'.islower()`?

17

ВЗЛОМ ПРОСТОГО ПОДСТАНОВОЧНОГО ШИФРА

"Шифрование – основной способ защиты приватности. Шифрование, в сущности, делает информацию недоступной широкой общественности. Законы, запрещающие использование криптографии, бессильны за пределами границ конкретного государства".

Эрик Хьюз, "Манифест шифропанка" (1993)

В главе 16 вы узнали о невозможности взлома простого подстановочного шифра методом грубой силы ввиду слишком большого количества возможных ключей. Для взлома такого шифра нам потребуется более сложный алгоритм, в котором для поиска возможных вариантов обратного преобразования букв шифротекста в открытый текст используется словарь. В этой главе мы напишем программу, реализующую подобный алгоритм.

В этой главе...

- Шаблоны слов, слова-кандидаты и варианты дешифрования букв
- Регулярные выражения
- Метод `sub()` регулярных выражений

Дешифрование с использованием словарных шаблонов

Атака методом грубой силы предполагает простой перебор всех возможных ключей. Если ключ подобран корректно, то результат дешифрования будет представлять собой осмысленный текст. Однако путем предварительного анализа шифротекста можно значительно сократить количество потенциальных ключей, подлежащих тестированию, а иногда даже определить полный или частичный ключ.

Предположим, что исходный текст в основном состоит из английских слов, включенных в словарь наподобие того, который мы использовали в главе 11. Шифротекст, естественно, не будет содержать подобных слов, но в нем все равно будут встречаться группы букв, разделенных пробелами, подобно словам в предложениях. В отношении таких групп мы употребляем термин *шифрослово*. В подстановочном шифре каждой букве алфавита соответствует ровно одна уникальная зашифрованная буква. Такие буквы мы называем *шифробуквами*. Поскольку каждая буква исходного текста шифруется одной конкретной шифробуквой и в этой версии шифра мы не трогаем пробелы, в исходном тексте и шифротексте будут встречаться одни и те же *шаблоны слов*.

Например, сообщению "MISSISSIPPI SPILL" может соответствовать шифротекст "RJBBJBBJXXJ BXJHH". В обоих случаях первое слово содержит одинаковое количество букв. То же самое относится и ко второму слову. Исходный текст и шифротекст характеризуются одним и тем же шаблоном (т.е. закономерностью) расположения букв и пробелов. Обратите внимание на то, что повторяющиеся буквы сообщения встречаются в том же количестве и в тех же позициях шифротекста.

Таким образом, можно смело предположить, что каждому шифрослову соответствует некое слово, входящее в словарь английского языка, и их шаблоны должны совпадать. Следовательно, если бы мы смогли найти в словаре слово, в которое дешифруется данное шифрослово, то это указало бы нам, как дешифровать каждую его шифробукву. И если с помощью данной методики нам удастся восстановить достаточное число шифробукв, то мы сможем полностью дешифровать сообщение.

Поиск шаблонов слов

В качестве примера исследуем шаблон шифрослова "HGHHU". В нем видны определенные закономерности, которые должны быть свойственны и соответствующему слову в исходном тексте. Оба слова должны иметь следующие общие характеристики:

1) длина слова составляет пять букв;
2) первая, третья и четвертая буквы слова должны быть одинаковыми;
3) слово должно содержать три разные буквы: первую, вторую и пятую.

Задумаемся над тем, какие английские слова укладываются в этот шаблон. Одним из слов, удовлетворяющих описанному критерию, является слово *puppy*, состоящее из пяти букв ('P', 'U', 'P', 'P', 'Y'), три из которых разные ('P', 'U', 'Y'), причем их позиции соответствуют шаблону ('P' — первая, третья и четвертая, 'U' — вторая и 'Y' — пятая). Слова *mommy*, *bobby*, *lulls* и *nanny* тоже подпадают под это описание. Все они, равно как и любое другое слово из файла словаря, удовлетворяющее заданным критериям, являются возможными вариантами дешифрования слова "HGHHU".

Чтобы представить шаблон слова в форме, понятной компьютеру, мы преобразуем его в набор чисел, разделенных точками. Эти числа указывают на повторяемость букв в шаблоне.

Составить шаблон слова несложно: первой букве ставится в соответствие число 0, а первому вхождению любой другой буквы — следующее число, на единицу превышающее предыдущее использованное. Например, шаблоном слова *cat* будет 0.1.2, а шаблоном слова *classification* — 0.1.2.3.3.4.5.4.0.2.6.4.7.8.

В случае простого подстановочного шифра не имеет значения, какой именно ключ используется для шифрования: слово исходного текста и соответствующее ему шифрослово *всегда* будут подчиняться одному и тому же шаблону. Если шаблон шифрослова "HGHHU" — 0.1.0.0.2, значит, исходное слово имеет такой же шаблон.

Поиск возможных вариантов дешифрования букв

Чтобы дешифровать шифрослово "HGHHU", в файле словаря нужно найти все слова, соответствующие шаблону 0.1.0.0.2. Мы будем называть такие слова *кандидатами* для данного шифрослова. Вот как, например, выглядит список кандидатов для шаблона "HGHHU":

- *puppy*;
- *mommy*;
- *bobby*;
- *lulls*;
- *nanny*.

Используя шаблоны слов, мы можем выдвигать предположения относительно того, в какие буквы исходного текста могут дешифровываться шифробуквы. Такие буквы мы будем называть *вариантами дешифрования* для данной шифробуквы. Чтобы взломать сообщение, зашифрованное с помощью простого подстановочного шифра, мы должны найти варианты

дешифрования для всех шифробукв, а затем определить фактические буквы исходного текста, действуя методом исключения. В табл. 17.1 приведены варианты дешифрования шаблона "HGHHU".

Таблица 17.1. Варианты дешифрования шаблона "HGHHU"

Шифробуквы	**H**	**G**	**H**	**H**	**U**
Варианты дешифрования	P	U	P	P	Y
	M	O	M	M	Y
	B	O	B	B	Y
	L	U	L	L	S
	N	A	N	N	Y

Вот что мы получаем на основе табл. 17.1.

1. Для буквы 'H' имеются следующие варианты дешифрования: 'P', 'M', 'B', 'L' и 'N'.
2. Для буквы 'G' имеются следующие варианты дешифрования: 'U', 'O' и 'A'.
3. Для буквы 'U' имеются следующие варианты дешифрования: 'Y' и 'S'.
4. Для всех остальных шифробукв, кроме 'H', 'G' и 'U', в данном примере нет вариантов дешифрования.

В результате формируется *дешифровальный словарь*, в котором буквам алфавита ставятся в соответствие потенциальные варианты дешифрования. По мере накопления зашифрованных сообщений мы будем находить варианты дешифрования для каждой буквы алфавита, но в данном примере наш шифротекст включает лишь шифробуквы 'H', 'G' и 'U', а варианты дешифрования для остальных шифробукв не определены.

Обратите внимание на то, что буква 'U' имеет только два варианта дешифрования ('Y' и 'S') ввиду перекрывания слов-кандидатов, многие из которых заканчиваются буквой 'Y'. *Чем больше перекрывание, тем меньше вариантов дешифрования и тем легче вычислить, в какую букву должна быть дешифрована та или иная шифробуква.*

Чтобы перевести табл. 17.1 в код на языке Python, представим дешифровальный словарь значениями словарного типа (пары "ключ: значение" для букв `'H'`, `'G'` и `'U'` выделены полужирным шрифтом).

```
{'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': ['U', 'O',
'A'], 'H': ['P', 'M', 'B', 'L', 'N'], 'I': [], 'J': [], 'K': [], 'L': [],
'M': [], 'N': [], 'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [],
'U': ['Y', 'S'], 'V': [], 'W': [], 'X': [], 'Y': [], 'Z': []}
```

Этот словарь включает 26 пар "ключ: значение": по одному ключу для каждой буквы алфавита и список вариантов дешифрования для каждой буквы. На данный момент в словаре содержатся варианты дешифрования для букв `'H'`, `'G'` и `'U'`. Всем остальным ключам соответствуют значения в виде пустых списков, `[]`, поскольку варианты дешифрования для них пока не определены.

Если нам удастся свести количество вариантов дешифрования для некоторой шифробуквы всего лишь к одной букве за счет перекрывания дешифровальных словарей, то мы сможем расшифровать данную букву. Даже если нам не удастся добиться этого для всех 26 шифробукв, может оказаться так, что, взломав значительную часть шифробукв, мы будем в состоянии расшифровать бо́льшую часть шифротекста.

Теперь, когда вы познакомились с основной терминологией, перейдем к рассмотрению алгоритма взлома.

Обзор процесса взлома

Простой подстановочный шифр можно легко взломать, используя шаблоны слов. Основные этапы процедуры взлома можно сформулировать следующим образом.

1. Определите шаблоны для всех шифрослов в тексте.
2. Найдите слова-кандидаты, в которые может быть расшифровано каждое шифрослово.
3. Для каждого шифрослова создайте словарь, представляющий варианты дешифрования для каждой шифробуквы.
4. Объедините словари в единую структуру, которую мы назовем *пересечением словарей*.
5. Удалите из пересечения буквы, для которых удалось установить, каким шифробуквам они соответствуют.
6. Дешифруйте шифротекст с помощью установленных шифробукв.

Чем больше слов в шифротексте, тем выше вероятность перекрывания словарей и тем меньше остается вариантов дешифрования для каждой шифробуквы. Это означает, что *чем длиннее сообщение, зашифрованное с помощью простого подстановочного шифра, тем легче его взломать*.

Прежде чем рассматривать исходный код программы, обсудим, как упростить выполнение первых двух этапов описанной выше процедуры взлома. Мы используем тот же файл словаря, что и в главе 11. Кроме того, нам понадобится вспомогательный модуль *wordPatterns.py* для получения упорядоченного списка шаблонов по каждому слову из словаря.

Модуль Word Patterns

Чтобы получить шаблоны слов для каждого слова в словаре *dictionary.txt*, загрузите файл *makeWordPatterns.py*, доступный на сайте издательства (см. введение). Убедитесь в том, что оба этих файла находятся в той же папке, что и программа *simpleSubHacker.py*, рассматриваемая в этой главе.

Программа *makeWordPatterns.py* содержит функцию `getWordPattern()`, которая получает строку (например, `'puppy'`) и возвращает ее шаблон (например, `'0.1.0.0.2'`). Когда вы запустите программу, она должна создать модуль Python *wordPatterns.py*, который включает единственную инструкцию присваивания, занимающую более 43 000 строк кода!

```
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'],
--опущено--
```

Переменная `allPatterns` содержит словарь, в котором ключами являются строки шаблонов слов, а значениями — английские слова, соответствующие данному шаблону. Например, чтобы найти все слова, соответствующие шаблону 0.1.2.1.3.4.5.4.6.7.8, введите в интерактивной оболочке следующие инструкции.

```
>>> import wordPatterns
>>> wordPatterns.allPatterns['0.1.2.1.3.4.5.4.6.7.8']
['BENEFICIARY', 'HOMOGENEITY', 'MOTORCYCLES']
```

В словаре `allPatterns` значением ключа `'0.1.2.1.3.4.5.4.6.7.8'` является список `['BENEFICIARY', 'HOMOGENEITY', 'MOTORCYCLES']`, который содержит три слова, соответствующих данному шаблону.

Примечание

Если при попытке импортировать модуль `wordPatterns` в интерактивной оболочке вы получите сообщение об ошибке `ModuleNotFoundError`, введите следующие инструкции.

```
>>> import sys
>>> sys.path.append('имя_папки')
```

В качестве имени папки укажите каталог, в котором сохранен файл wordPatterns.py. Тем самым вы сообщите интерпретатору, в какой папке следует искать недостающие модули.

Исходный код программы Simple Substitution Hacker

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *simpleSubHacker.py*. Убедитесь в том, что файлы *pyperclip.py*, *simpleSubCipher.py* и *wordPatterns.py* находятся в той же самой папке. Запустите программу, нажав клавишу <F5>.

simpleSubHacker.py

```
 1. # Программа взлома простого подстановочного шифра
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import os, re, copy, pyperclip, simpleSubCipher, wordPatterns,
      makeWordPatterns
 5.
 6.
 7.
 8.
 9.
10. LETTERS = '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, ia 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.     # Определяем варианты дешифрования шифротекста
17.     print('Hacking...')
18.     letterMapping = hackSimpleSub(message)
19.
20.     # Выводим результаты
21.     print('Mapping:')
22.     print(letterMapping)
23.     print()
24.     print('Original ciphertext:')
25.     print(message)
26.     print()
27.     print('Copying hacked message to clipboard:')
28.     hackedMessage = decryptWithCipherletterMapping(message,
          letterMapping)
```

```
    pyperclip.copy(hackedMessage)
    print(hackedMessage)


def getBlankCipherletterMapping():
    # Возвращает пустой дешифровальный словарь
    return {'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': []}


def addLettersToMapping(letterMapping, cipherword, candidate):
    # Параметр letterMapping - это дешифровальный словарь,
    # обрабатываемый данной функцией.
    # Параметр cipherword - это строковое значение шифрослова.
    # Параметр candidate - это английское слово-кандидат,
    # в которое может быть дешифровано данное шифрослово.

    # Функция добавляет в дешифровальный словарь
    # буквы слова-кандидата в качестве вариантов
    # дешифрования шифробукв.


    for i in range(len(cipherword)):
        if candidate[i] not in letterMapping[cipherword[i]]:
            letterMapping[cipherword[i]].append(candidate[i])



def intersectMappings(mapA, mapB):
    # Чтобы построить пересечение словарей, создаем пустой словарь и
    # добавляем в него только варианты, существующие в ОБОИХ словарях
    intersectedMapping = getBlankCipherletterMapping()
    for letter in LETTERS:

        # Пустой список означает "возможна любая буква";
        # в этом случае копируем список целиком
        if mapA[letter] == []:
            intersectedMapping[letter] = copy.deepcopy(mapB[letter])
        elif mapB[letter] == []:
            intersectedMapping[letter] = copy.deepcopy(mapA[letter])
        else:
            # Если буква из словаря mapA[letter] существует в словаре
            # mapB[letter], добавляем ее в intersectedMapping[letter]
            for mappedLetter in mapA[letter]:
                if mappedLetter in mapB[letter]:
```

```
                    intersectedMapping[letter].append(mappedLetter)

    return intersectedMapping


def removeSolvedLettersFromMapping(letterMapping):
    # Шифробуквы, транслируемые только в одну букву, считаются
    # дешифрованными, и соответствующие буквы могут быть удалены из
    # остальных списков. Например, если 'A' транслируется в ['M', 'N'],
    # а 'B' - в ['N'], то мы знаем, что 'B' соответствует 'N', поэтому
    # 'N' можно удалить из списка для 'A'. Это также означает, что 'A'
    # транслируется в ['M']. А поскольку 'A' теперь соответствует
    # единственной букве, можно удалить 'M' из остальных списков
    # (вот почему словарь сокращается в цикле).

    loopAgain = True
    while loopAgain:
        # Сначала предполагаем, что цикл не будет выполнен повторно
        loopAgain = False

        # solvedLetters - список букв в верхнем регистре, имеющих
        # единственное соответствие в словаре letterMapping
        solvedLetters = []
        for cipherletter in LETTERS:
            if len(letterMapping[cipherletter]) == 1:
                solvedLetters.append(letterMapping[cipherletter][0])

        # Если буква установлена, то она не может быть вариантом
        # дешифрования для другой шифробуквы, поэтому она должна
        # быть удалена из других списков
        for cipherletter in LETTERS:
            for s in solvedLetters:
                if len(letterMapping[cipherletter]) != 1 and s in
                  letterMapping[cipherletter]:
                    letterMapping[cipherletter].remove(s)
                    if len(letterMapping[cipherletter]) == 1:
                        # Дешифрована новая буква, продолжаем цикл
                        loopAgain = True
    return letterMapping


def hackSimpleSub(message):
    intersectedMap = getBlankCipherletterMapping()
    cipherwordList = nonLettersOrSpacePattern.sub('',
      message.upper()).split()
    for cipherword in cipherwordList:
        # Получить новый дешифровальный словарь для каждого шифрослова
        candidateMap = getBlankCipherletterMapping()
```

```
119.
120.         wordPattern = makeWordPatterns.getWordPattern(cipherword)
121.         if wordPattern not in wordPatterns.allPatterns:
122.             continue  # этого слова нет в словаре, продолжаем
123.
124.         # Добавить буквы каждого слова-кандидата в словарь
125.         for candidate in wordPatterns.allPatterns[wordPattern]:
126.             addLettersToMapping(candidateMap, cipherword, candidate)
127.
128.         # Найти пересечение нового словаря с существующим пересечением
129.         intersectedMap = intersectMappings(intersectedMap,
                 candidateMap)
130.
131.     # Удалить решенные буквы из других списков
132.     return removeSolvedLettersFromMapping(intersectedMap)
133.
134.
135. def decryptWithCipherletterMapping(ciphertext, letterMapping):
136.     # Возвращает строку шифротекста, дешифрованную с помощью словаря
137.     # шифробукв, в которой неоднозначности заменены подчеркиваниями
138.
139.     # Сначала создаем ключ на основе словаря letterMapping
140.     key = ['x'] * len(LETTERS)
141.     for cipherletter in LETTERS:
142.         if len(letterMapping[cipherletter]) == 1:
143.             # Если имеется только одна буква, добавляем ее в ключ
144.             keyIndex = LETTERS.find(letterMapping[cipherletter][0])
145.             key[keyIndex] = cipherletter
146.         else:
147.             ciphertext = ciphertext.replace(cipherletter.lower(), '_')
148.             ciphertext = ciphertext.replace(cipherletter.upper(), '_')
149.     key = ''.join(key)
150.
151.     # Дешифруем шифротекст с помощью созданного ключа
152.     return simpleSubCipher.decryptMessage(key, ciphertext)
153.
154.
155. if __name__ == '__main__':
156.     main()
```

Пример выполнения программы Simple Substitution Hacker

Когда вы запустите программу, она попытается взломать шифротекст, хранящийся в переменной `message`. Вы должны получить следующий результат.

```
Hacking...
Mapping:
{'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': ['I'], 'T': ['U'], 'U':
['G'], 'V': [], 'W': ['C'], 'X': ['N'], 'Y': ['F'], 'Z': ['Z']}

Original ciphertext:
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, ia 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

Copying hacked message to clipboard:
If a man is offered a fact which goes against his instincts, he will
scrutinize it closel_, and unless the evidence is overwhelming, he will
refuse to _elieve it. If, on the other hand, he is offered something
which affords a reason for acting in accordance to his instincts, he
will acce_t it even on the slightest evidence. The origin of m_ths is
e__lained in this wa_. -_ertrand Russell
```

Исследуем код программы более подробно.

Импорт модулей и настройка констант

Рассмотрим начальные строки программы. В строке 4 импортируются семь модулей — больше, чем в любой из программ, которые мы обсуждали до этого. В строке 10 в глобальной переменной `LETTERS` сохраняется символьный набор, состоящий из букв алфавита в верхнем регистре.

```
  1. # Программа взлома простого подстановочного шифра
  2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
  3.
  4. import os, re, copy, pyperclip, simpleSubCipher, wordPatterns,
       makeWordPatterns
--опущено--
 10. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Модуль `re` содержит функции для работы с регулярными выражениями, с помощью которых можно выполнять сложные манипуляции со строками. Познакомимся с регулярными выражениями поближе.

Поиск символов с помощью регулярных выражений

Регулярные выражения — это строковые шаблоны, соответствующие определенным поисковым строкам. Например, шаблон `'[^A-Z\s]'` в строке 11 — это регулярное выражение, которое сообщает Python о том, что необходимо найти любой символ, не являющийся буквой от 'A' до 'Z' в верхнем регистре или пробельным символом (таким, как пробел, табуляция или символ новой строки).

```
11. nonLettersOrSpacePattern = re.compile('[^A-Z\s]')
```

Функция `re.compile()` создает *объект шаблона регулярного выражения* (далее для краткости — *объект шаблона*), с которым могут работать другие функции модуля `re`. Мы будем использовать этот объект для удаления любых небуквенных символов из шифротекста в разделе "Функция `hackSimpleSub()`".

С помощью регулярных выражений можно выполнять самые разнообразные манипуляции со строками. Дополнительная информация о регулярных выражениях доступна по следующему адресу:

`https://automatetheboringstuff.com/chapter7/`

Функция `main()`

Как и в предыдущих программах взлома, рассмотренных в книге, шифротекст хранится в переменной `message`, которая передается функции `hackSimpleSub()` в строке 18.

```
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, ia 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.     # Определяем варианты дешифрования шифротекста
17.     print('Hacking...')
18.     letterMapping = hackSimpleSub(message)
```

Вместо того чтобы возвращать дешифрованное сообщение или значение `None`, если дешифрование невозможно, функция `hackSimpleSub()` возвращает пересечение словарей шифробукв, из которого удалены дешифрованные буквы. (О том, как создается такое пересечение, мы поговорим далее.)

Затем это пересечение передается функции `decryptWithCipherletterMapping()` для дешифрования шифротекста, сохраненного в переменной `message`, как будет показано в разделе "Дешифрование сообщения".

Пересечение, сохраненное в переменной `letterMapping`, — это словарь, ключами которого являются 26 однобуквенных строк в верхнем регистре, представляющих шифробуквы. Значениями словаря будут списки вариантов дешифрования для каждой шифробуквы. Если каждой шифробукве сопоставлен только один вариант дешифрования, то мы имеем полностью решенный словарь и можем расшифровать любой шифротекст, в котором используются те же шифр и ключ.

Каждый сгенерированный словарь шифробукв зависит от используемого шифротекста. Иногда мы получим лишь частично решенный словарь, в котором для некоторых шифробукв отсутствуют варианты дешифрования или остается несколько вариантов. Короткие шифротексты, содержащие не все буквы алфавита, будут с большей вероятностью приводить к неполным словарям.

Вывод результатов взлома

Далее в программе вызывается функция `print()` для вывода на экран переменной `letterMapping`, исходного шифротекста и дешифрованного сообщения.

```
20.     # Выводим результаты
21.     print('Mapping:')
22.     print(letterMapping)
23.     print()
24.     print('Original ciphertext:')
25.     print(message)
26.     print()
27.     print('Copying hacked message to clipboard:')
28.     hackedMessage = decryptWithCipherletterMapping(message,
          letterMapping)
29.     pyperclip.copy(hackedMessage)
30.     print(hackedMessage)
```

В строке 28 дешифрованное сообщение сохраняется в переменной `hackedMessage`, которая копируется в буфер обмена и выводится на экран, чтобы пользователь мог проверить текст. Для получения дешифрованного сообщения мы используем функцию `decryptWithCipherletterMapping()`, которая будет определена далее.

Теперь рассмотрим все функции, с помощью которых создаются дешифровальные словари.

Создание словарей шифробукв

Программа строит дешифровальный словарь для каждого слова шифротекста. Для этого нам потребуется несколько вспомогательных функций. Одна из них будет создавать пустой словарь шифробукв, и мы сможем вызывать ее для каждого шифрослова.

Еще одна функция будет получать шифрослово, его текущий словарь шифробукв и слово-кандидат. Мы будем вызывать эту функцию для каждого шифрослова и каждого слова-кандидата. Она будет добавлять все варианты дешифрования, определенные на основе данного слова-кандидата, в дешифровальный словарь шифрослова и возвращать этот словарь.

Получив словари шифробукв для нескольких слов шифротекста, мы используем другую функцию для их объединения. И еще одна вспомогательная функция будет находить максимально возможное число решений, сопоставляя дешифрованные буквы с каждой шифробуквой. Как уже отмечалось, мы не всегда сможем решить все шифробуквы, но о том, как справиться с этой проблемой, вы узнаете в разделе "Дешифрование сообщения".

Создание пустого словаря шифробукв

Сначала необходимо создать пустой словарь шифробукв.

```
33. def getBlankCipherletterMapping():
34.     # Возвращает пустой дешифровальный словарь
35.     return {'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': []}
```

Функция `getBlankCipherletterMapping()` возвращает словарь, ключами которого являются односимвольные строки, соответствующие 26 буквам алфавита.

Добавление букв в дешифровальный словарь

В строке 38 создается функция `addLettersToMapping()`, предназначенная для добавления букв в словарь.

```
38. def addLettersToMapping(letterMapping, cipherword, candidate):
```

У функции три параметра: словарь шифробукв (`letterMapping`), шифрослово (`cipherword`) и слово-кандидат, в которое может быть дешифро-

вано данное шифрослово (candidate). Функция сопоставляет каждую букву слова-кандидата с шифробуквой, занимающей соответствующую позицию в шифрослове, и добавляет эту букву в словарь letterMapping, если ее там еще нет.

Например, если 'PUPPY' — слово-кандидат для шифрослова 'HGHHU', то функция addLettersToMapping() добавит в словарь letterMapping значение 'P' для ключа 'H'. Затем функция перейдет к следующей букве и присоединит 'U' к списку, связанному с ключом 'G'.

Если буква уже находится в списке вариантов дешифрования, то функция addLettersToMapping() не добавит ее повторно в список. Например, в слове 'PUPPY' она не станет добавлять букву 'P' в ключ 'H' для следующих двух вхождений буквы 'P', поскольку она там уже имеется. Наконец, функция изменит значение ключа 'U', добавив в его список вариантов дешифрования букву 'Y'.

В функции addLettersToMapping() предполагается, что вызовы len(cipherword) и len(candidate) возвращают одно и то же значение, поскольку мы должны передавать только пары "шифрослово — кандидат" с совпадающими шаблонами.

Функция проходит по каждому индексу строки cipherword, чтобы проверить, не добавлена ли соответствующая буква в список вариантов дешифрования.

```
50.     for i in range(len(cipherword)):
51.         if candidate[i] not in letterMapping[cipherword[i]]:
52.             letterMapping[cipherword[i]].append(candidate[i])
```

В процессе итерирования по буквам шифрослова в качестве индекса в слове-кандидате тоже используется переменная i. Мы можем так поступать, поскольку вариантом дешифрования для шифробуквы cipherword[i], подлежащим добавлению, является буква candidate[i]. Например, для доступа к первым буквам каждой из строк будут использоваться выражения cipherword[0] и candidate[0]. Далее выполнение переходит к инструкции if в строке 51.

Эта инструкция проверяет отсутствие варианта candidate[i] в списке вариантов для данной шифробуквы, пропуская варианты, уже имеющиеся в списке. Соответствующая буква в словаре ищется с помощью выражения letterMapping[cipherword[i]], поскольку cipherword[i] — это ключ в словаре letterMapping, к которому необходимо осуществить доступ. Такая проверка нужна для того, чтобы предотвратить дублирование букв в списке вариантов дешифрования.

Например, первая буква 'P' в слове 'PUPPY' может быть добавлена в словарь letterMapping на первой итерации цикла, но когда i станет равно 2 на третьей итерации, буква 'P' в позиции candidate[2] не будет добавлена в словарь, поскольку она уже была добавлена в него на первой итерации.

Если вариант дешифрования отсутствует в словаре, то в строке 52 новая буква, candidate[i], добавляется в список вариантов дешифрования по ключу letterMapping[cipherword[i]].

Помните: в Python в качестве параметра передается ссылка на словарь, а не сам словарь, поэтому любые изменения, внесенные в словарь letterMapping в функции addLettersToMapping(), будут сохранены и после ее завершения. Соответственно, изменится также словарь candidateMap, передаваемый в функцию addLettersToMapping() в строке 126.

Пройдя по всем индексам в строке cipherword, функция завершит добавление букв в словарь letterMapping. Теперь необходимо выяснить, как программа сравнивает полученный словарь со словарями других шифрослов для выявления пересечений.

Пересечение двух словарей

Функция hackSimpleSub() использует функцию intersectMappings(), которая получает два словаря шифробукв, передаваемых ей в качестве параметров mapA и mapB, и возвращает объединенный словарь. Предварительно создается пустой словарь, а затем в него добавляются варианты дешифрования букв, но только в том случае, если они имеются в *обоих* словарях, что позволяет избежать появления дубликатов.

```
56. def intersectMappings(mapA, mapB):
57.     # Чтобы построить пересечение словарей, создаем пустой словарь и
58.     # добавляем в него только варианты, существующие в ОБОИХ словарях
59.     intersectedMapping = getBlankCipherletterMapping()
```

Сначала в строке 59 с помощью функции getBlankCipherletterMapping() создается пустой словарь шифробукв, который сохраняется в переменной intersectedMapping.

В строке 60 начинается цикл for, в котором перебираются все буквы, хранящиеся в константе LETTERS. Переменная letter используется в качестве ключа в словарях mapA и mapB.

```
60.     for letter in LETTERS:
61.
62.         # Пустой список означает "возможна любая буква";
```

```
63.         # в этом случае копируем список целиком
64.         if mapA[letter] == []:
65.             intersectedMapping[letter] = copy.deepcopy(mapB[letter])
66.         elif mapB[letter] == []:
67.             intersectedMapping[letter] = copy.deepcopy(mapA[letter])
```

В строке 64 проверяется, не является ли список вариантов дешифрования буквы в словаре `mapA` пустым. Пустой список означает, что данная шифробуква может быть дешифрована в любую букву. В таком случае в строке 65 в объединенный словарь копируется весь список вариантов дешифрования из словаря `mapB`. Аналогичная проверка делается в строке 66 для словаря `mapB`. Если оба списка пустые, то условие в строке 64 вернет `True`, и тогда в строке 65 в объединенный словарь просто будет скопирован пустой список из `mapB`.

Блок `else` в строке 68 обрабатывает случай, когда ни список `mapA`, ни список `mapB` для данной буквы не являются пустыми.

```
68.         else:
69.             # Если буква из списка mapA[letter] существует в списке
70.             # mapB[letter], добавляем ее в intersectedMapping[letter]
71.             for mappedLetter in mapA[letter]:
72.                 if mappedLetter in mapB[letter]:
73.                     intersectedMapping[letter].append(mappedLetter)
74.
75.     return intersectedMapping
```

Если списки не пустые, то в строке 71 организуется цикл по буквам, хранящимся в списке `mapA[letter]`. В строке 72 проверяется, встречается ли буква в списке `mapB[letter]`. Если это так, то в строке 73 совпадающая буква добавляется в список `intersectedMapping[letter]` объединенного словаря.

По окончании цикла, начавшегося в строке 60, словарь `intersectedMapping` должен содержать лишь варианты дешифрования букв, которые существуют как в словаре `mapA`, так и в словаре `mapB`. Этот объединенный словарь возвращается в строке 75. Рассмотрим примеры подобных объединений.

Как работают вспомогательные функции

Теперь, когда у нас имеются определения вспомогательных функций, попробуем поработать с ними в интерактивной оболочке, чтобы лучше понять их специфику. Мы создадим пересечение словарей для шифротекста `'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC'`, содержащего всего три шиф-

рослова. Для этого нам потребуется создать по одному словарю для каждого слова, а затем объединить словари.

Импортируем модуль *simpleSubHacker.py* в интерактивной оболочке:

```
>>> import simpleSubHacker
```

Затем вызовем функцию `getBlankCipherletterMapping()` для создания пустого словаря шифробукв и сохраним его в переменной `letterMapping1`.

```
>>> letterMapping1 = simpleSubHacker.getBlankCipherletterMapping()
>>> letterMapping1
{'A': [], 'C': [], 'B': [], 'E': [], 'D': [], 'G': [], 'F': [],
'I': [], 'H': [], 'K': [], 'J': [], 'M': [], 'L': [], 'O': [],
'N': [], 'Q': [], 'P': [], 'S': [], 'R': [], 'U': [], 'T': [],
'W': [], 'V': [], 'Y': [], 'X': [], 'Z': []}
```

Приступим к взлому первого шифрослова, `'OLQIHXIRCKGNZ'`. Прежде всего, нужно получить для него шаблон, вызвав функцию `getWordPattern()` из модуля `makeWordPatterns`.

```
>>> import makeWordPatterns
>>> makeWordPatterns.getWordPattern('OLQIHXIRCKGNZ')
0.1.2.3.4.5.3.6.7.8.9.10.11
```

Чтобы определить, какие слова, содержащиеся в файле английского словаря, соответствуют шаблону 0.1.2.3.4.5.3.6.7.8.9.10.11 (т.е. являются словами-кандидатами шифрослова `'OLQIHXIRCKGNZ'`), импортируем модуль `wordPatterns` и выполним поиск этого шаблона.

```
>>> import wordPatterns
>>> candidates = wordPatterns.allPatterns['0.1.2.3.4.5.3.6.7.8.9.10.11']
>>> candidates
['UNCOMFORTABLE', 'UNCOMFORTABLY']
```

Шаблону шифрослова `'OLQIHXIRCKGNZ'` соответствуют только два английских слова: `'UNCOMFORTABLE'` и `'UNCOMFORTABLY'`. Они являются нашими кандидатами, поэтому сохраним их в переменной `candidates` в виде списка.

Далее следует сопоставить буквы этих слов с буквами шифрослова, используя функцию `addLettersToMapping()`. Сначала сопоставим слово `'UNCOMFORTABLE'`, обратившись к первому элементу списка кандидатов.

```
>>> simpleSubHacker.addLettersToMapping(letterMapping1, 'OLQIHXIRCKGNZ',
candidates[0])
>>> letterMapping1
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['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']}
```

Как показывает анализ словаря `letterMapping1`, буквы шифрослова `'OLQIHXIRCKGNZ'` транслируются в буквы слова `'UNCOMFORTABLE'` следующим образом: `'O'` — `['U']`, `'L'` — `['N']`, `'Q'` — `['C']` и т.д.

Но поскольку шифрослово `'OLQIHXIRCKGNZ'` может быть также дешифровано в слово `'UNCOMFORTABLY'`, необходимо и его добавить в словарь. Введите в интерактивной оболочке следующие инструкции.

```
>>> simpleSubHacker.addLettersToMapping(letterMapping1, 'OLQIHXIRCKGNZ',
candidates[1])
>>> letterMapping1
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['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']}
```

Обратите внимание на то, что содержимое словаря `letterMapping1` почти не изменилось, за исключением того, что теперь шифробукве `'Z'` соответствует не только буква `'E'`, но и `'Y'`. Это объясняется тем, что функция `addLettersToMapping()` добавляет букву в список только в том случае, если до этого она в нем отсутствовала.

Итак, мы получили словарь шифробукв для первого из трех шифрослов. Теперь необходимо построить словарь для второго шифрослова, `'PLQRZKBZB'`, повторив весь процесс.

```
>>> letterMapping2 = simpleSubHacker.getBlankCipherletterMapping()
>>> wordPat = makeWordPatterns.getWordPattern('PLQRZKBZB')
>>> candidates = wordPatterns.allPatterns[wordPat]
>>> candidates
['CONVERSES', 'INCREASES', 'PORTENDED', 'UNIVERSES']
>>> for candidate in candidates:
...   simpleSubHacker.addLettersToMapping(letterMapping2, 'PLQRZKBZB',
candidate)
...
>>> letterMapping2
{'A': [], 'C': [], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': [], 'F': [],
'I': [], 'H': [], 'K': ['R', 'A', 'N'], 'J': [], 'M': [], 'L': ['O',
```

```
'N'], 'O': [], 'N': [], 'Q': ['N', 'C', 'R', 'I'], 'P': ['C', 'I', 'P',
'U'], 'S': [], 'R': ['V', 'R', 'T'], 'U': [], 'T': [], 'W': [], 'V': [],
'Y': [], 'X': [], 'Z': ['E']}
```

Вместо того чтобы четырежды вводить функцию `addLettersToMapping()` для каждого из четырех слов-кандидатов, мы организуем цикл `for` по словам-кандидатам в списке. Таким образом, мы получили словарь шифробукв для второго шифрослова.

Следующий шаг — создание пересечения словарей, сохраненных в переменных `letterMapping1` и `letterMapping2`. Для этого мы передаем словари в функцию `intersectMappings()`. Введите в интерактивной оболочке следующие инструкции.

```
>>> intersectedMapping = simpleSubHacker.intersectMappings(letterMapping1,
letterMapping2)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': ['B'],
'F': [], 'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L':
['N'], 'O': ['U'], 'N': ['L'], 'Q': ['C'], 'P': ['C', 'I', 'P', 'U'],
'S': [], 'R': ['R'], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X':
['F'], 'Z': ['E']}
```

Теперь список вариантов дешифрования для любой шифробуквы, входящей в объединенный словарь, должен содержать лишь варианты, *общие* для словарей `letterMapping1` и `letterMapping2`. Например, список для ключа `'Z'` включает лишь одну букву, `['E']`, поскольку словарь `letterMapping1` содержал список `['E', 'Y']`, тогда как словарь `letterMapping2` — только список `['E']`.

Далее повторяем все то же самое для третьего шифрослова, `'MPBKSSIPLC'`.

```
>>> letterMapping3 = simpleSubHacker.getBlankCipherletterMapping()
>>> wordPat = makeWordPatterns.getWordPattern('MPBKSSIPLC')
>>> candidates = wordPatterns.allPatterns[wordPat]
>>> for i in range(len(candidates)):
...   simpleSubHacker.addLettersToMapping(letterMapping3, 'MPBKSSIPLC',
candidates[i])
...
>>> letterMapping3
{'A': [], 'C': ['Y', 'T'], 'B': ['M', 'S'], 'E': [], 'D': [], 'G': [],
'F': [], 'I': ['E', 'O'], 'H': [], 'K': ['I', 'A'], 'J': [], 'M': ['A',
'D'], 'L': ['L', 'N'], 'O': [], 'N': [], 'Q': [], 'P': ['D', 'I'], 'S':
['T', 'P'], 'R': [], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X':
[], 'Z': []}
```

Введите в интерактивной оболочке следующие инструкции, чтобы создать пересечение словарей letterMapping3 и intersectedMapping (последний представляет собой пересечение словарей letterMapping1 и letterMapping2).

```
>>> intersectedMapping = simpleSubHacker.intersectMappings(letterMapping3,
intersectedMapping)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], F': [],
'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['A', 'D'], 'L': ['N'],
'O': ['U'], 'N': ['L'], 'Q': ['C'], 'P': ['I'], 'S': ['T', 'P'], 'R':
['R'], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

В этом примере мы можем найти решения лишь для ключей, в списке которых содержится единственное значение. Например, шифробуква 'K' дешифруется в 'A'. Но обратите внимание на то, что ключ 'M' может быть дешифрован либо в 'A', либо в 'D'. Поскольку мы знаем, что 'K' дешифруется в 'A', можно сделать вывод, что ключ 'M' должен дешифроваться в 'D', а не в 'A'. Другими словами, если достоверно известно, что определенная буква уже используется для какой-то шифробуквы, то она не может быть использована для другой шифробуквы, поскольку такова особенность простого подстановочного шифра.

Рассмотрим, каким образом функция removeSolvedLettersFromMapping() находит такие буквы и удаляет их из списков вариантов дешифрования. Нам еще понадобится только что созданный объединенный словарь intersectedMapping, поэтому не закрывайте окно IDLE.

Выявление достоверно установленных букв в словарях

Функция removeSolvedLettersFromMapping() ищет в параметре letterMapping шифробуквы, имеющие лишь один вариант дешифрования. Они считаются достоверно установленными, а значит, никакие другие шифробуквы не могут содержать в своих списках данные варианты дешифрования. Это может вызвать цепную реакцию, ведь если буква исключается из списка вариантов дешифрования, содержащего всего два элемента, то мы получаем новую достоверно дешифрованную шифробукву. Функция обрабатывает подобные ситуации в цикле, удаляя из словаря достоверно установленные буквы.

```
 78. def removeSolvedLettersFromMapping(letterMapping)::
--опущено--
 88.     loopAgain = True
```

```
89.     while loopAgain:
90.         # Сначала предполагаем, что цикл не будет выполнен повторно
91.         loopAgain = False
```

Не забывайте, что в параметре `letterMapping` передается ссылка на словарь, поэтому все изменения, внесенные в словарь внутри функции, сохранятся и после ее завершения. В строке 88 создается переменная `loopAgain`, хранящая булево значение, которое определяет, должна ли функция пройти цикл повторно, если будет найдена еще одна достоверно установленная буква.

В строке 89 функция входит в цикл `while`. В начале цикла переменная `loopAgain` устанавливается равной `False`. Тем самым предполагается, что данная итерация станет последней. Переменная `loopAgain` будет установлена в `True` только в том случае, если в ходе итерации программа найдет новую шифробукву с достоверно установленным вариантом дешифрования.

Далее определяются шифробуквы, для которых имеется ровно один вариант дешифрования. Такие варианты дешифрования будут удаляться из словаря.

```
93.         # solvedLetters - список букв в верхнем регистре, имеющих
94.         # единственное соответствие в словаре letterMapping
95.         solvedLetters = []
96.         for cipherletter in LETTERS:
97.             if len(letterMapping[cipherletter]) == 1:
98.                 solvedLetters.append(letterMapping[cipherletter][0])
```

В строке 96 организуется цикл `for` по всем возможным 26 шифробуквам. Мы будем проверять списки вариантов дешифрования для каждой шифробуквы (т.е. списки `letterMapping[cipherletter]`).

В строке 97 проверяется, не равна ли длина такого списка 1. Если это так, то мы знаем, что для данной шифробуквы существует всего один вариант дешифрования, а значит, шифробуква дешифрована. В строке 98 соответствующий вариант дешифрования добавляется в список `solvedLetters`. Этой букве всегда соответствует элемент `letterMapping[cipherletter][0]`, так как в списке `letterMapping[cipherletter]` всего один элемент.

По завершении цикла `for`, начавшегося в строке 96, переменная `solvedLetters` будет содержать список всех достоверно установленных вариантов дешифрования шифробукв. Далее функция проверяет, не числятся ли они в списках вариантов дешифрования других шифробукв, и удаляет их из этих списков.

С этой целью в строке 103 организуется цикл `for` по всем 26 возможным шифробуквам для просмотра их списков.

```
103.         for cipherletter in LETTERS:
104.             for s in solvedLetters:
105.                 if len(letterMapping[cipherletter]) != 1 and s in
                       letterMapping[cipherletter]:
106.                     letterMapping[cipherletter].remove(s)
107.                     if len(letterMapping[cipherletter]) == 1:
108.                         # Решена новая буква, продолжаем цикл
109.                         loopAgain = True
110.     return letterMapping
```

В строке 104 для каждой шифробуквы организуется циклический просмотр всех букв в списке `solvedLetters` с целью проверки того, не встречаются ли они в списке вариантов дешифрования `letterMapping [cipherletter]`.

В строке 105 проверяется как выполнение условия `len(letter-Mapping[cipherletter]) != 1` (т.е. шифробуква еще не дешифрована), так и наличие достоверно установленной буквы в списке вариантов дешифрования данной шифробуквы. В случае выполнения обоих критериев достоверно установленная буква `s` удаляется из списка вариантов дешифрования в строке 106.

Если в результате такого удаления в списке вариантов дешифрования остается только одна буква, то значение переменной `loopAgain` устанавливается равным `True` в строке 109, и тем самым функция сможет удалить эту букву из других списков на следующей итерации цикла.

Как только на очередной итерации для переменной `loopAgain` не будет установлено значение `True`, цикл `while`, начатый в строке 89, завершится, и функция вернет словарь `letterMapping` в строке 110.

Теперь переменная `letterMapping` будет содержать частично или полностью решенный словарь шифробукв.

Тестирование функции `removeSolvedLetterFromMapping()`

Рассмотрим работу функции `removeSolvedLetterFromMapping()` на конкретном примере, протестировав ее в интерактивной оболочке. Вернитесь в окно IDLE, в котором создали словарь `intersectedMapping`. (Не переживайте, если успели закрыть это окно. Просто введите заново инструкции, приведенные в разделе "Как работают вспомогательные функции", после чего вы сможете выполнить приведенный ниже пример.)

Чтобы удалить все достоверно установленные буквы из словаря `intersectedMapping`, введите в интерактивной оболочке следующие инструкции.

```
>>> letterMapping = simpleSubHacker.removeSolvedLettersFromMapping(
intersectedMapping)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], 'F':
[], 'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['D'], 'L': ['N'],
'O': ['U'], 'N': ['L'], 'Q': ['C'], 'P': ['I'], 'S': ['P'], 'R': ['R'],
'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

Обратите внимание на то, что у буквы `'M'` остался лишь один вариант дешифрования — `'D'`, как мы и предсказывали. Более того, теперь у каждой шифробуквы имеется единственный вариант дешифрования, а значит, мы можем приступить к дешифрованию сообщения, используя полученный словарь. Нам еще раз понадобится обратиться к примеру в окне интерактивной оболочки, поэтому оставьте его открытым.

Функция `hackSimpleSub()`

Теперь, когда вы увидели на конкретных примерах, как функции `getBlankCipherletterMapping()`, `addLettersToMapping()`, `intersectMappings()` и `removeSolvedLettersFromMapping()` манипулируют словарями шифробукв, мы можем воспользоваться ими для дешифрования сообщения.

В строке 113 определяется функция `hackSimpleSub()`, которая получает шифротекст сообщения и задействует вспомогательные функции для того, чтобы вернуть частично или полностью решенный словарь шифробукв.

```
113. def hackSimpleSub(message):
114.     intersectedMap = getBlankCipherletterMapping()
115.     cipherwordList = nonLettersOrSpacePattern.sub('',
           message.upper()).split()
```

В строке 114 мы создаем новый словарь шифробукв и сохраняем его в переменной `intersectedMap`. В конечном счете эта переменная будет содержать объединенный словарь, являющийся пересечением словарей каждого из шифрослов.

В строке 115 мы удаляем из сообщения все небуквенные символы. Объекту регулярного выражения `nonLettersOrSpacePattern` соответствует любая строка, не являющаяся буквой или пробельным символом. Для этого объекта вызывается метод `sub()`, имеющий два аргумента. Данный метод ищет поисковые шаблоны во втором аргументе и заменяет их строкой, указанной в первом аргументе, после чего возвращает строку, в которой

выполнены все подобные замены. В нашем случае метод `sub()` просматривает строку сообщения в верхнем регистре и заменяет все небуквенные символы в ней пустыми строками (`''`). В результате возвращается строка, из которой удалены все знаки препинания и цифры, и эта строка разбивается на слова с помощью метода `split()`.

Переменная `cipherwordList`, создаваемая в строке 115, будет содержать список шифрослов в верхнем регистре, ранее хранившихся в параметре `message`.

В строке 116 запускается цикл `for`, в котором переменная `cipherword` пробегает по всем строкам списка `cipherwordList`. В теле цикла создается пустой словарь, после чего запрашивается список слов-кандидатов для текущего шифрослова, буквы каждого слова-кандидата добавляются в словарь шифробукв и строится пересечение этого словаря со словарем `intersectedMap`.

```
116.     for cipherword in cipherwordList:
117.         # Получить новый дешифровальный словарь для каждого шифрослова
118.         candidateMap = getBlankCipherletterMapping()
119.
120.         wordPattern = makeWordPatterns.getWordPattern(cipherword)
121.         if wordPattern not in wordPatterns.allPatterns:
122.             continue  # этого слова нет в словаре, продолжаем
123.
124.         # Добавить буквы каждого слова-кандидата в словарь
125.         for candidate in wordPatterns.allPatterns[wordPattern]:
126.             addLettersToMapping(candidateMap, cipherword, candidate)
127.
128.         # Найти пересечение нового словаря с существующим пересечением
129.         intersectedMap = intersectMappings(intersectedMap,
                 candidateMap)
```

В строке 118 с помощью функции `getBlankCipherletterMapping()` создается новый пустой словарь шифробукв, который сохраняется в переменной `candidateMap`.

Чтобы найти слова-кандидаты для текущего шифрослова, в строке 120 мы вызываем функцию `getWordPattern()` из модуля `makeWordPatterns`. В некоторых случаях шифрословом может оказаться имя или редко употребляемое слово, отсутствующее в нашем английском словаре, и тогда вполне вероятно, что его шаблон тоже будет отсутствовать в файле *wordPatterns.py*. Если шаблон шифрослова не встречается среди ключей словаря `wordPatterns.allPatterns`, значит, слово исходного текста отсутствует в файле словаря. В таком случае дешифровальный словарь для

него не строится, и инструкция continue в строке 122 вызывает переход к следующей итерации цикла.

Если выполнение достигло строки 125, то это означает, что шаблон слова встречается в словаре wordPatterns.allPatterns. Значениями словаря allPatterns являются списки английских слов, а ключами — шаблоны слов. Поскольку значения существуют в виде списка, мы итерируем по ним, используя цикл for. На каждой итерации цикла переменной candidate присваивается очередное слово-кандидат.

В строке 126 вызывается функция addLettersToMapping(), которая обновляет словарь шифробукв candidateMap, используя буквы каждого из слов-кандидатов. Эта функция непосредственно изменяет содержимое словаря.

После добавления всех букв из слов-кандидатов в словарь шифробукв, хранящийся в переменной candidateMap, в строке 129 создается пересечение словаря candidateMap со словарем intersectedMap, и результат записывается обратно в переменную intersectedMap.

Далее функция возвращается в начало цикла for в строке 116, чтобы создать новый словарь для следующего шифрослова из списка cipherwordList и найти его пересечение со словарем intersectedMap. Формирование пересечений продолжается до тех пор, пока не будет пройден весь список cipherwordList.

Получив окончательное пересечение, охватывающее словари всех шифрослов сообщения, мы передаем его функции removeSolvedLettersFromMapping() в строке 132, которая исключает из него достоверно установленные буквы.

```
131.     # Удалить решенные буквы из других списков
132.     return removeSolvedLettersFromMapping(intersectedMap)
```

Обработанный словарь шифробукв, полученный в результате выполнения функции removeSolvedLettersFromMapping(), возвращается функцией hackSimpleSub(). Теперь у нас есть частичное решение, поэтому можно приступать к дешифрованию сообщения.

Строковый метод replace()

Строковый метод replace() возвращает новую строку с замещенными символами. Первый аргумент — искомая подстрока, второй — строка замены. Введите в интерактивной оболочке следующие инструкции.

```
>>> 'mississippi'.replace('s', 'X')
'miXXiXXippi'
>>> 'dog'.replace('d', 'bl')
'blog'
>>> 'jogger'.replace('ger', 's')
'jogs'
```

Мы используем строковый метод `replace()` в функции `decryptWithCipherletterMapping()`.

Дешифрование сообщения

Чтобы расшифровать наше сообщение, мы используем функцию `decryptMessage()`, которая была создана в программе *simpleSubCipher.py*. Однако функция `simpleSubCipher.decryptMessage()` выполняет дешифрование с помощью ключа, а не словаря шифробукв, поэтому вызвать ее напрямую нельзя. Для решения проблемы мы напишем функцию `decryptWithCipherletterMapping()`, которая получает дешифровальный словарь, преобразует его в ключ, а затем передает этот ключ вместе с сообщением функции `simpleSubCipher.decryptMessage()`, возвращая дешифрованную строку. Вспомните, что ключами в простом подстановочном шифре являются строки, состоящие из 26 символов. Символ с индексом 0 в строке ключа шифрует букву 'A', символ с индексом 1 шифрует букву 'B' и т.д.

Для начала нам нужно создать список-заполнитель следующего вида: `['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']`. Буква `'x'` подходит в качестве заполнителя, поскольку в реальном ключе используются только буквы в верхнем регистре. (В качестве заполнителя можно выбрать любой символ, не являющийся буквой в верхнем регистре.) Поскольку не все буквы могут быть дешифрованы, мы должны иметь возможность различать элементы списка, заполненные вариантами дешифрования, и элементы, для которых варианты дешифрования не были найдены. Буква `'x'` как раз и обозначает не дешифрованные буквы.

Рассмотрим, как все это реализовано в исходном коде.

```
135. def decryptWithCipherletterMapping(ciphertext, letterMapping):
136.     # Возвращает строку шифротекста, дешифрованную с помощью словаря
137.     # шифробукв, в которой неоднозначности заменены подчеркиваниями
138.
139.     # Сначала создаем ключ на основе словаря letterMapping
```

```
140.     key = ['x'] * len(LETTERS)
141.     for cipherletter in LETTERS:
142.         if len(letterMapping[cipherletter]) == 1:
143.             # Если имеется только одна буква, добавляем ее в ключ
144.             keyIndex = LETTERS.find(letterMapping[cipherletter][0])
145.             key[keyIndex] = cipherletter
```

В строке 140 создается список заполнителей путем реплицирования списка `['x']`, состоящего из одного элемента, 26 раз. `LETTERS` — это строка, содержащая все буквы алфавита, поэтому вызов `len(LETTERS)` возвращает значение 26. Результатом применения оператора умножения (`*`) к списку является репликация списка заданное число раз.

В строке 141 организуется цикл, в котором переменная `cipherletter` "пробегает" по всем буквам, содержащимся в константе `LETTERS`, и если вариант дешифрования для шифробуквы `cipherletter` достоверно установлен (т.е. список `letterMapping[cipherletter]` содержит всего одну букву), то заполнитель `'x'` в соответствующей позиции списка `key` заменяется буквой `cipherletter`.

Элемент `letterMapping[cipherletter][0]` в строке 144 — это дешифрованная буква, а `keyIndex` — ее индекс в списке `LETTERS`, возвращаемый методом `find()`. В строке 145 в качестве значения элемента списка `key` с этим индексом устанавливается шифробуква.

Но если шифробуква так и не была дешифрована, то функция вставляет вместо нее символ подчеркивания, указывающий на то, что символ пока не определен. В строке 147 символом подчеркивания заменяются буквы в нижнем регистре, а в строке 148 — буквы в верхнем регистре.

```
146.         else:
147.             ciphertext = ciphertext.replace(cipherletter.lower(), '_')
148.             ciphertext = ciphertext.replace(cipherletter.upper(), '_')
```

После выполнения всех описанных замен функция объединяет список строк в одну строку с помощью метода `join()`, создавая ключ для простого подстановочного шифра. Далее эта строка передается функции `decryptMessage()` из модуля `simpleSubCipher`.

```
149.     key = ''.join(key)
150.
151.     # Дешифруем шифротекст с помощью созданного ключа
152.     return simpleSubCipher.decryptMessage(key, ciphertext)
```

Наконец, в строке 152 возвращается дешифрованное сообщение, полученное от функции `decryptMessage()`. Теперь мы располагаем всеми

функциями, которые необходимы для построения полного пересечения всех словарей шифробукв, взлома ключа и дешифрования сообщения. Рассмотрим пример того, как эти функции применяются в интерактивной оболочке.

Дешифрование сообщения в интерактивной оболочке

Вернемся к примеру из раздела "Как работают вспомогательные функции". Для дешифрования сообщения `'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC'` мы воспользуемся переменной `intersectedMapping`, которая ранее была создана в интерактивной оболочке.

Введите в интерактивной оболочке следующие инструкции.

```
>>> simpleSubHacker.decryptWithCipherletterMapping('OLQIHXIRCKGNZ
PLQRZKBZB MPBKSSIPLC', intersectedMapping)
UNCOMFORTABLE INCREASES DISAPPOINT
```

Итак, шифротекст дешифруется в сообщение "Uncomfortable increases disappoint". Как видите, функция `decryptWithCipherletterMapping()` сработала идеально и вернула полностью дешифрованное сообщение. Однако этот пример не демонстрирует того, что происходит в тех случаях, когда не все символы, появляющиеся в шифротексте, удается расшифровать. Мы можем увидеть, что произойдет при отсутствии варианта дешифрования для какой-либо шифробуквы, удалив из объединенного пересечения `intersectedMapping` решения для шифробукв `'M'` и `'S'` с помощью следующих инструкций.

```
>>> intersectedMapping['M'] = []
>>> intersectedMapping['S'] = []
```

Попробуем снова дешифровать шифротекст с помощью словаря `intersectedMapping`.

```
>>> simpleSubHacker.decryptWithCipherletterMapping('OLQIHXIRCKGNZ
PLQRZKBZB MPBKSSIPLC', intersectedMapping)
UNCOMFORTABLE INCREASES _ISA__OINT
```

На этот раз часть шифротекста осталась не дешифрованной. Шифробуквы, для которых не нашлось варианта дешифрования, были заменены символами подчеркивания.

В этом примере текст, используемый для демонстрации взлома, был очень коротким. Обычно длина дешифруемых текстов намного больше.

(Шифротекст был специально подобран таким образом, чтобы его можно было взломать. Как правило, столь короткие тексты не удается взломать с помощью шаблонов слов.) Для взлома более длинных шифротекстов необходимо создавать словари шифробукв по каждому шифрослову и находить их совместные пересечения. Именно с этой целью функция `hackSimpleSub()` вызывает другие функции, созданные в программе.

Вызов функции `main()`

Функция `main()` вызывается в строках 155 и 156 только в том случае, если файл *simpleSubHacker.py* был запущен как программа, а не импортируется в виде модуля другой программой.

```
155. if __name__ == '__main__':
156.     main()
```

Примечание

Рассмотренный метод взлома хорошо работает лишь в том случае, когда пробелы не шифруются. Если хотите затруднить (но не сделать невозможным) взлом зашифрованных сообщений, можете расширить символьный набор, чтобы наряду с буквами программа шифровала пробелы, числа и знаки препинания. Это усложняет процедуру взлома, именно поэтому в книге мы ограничиваемся только шифрованием букв.

Резюме

Программа *simpleSubHacker.py* оказалась на данный момент самой сложной в книге! Вы узнали о том, как использовать словари шифробукв для определения вариантов дешифрования каждой буквы шифротекста. Вы также узнали, как можно сузить количество возможных решений путем добавления слов-кандидатов, нахождения пересечений словарей и удаления достоверно установленных букв из списков других шифробукв. Вместо того чтобы пытаться взломать все 403 291 461 126 605 635 584 000 000 возможных ключей методом грубой силы, мы написали программу на Python, которая позволяет восстановить весь (или почти весь) оригинальный ключ простого подстановочного шифра.

Основным преимуществом простого подстановочного шифра является большое количество возможных ключей. К числу его недостатков следует отнести относительную легкость сравнения шифрослов со словами из файла словаря для выявления шаблонов дешифрования шифробукв. В главе 18

мы исследуем более мощный полиалфавитный подстановочный шифр, называемый шифром Виженера, который в течение нескольких столетий считался неподдающимся взлому.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Каким будет шаблон для слова "hello"?
2. Имеют ли слова "mammoth" и "goggles" общий шаблон?
3. Какое из слов — "Alleged", "efficiently" или "poodle" — является возможным вариантом дешифрования шифрослова "PYYACAO"?

18

ПРОГРАММИРОВАНИЕ ШИФРА ВИЖЕНЕРА

"Я считал тогда и считаю сейчас, что благотворное влияние общедоступных инструментов криптографии на нашу безопасность и свободу значительно перевешивает тот неизбежный ущерб, который связан с возможностью их использования преступными элементами и террористами".

Мэтт Блейз, AT&T Labs, сентябрь 2001 г.

Шифр Виженера был впервые описан итальянским криптографом Джовани Баттиста Белласо в 1553 году, однако получил имя французского дипломата Блеза де Виженера, одного из многих людей, которые повторно изобретали этот шифр в последующие годы. Шифр заработал репутацию "le chiffre indéchiffrable", что означает "шифр, не поддающийся разгадыванию", и оставался недоступным для взлома до тех пор, пока английский математик Чарльз Бэббидж не взломал его в XIX веке.

Поскольку шифр Виженера имеет слишком много возможных ключей, чтобы его можно было взломать методом грубой силы даже с использованием нашего модуля обнаружения английских слов, он представляет собой один из наиболее стойких шифров среди тех, которые обсуждались в книге. Он неуязвим даже для атаки с применением шаблонов слов, описанной в главе 17.

В этой главе...

- Вспомогательные ключи
- Создание строк путем присоединения и объединения списков

Использование многобуквенных ключей в шифре Виженера

В отличие от шифра Цезаря шифр Виженера образуется цепочкой подстановок. Поскольку в нем используется более одного набора подстановок, его называют *полиалфавитным подстановочным шифром*. Одолеть шифр Виженера при помощи одного только частотного анализа невозможно. Вместо числового ключа со значениями в интервале от 0 до 25, как в шифре Цезаря, в шифре Виженера применяется буквенный ключ.

Ключ шифра Виженера представляет собой последовательность букв (например, английское слово), разбиваемую на множество вспомогательных однобуквенных ключей (подключей), которыми шифруются буквы открытого (незашифрованного) текста. Например, если в качестве ключа шифра Виженера выбрано слово "PIZZA", то первым подключом является 'P', вторым — 'I', третьим и четвертым — 'Z', пятым — 'A'. Первым подключом шифруется первая буква слова открытого текста, вторым — вторая буква и т.д. Когда мы доходим до шестой буквы открытого текста, мы возвращаемся к первому подключу.

Процедура применения шифра Виженера аналогична использованию нескольких шифров Цезаря подряд (рис. 18.1). Вместо того чтобы шифровать весь текст одним шифром Цезаря, мы применяем различные шифры Цезаря к каждой букве исходного текста.

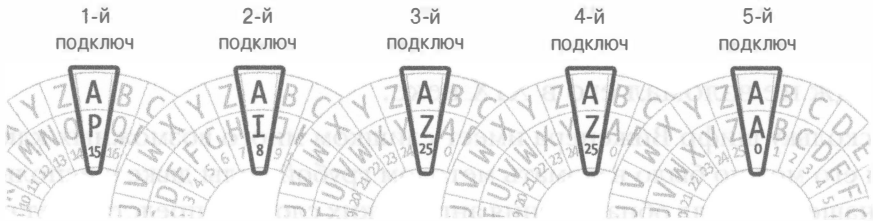


Рис. 18.1. *Шифр Виженера создается с помощью комбинации нескольких шифров Цезаря*

Каждый подключ преобразуется в целое число и служит ключом шифра Цезаря. Например, букве 'A' соответствует ключ 0 шифра Цезаря, букве 'B' — ключ 1 и т.д. вплоть до буквы 'Z', которой соответствует ключ 25 (рис. 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

Рис. 18.2. *Соответствие ключей и букв в шифре Цезаря*

Обратимся к примеру. Ниже приведено сообщение "COMMON SENSE IS NOT SO COMMON" вместе с ключом Виженера "PIZZA". Под каждой буквой исходного текста записан подключ, который ее шифрует.

```
COMMONSENSEISNOTSOCOMMON
PIZZAPIZZAPIZZAPIZZAPIZZ
```

Чтобы зашифровать первую букву исходного текста, 'C', используя подключ 'P', зашифруйте ее с помощью числового ключа шифра Цезаря, соответствующего данному подключу, в результате чего вы получите шифробукву 'R'. Повторяйте этот процесс для каждой буквы исходного текста, циклически перебирая подключи (табл. 18.1). Целые числа, указанные в скобках рядом с буквами исходного текста и подключей, суммируются попарно, и полученные целые числа соответствуют буквам шифротекста.

Таблица 18.1. Шифрование букв с помощью шифра Виженера

Буква открытого текста	Подключ	Буква шифротекста	Буква открытого текста	Подключ	Буква шифротекста
C (2)	P (15)	R (17)	S (18)	Z (25)	R (17)
O (14)	I (8)	W (22)	N (13)	Z (25)	M (12)
M (12)	Z (25)	L (11)	O (14)	A (0)	O (14)
M (12)	Z (25)	L (11)	T (19)	P (15)	I (8)
O (14)	A (0)	O (14)	S (18)	I (8)	A (0)
N (13)	P (15)	C (2)	O (14)	Z (25)	N (13)
S (18)	I (8)	A (0)	C (2)	Z (25)	B (1)
E (4)	Z (25)	D (3)	O (14)	A (0)	O (14)
N (13)	Z (25)	M (12)	M (12)	P (15)	B (1)
S (18)	A (0)	S (18)	M (12)	I (8)	U (20)
E (4)	P (15)	T (19)	O (14)	Z (25)	N (13)
I (8)	I (8)	Q (16)	N (13)	Z (25)	M (12)

Шифр Виженера с ключом "PIZZA" (состоящим из подключей 15, 8, 25, 25, 0) превращает открытый текст "COMMON SENSE IS NOT SO COMMON" в шифротекст "RWLLOC ADMST QR MOI ANBOBUNM".

Чем длиннее ключ шифра Виженера, тем он надежнее

Чем больше букв в ключе шифра Виженера, тем устойчивее зашифрованное сообщение к атакам методом грубой силы. Выбор слова "PIZZA" в качестве ключа шифра Виженера неудачен в силу того, что он содержит всего пять букв. Для ключа длиной пять букв существует 11 881 376 возможных комбинаций (количество букв, равное 26, будучи возведенным в пятую степень, дает $26^5 = 26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 = 11\,881\,376$ возможных вариантов). Конечно, одиннадцать миллионов ключей — слишком много для человека, пытающегося взломать шифр методом грубой силы, но для компьютера это дело лишь нескольких часов. Сначала он попытается дешифровать сообщение с помощью ключа "AAAAA" и проверит, получен ли в результате вразумительный текст на английском языке. Затем он попытается использовать ключи "AAAAB", "AAAAC" и т.д., пока не найдет ключ "PIZZA".

К счастью, с каждой дополнительной буквой количество возможных ключей увеличивается в 26 раз. В случае квадриллиона вариантов даже компьютеру потребуется затратить много лет на взлом шифра. Зависимость числа возможных ключей от длины ключа показана в табл. 18.2.

Таблица 18.2. Количество возможных ключей шифра Виженера в зависимости от длины ключа

Длина ключа	Уравнение	Количество возможных ключей
1	26	= 26
2	26 · 26	= 676
3	676 · 26	= 17 576
4	17 576 · 26	= 456 976
5	456 976 · 26	= 11 881 376
6	11 881 376 · 26	= 308 915 776
7	308 915 776 · 26	= 8 031 810 176
8	8 031 810 176 · 26	= 208 827 064 576
9	208 827 064 576 · 26	= 5 429 503 678 976
10	5 429 503 678 976 · 26	= 141 167 095 653 376
11	141 167 095 653 376 · 26	= 3 670 344 486 987 776
12	3 670 344 486 987 776 · 26	= 95 428 956 661 682 176
13	95 428 956 661 682 176 · 26	= 2 481 152 873 203 736 576
14	2 481 152 873 203 736 576 · 26	= 64 509 974 703 297 150 976

В случае ключей длиной двенадцать и более букв взлом шифра за разумное время с помощью обычного ноутбука становится невозможным.

Выбор ключа, предотвращающего словарные атаки

В качестве ключа шифра Виженера не обязательно выбирать слова наподобие "PIZZA". Им может служить любая комбинация букв произвольной длины, например 12-буквенный ключ "DURIWKNMFICK". В реальности лучше отказаться от использования слов, которые можно найти в словаре. Несмотря на то что слово "RADIOLOGISTS" также насчитывает 12 букв и его легче запомнить, чем слово "DURIWKNMFICK", криптоаналитик может догадаться, что шифровальщик использует реальное английское слово в качестве ключа.

Попытка взлома шифра методом грубой силы путем полного перебора всех слов английского языка, включенных в словарь, называется *переборoм по словарю*, или *словарной атакой* (dictionary attack). Всего существует 95 428 956 661 682 176 возможных 12-буквенных ключей, но наш файл словаря включает лишь около 1800 двенадцатибуквенных слов. Если мы используем в качестве ключа 12-буквенное слово, взятое из словаря, то взломать его путем перебора по словарю будет легче, чем ключ из трех случайных букв (поскольку таких ключей насчитывается 17 576).

Разумеется, шифровальщик обладает тем преимуществом, что криптоаналитику неизвестна точная длина ключа шифра Виженера. Однако криптоаналитик может испытать сначала все однобуквенные ключи, затем двухбуквенные, трехбуквенные и т.д., что в конечном счете позволит ему достаточно быстро определить ключ, который встречается в словаре.

Исходный код программы `Vigenere Cipher`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *vigenereCipher.py*. Убедитесь в том, что *pyperclip.py* находится в той же самой папке. Запустите программу, нажав клавишу <F5>.

vigenereCipher.py

```
1. # Шифр Виженера
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
5.
6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
7.
```

```
 8. def main():
 9.     # Этот текст можно скопировать из файла примера (см. введение)
10.     myMessage = """Alan Mathison Turing was a British mathematician,
          logician, cryptanalyst, and computer scientist."""
11.     myKey = 'ASIMOV'
12.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
13.
14.     if myMode == 'encrypt':
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
24.
25.
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
32.
33.
34. def translateMessage(key, message, mode):
35.     translated = []  # хранит зашифрованное/дешифрованное сообщение
36.
37.     keyIndex = 0
38.     key = key.upper()
39.
40.     for symbol in message:  # цикл по символам строки message
41.         num = LETTERS.find(symbol.upper())
42.         if num != -1:  # symbol.upper() найден в строке LETTERS
43.             if mode == 'encrypt':
44.                 num += LETTERS.find(key[keyIndex])  # добавить в случае
                                                          шифровании
45.             elif mode == 'decrypt':
46.                num -= LETTERS.find(key[keyIndex])  # вычесть в случае
                                                        дешифровании
47.
48.             num %= len(LETTERS)  # обработка "завертывания"
49.
50.             # Добавить зашифрованный/дешифрованный символ в конец
51.             if symbol.isupper():
52.                 translated.append(LETTERS[num])
53.             elif symbol.islower():
```

```
54.                 translated.append(LETTERS[num].lower())
55.
56.             keyIndex += 1  # перейти к следующей букве ключа
57.             if keyIndex == len(key):
58.                 keyIndex = 0
59.         else:
60.             # Присоединить символ без шифрования/дешифрования
61.             translated.append(symbol)
62.
63.     return ''.join(translated)
64.
65.
66. # Если файл vigenereCipher.py выполняется как программа
67. # (а не импортируется как модуль), вызвать функцию main()
68. if __name__ == '__main__':
69.     main()
```

Пример выполнения программы `Vigenere Cipher`

Выполнив программу, вы должны получить следующий результат.

```
Encrypted message:
Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi, lgouqdaf, kdmktsvmztsl,
izr xoexghzr kkusitaaf.

The message has been copied to the clipboard.
```

Программа отображает зашифрованное сообщение и копирует шифротекст в буфер обмена.

Импорт модулей, настройка констант и функция `main()`

Программа начинается со стандартных комментариев, описывающих ее назначение. С помощью инструкции `import` подключается модуль `pyperclip`, а в переменной `LETTERS` хранится строка, содержащая все буквы алфавита в верхнем регистре. Функция `main()` имеет примерно ту же структуру, что и в остальных программах. Она начинается с определения переменных для хранения сообщения, ключа и режима работы.

```
1. # Шифр Виженера
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
5.
```

```
 6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 7.
 8. def main():
 9.     # Этот текст можно скопировать из файла примера (см. введение)
10.     myMessage = """Alan Mathison Turing was a British mathematician,
          logician, cryptanalyst, and computer scientist."""
11.     myKey = 'ASIMOV'
12.     myMode = 'encrypt'  # задать 'encrypt' или 'decrypt'
13.
14.     if myMode == 'encrypt':
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
```

Зашифрованное или дешифрованное сообщение (в зависимости от того, какое значение присвоено переменной `myMode`) сохраняется в переменной `translated`, которая выводится на экран (строка 20) и копируется в буфер обмена (строка 21).

Создание строк с помощью списковых методов `append()` и `join()`

Почти во всех программах, рассмотренных в книге, нам приходилось создавать строки программным путем. Сначала объявлялась переменная, содержащая пустую строку, а затем в нее добавлялись символы путем конкатенации. Введите в интерактивной оболочке следующий код.

```
>>> building = ''
>>> for c in 'Hello world!':
...     building += c
...
>>> print(building)
```

В данном случае в цикле перебираются все символы строки `'Hello world!'`, которые конкатенируются со строкой, хранящейся в переменной `building`. По завершении цикла переменная `building` будет содержать всю исходную строку.

Несмотря на кажущуюся простоту, данная методика реализуется в Python слишком неэффективно. Намного быстрее начать с пустого списка и при-

менять к нему метод append(). Когда список будет построен, его можно будет преобразовать в строку помощью метода join(). Приведенный ниже код дает тот же результат, что и в предыдущем примере, но гораздо быстрее.

```
>>> building = []
>>> for c in 'Hello world!':
...     building.append(c)
...
>>> building = ''.join(building)
>>> print(building)
```

Применение списков вместо строк позволяет значительно ускорить работу программы. Разницу в быстродействии можно замерить с помощью функции time.time(). Откройте в редакторе файлов новое окно и введите в нем следующий код.

stringTest.py

```
import time

startTime = time.time()
for trial in range(10000):
    building = ''
    for i in range(10000):
        building += 'x'
print('String concatenation:  ', (time.time() - startTime))

startTime = time.time()
for trial in range(10000):
    building = []
    for i in range(10000):
        building.append('x')
    building = ''.join(building)
print('List appending:        ', (time.time() - startTime))
```

Сохраните программу в файле *stringTest.py* и запустите ее. Результаты будут примерно такими.

```
String concatenation:  40.317070960998535
List appending:        10.488219022750854
```

В переменную startTime записывается текущее время, после чего выполняется цикл, в котором к строке присоединяется 10 000 символов путем конкатенации. По завершении цикла программа сообщает общее время, затраченное на эту операцию. Далее в переменную startTime вновь

записывается текущее время, и выполняется цикл, в котором создается строка той же длины, но путем присоединения элементов списка. В конце программа тоже сообщает время, затраченное на эту операцию. На компьютере автора на создание 10 000 строк длиной 10 000 символов каждая ушло примерно 40 секунд, тогда как на выполнение той же задачи с помощью списков — всего 10 секунд. Таким образом, если в программе создается большое количество строк, использование списков позволит существенно ускорить ее работу.

В дальнейшем мы будем создавать строки в программах именно с помощью списковых методов.

Шифрование и дешифрование сообщения

Поскольку код шифрования и дешифрования почти одинаков, мы создадим две функции-обертки, `encryptMessage()` и `decryptMessage()`, которые будут вызывать функцию `translateMessage()`, содержащую сам код.

```
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
```

Функция `translateMessage()` создает зашифрованную (дешифрованную) строку символ за символом, сохраняя их в списке `translated`, содержимое которого может быть объединено, когда строка будет готова.

```
34. def translateMessage(key, message, mode):
35.     translated = []  # хранит зашифрованное/дешифрованное сообщение
36.
37.     keyIndex = 0
38.     key = key.upper()
```

Не забывайте о том, что шифр Виженера — это уже знакомый вам шифр Цезаря, за исключением того, что ключ, применяемый к букве, зависит от ее позиции в сообщении. Начальным значением переменной `keyIndex`, с помощью которой отслеживается текущий подключ, является `0`, поскольку для шифрования/дешифрования первого символа сообщения используется буква `key[0]`.

В программе предполагается, что ключ содержит только буквы в верхнем регистре. Чтобы гарантировать допустимость ключа, в строке 38 к строке ключа применяется метод `upper()`.

Остальной код функции `translateMessage()` напоминает код, который применялся для реализации шифра Цезаря.

```
40.     for symbol in message:  # цикл по символам строки message
41.         num = LETTERS.find(symbol.upper())
42.         if num != -1:  # symbol.upper() найден в строке LETTERS
43.             if mode == 'encrypt':
44.                 num += LETTERS.find(key[keyIndex])  # добавить в случае
                                                          шифрования
45.             elif mode == 'decrypt':
46.                num -= LETTERS.find(key[keyIndex])  # вычесть в случае
                                                         дешифрования
```

В цикле `for`, который начинается в строке 40, символы сообщения присваиваются переменной `symbol` на каждой итерации цикла. В строке 41 определяется индекс текущего символа, переведенного в верхний регистр, в строке `LETTERS`, в результате чего буква преобразуется в число.

Если значение `num` не равно `-1`, то это означает, что символ в верхнем регистре был найден в строке `LETTERS` (т.е. данный символ является буквой). Переменная `keyIndex` позволяет отслеживать текущий подключ, который является результатом вычисления выражения `key[keyIndex]`.

Конечно, это всего лишь однобуквенная строка. Чтобы преобразовать подключ в число, нужно найти индекс данной буквы в строке `LETTERS`. Затем это число суммируется (в случае шифрования) с числовым кодом символа в строке 44 или вычитается (в случае дешифрования) из него в строке 46.

При работе с шифром Цезаря мы проверяли, не является ли новое значение `num` отрицательным (в этом случае мы прибавляли к нему число `len(LETTERS)`) или же превышающим значение `len(LETTERS)` (в таком случае мы вычитали из него число `len(LETTERS)`). Это позволяет учесть случаи "завертывания".

Однако существует более простой способ. Те же самые вычисления можно уместить в одной строке кода, разделив значение `num` на `len(LETTERS)` с остатком:

```
48.             num %= len(LETTERS)  # обработка "завертывания"
```

Например, если бы значение `num` было равно `-8`, то к нему нужно было бы прибавить `26` (т.е. значение `len(LETTERS)`), чтобы получить `18`, но это же число является результатом операции `-8 % 26`. Или же, если бы значение `num` было равно `31`, то из него требовалось бы вычесть `26`, чтобы получить `5`, но результатом операции `31 % 26` тоже является число `5`.

Операция деления с остатком в строке 48 справляется с обоими случаями "завертывания".

Зашифрованным (или дешифрованным) символом является буква `LETTERS[num]`. Но мы хотим, чтобы регистр полученного символа совпадал с регистром исходного символа.

```
50.             # Добавить зашифрованный/дешифрованный символ в конец
51.             if symbol.isupper():
52.                 translated.append(LETTERS[num])
53.             elif symbol.islower():
54.                 translated.append(LETTERS[num].lower())
```

Поэтому, если переменная `symbol` содержит букву в верхнем регистре, условие в строке 51 выполняется, и тогда в строке 52 символ `LETTERS[num]` присоединяется к списку `translated`, поскольку все символы в строке `LETTERS` уже переведены в верхний регистр.

Если же в переменной `symbol` хранится буква в нижнем регистре, то выполняется условие в строке 53, и тогда в строке 54 к списку `translated` присоединяется буква `LETTERS[num]`, преобразованная ее в нижний регистр. Тем самым мы обеспечиваем согласование регистра букв в исходном и зашифрованном (дешифрованном) сообщениях.

После преобразования символа мы хотим убедиться в том, что на следующей итерации цикла `for` будет выбран очередной подключ. Для этого в строке 56 значение `keyIndex` увеличивается на единицу.

```
56.             keyIndex += 1  # перейти к следующей букве ключа
57.             if keyIndex == len(key):
58.                 keyIndex = 0
```

Но если в данный момент уже задействован последний из подключей, то значение `keyIndex` станет равно длине ключа. Это условие проверяется в строке 57, и если оно соблюдается, то в строке 58 значение `keyIndex` сбрасывается в `0`, чтобы выражение `key[keyIndex]` вновь указывало на первый подключ.

Выделение отступом инструкции `else` в строке 59 указывает на то, что она связана с инструкцией `if` в строке 42.

```
59.         else:
60.             # Присоединить символ без шифрования/дешифрования
61.             translated.append(symbol)
```

Код в строке 61 выполняется в том случае, если символ не был обнаружен в строке LETTERS. Это происходит тогда, когда символом, хранящимся в переменной symbol, является число или знак препинания, например '5' или '?'. В таком случае в строке 61 символ присоединяется к списку translated в неизменном виде.

Завершив формирование списка translated, мы вызываем метод join() для пустой строки, чтобы получить в результате объединенную строку:

```
63.     return ''.join(translated)
```

Функция возвращает полную строку зашифрованного или дешифрованного сообщения.

Вызов функции main()

Программа завершается строками 68 и 69.

```
68. if __name__ == '__main__':
69.     main()
```

Функция main() выполняется лишь в том случае, если программа была запущена непосредственно, а не импортируется в виде модуля другой программой, которой требуются ее функции encryptMessage() и decryptMessage().

Резюме

Шифр Виженера не намного сложнее шифра Цезаря, одного из первых шифров, с которым вы познакомились. Внеся всего несколько изменений в шифр Цезаря, мы создали шифр, количество возможных ключей которого делает невозможным взлом методом грубой силы.

Шифр Виженера неуязвим для атаки с использованием словарных шаблонов, которая применялась в нашей программе взлома простого подстановочного шифра. В течение сотен лет "неподдающийся" шифр Виженера оберегал секреты конфиденциальных сообщений, но в конечном счете и он не устоял. В главах 19 и 20 вы познакомитесь с методами частотного анализа, используя которые можно взломать шифр Виженера.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Какому шифру аналогичен шифр Виженера, если не считать того, что в нем используется несколько ключей вместо одного?
2. Сколько возможных вариантов существует для ключа шифра Виженера длиной 10 символов?
   А. Сотни.
   Б. Тысячи.
   В. Миллионы.
   Г. Более триллиона.
3. К какому типу шифров относится шифр Виженера?

19

ЧАСТОТНЫЙ АНАЛИЗ

"Умение интуитивно находить закономерности в хаосе бесполезно без погружения в сам хаос. Если там действительно есть закономерности, сознание их пока не видит. Но теперь, когда буквы прошли перед глазами и были выписаны на бумаге, в работу включается подсознание..."

Нил Стивенсон, "Криптономикон"

В этой главе вы узнаете о том, как определить частотность каждой буквы английского алфавита в тексте. Затем вы сравните эти данные с частотностью букв в шифротексте, что даст важную информацию для взлома шифра. Процесс определения того, насколько часто буквы появляются в открытых текстах и шифротекстах, называется *частотный анализ*. Изучение частотного анализа является важным шагом на пути к взлому шифра Виженера, чем мы займемся в главе 20.

В этой главе...

- Частотность букв и буквенный ряд ETAOIN
- Аргументы `key` и `reverse` метода `sort()`
- Передача функций как значений вместо вызова функций
- Преобразование словарей в списки с помощью методов `keys()`, `values()` и `items()`

Анализ частотности букв в тексте

Если подбрасывать монету множество раз, то примерно в половине случаев выпадет "орел", а в половине — "решка". Таким образом, частота выпадения "орла" и "решки" примерно одинакова. Эту частоту можно представить в процентном отношении, разделив количество наступивших событий (например, сколько раз выпал "орел") на количество попыток (сколько раз мы подбросили монету) и умножив результат на 100. Мы многое можем узнать о монете по частоте выпадения "орла" и "решки": правильно ли сбалансированы ее стороны и не является ли она поддельной, например с "орлом" на обеих сторонах.

Точно так же мы можем многое узнать о шифротексте, изучая частотность встречающихся в нем букв. Одни буквы английского алфавита встречаются чаще других. Например, в английских словах чаще всего встречаются буквы 'E', 'T', 'A' и 'O', тогда как буквы 'J', 'X', 'Q' и 'Z' встречаются реже. Эти различия в частотности мы используем для взлома сообщений, зашифрованных с помощью шифра Виженера.

На рис. 19.1 в графическом виде представлены данные о частотности букв в английском языке. Эта диаграмма была построена на основе текстов, взятых из книг, газет и других источников.

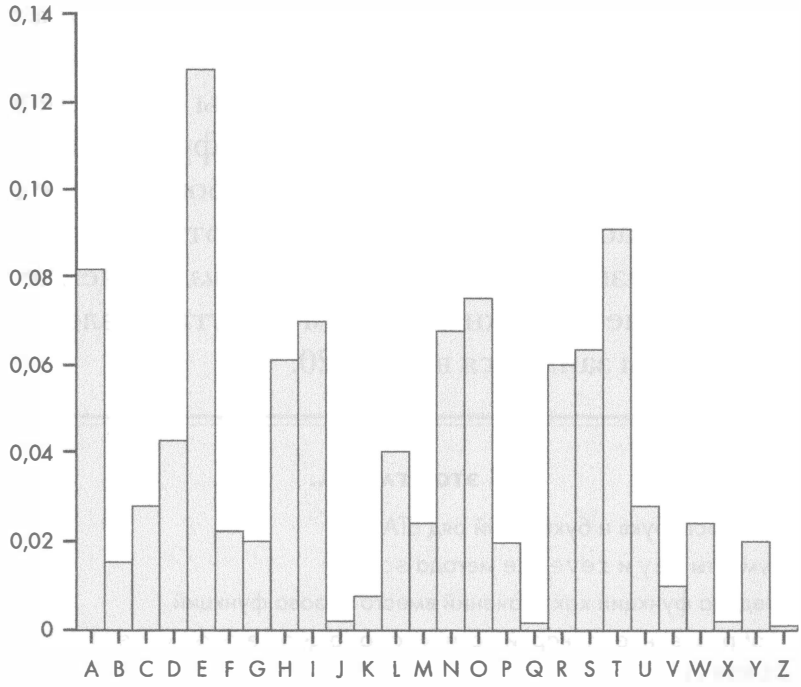


Рис. 19.1. *Частотность букв в типичных текстах на английском языке*

Расположив буквы в порядке убывания частотности, мы увидим, что чаще всего встречается буква 'E', за ней следует буква 'T', затем буква 'A' и т.д. (рис. 19.2).

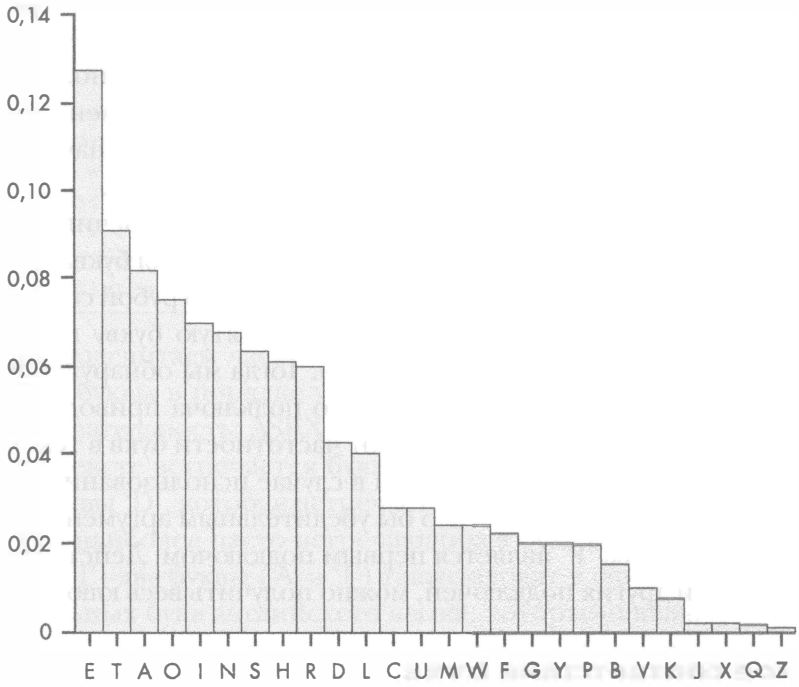


Рис. 19.2. *Буквы, расположенные в порядке убывания их частотности в типичных текстах на английском языке*

Шесть наиболее часто встречающихся букв в английском языке — ETAOIN. А вот полный список букв, расположенных в порядке убывания их частотности: ETAOINSHRDLCUMWFGYPBVKJXQZ.

Вспомните, что в перестановочном шифре сообщения шифруются путем расположения букв исходного сообщения в другом порядке. Это означает, что частотность букв в шифротексте не будет отличаться от их частотности в исходном тексте. Например, в шифротексте перестановочного шифра буквы 'E', 'T' и 'A' должны встречаться чаще, чем буквы 'X', 'Q' и 'Z'.

Точно так же буквы, которые чаще всего встречаются в шифротекстах, зашифрованных с помощью шифра Цезаря или простого подстановочного шифра, скорее всего, были получены шифрованием таких наиболее часто используемых букв, как 'E', 'T' или 'A'. В то же время буквы, которые встречаются в шифротексте намного реже других, скорее всего, были получены шифрованием букв 'X', 'Q' и 'Z' исходного текста.

Чрезвычайная полезность частотного анализа для взлома шифра Виженера объясняется тем, что он позволяет применять метод грубой силы для взлома по одному подключу за раз. Например, если сообщение было зашифровано с помощью ключа "PIZZA", то для нахождения сразу всего ключа нам потребовался бы полный перебор 26^5, или 11 881 376 ключей. Но для взлома методом грубой силы только одного из пяти подключей нам нужно испробовать лишь 26 возможных вариантов. Применение аналогичной процедуры в отношении каждого из пяти подключей означает, что нам придется испытать всего-навсего $26 \cdot 5$, или 130 подключей.

В случае ключа "PIZZA" каждая пятая буква сообщения, начиная с первой, будет шифроваться с помощью подключа 'P', каждая пятая буква, начиная со второй, – буквой 'I' и т.д. Мы можем применить метод грубой силы для определения первого подключа, дешифруя каждую пятую букву шифротекста при помощи всех 26 возможных подключей. Тогда мы обнаружили бы, что использование буквы 'P' в качестве первого подключа приводит к дешифрованным буквам, которые соответствуют частотности букв в текстах на английском языке в большей степени, чем в случае использования остальных 25 возможных подключей. Это стало бы убедительным аргументом в пользу того, что именно буква 'P' является первым подключом. Действуя в том же духе в отношении других подключей, можно получить весь ключ.

Частотное соответствие букв

Для нахождения частотности букв в сообщении мы применим алгоритм, который сортирует буквы в строке в порядке убывания их частотности. С помощью этой упорядоченной строки будет вычисляться величина, которая в данной книге называется *оценкой частотного соответствия* (frequency match score). Она показывает, насколько частотность букв в строке совпадает с частотностью букв в английском языке.

Чтобы рассчитать оценку частотного соответствия, мы присваиваем ей нулевое начальное значение, а затем увеличиваем ее на единицу всякий раз, когда одна из наиболее употребительных букв английского алфавита ('E', 'T', 'A', 'O', 'I', 'N') появляется среди шести наиболее часто встречающихся букв в шифротексте. Мы также добавляем к оценке единицу всякий раз, когда одна из наименее употребительных букв английского алфавита ('V', 'K', 'J', 'X', 'Q', 'Z') появляется среди шести наименее часто встречающихся букв в шифротексте.

Оценка частотного соответствия для строки может иметь значения от 0 (абсолютное несоответствие) до 12 (полное соответствие). Знание данной оценки дает важную информацию о характере исходного текста.

Вычисление оценки частотного соответствия букв для простого подстановочного шифра

Для вычисления оценки частотного соответствия букв в сообщении, зашифрованном с помощью простого подстановочного шифра, мы будем использовать следующий шифротекст.

```
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, ia 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. - acjclxo Ctrramm
```

Если мы подсчитаем частотность каждой буквы в этом шифротексте и расположим буквы в порядке убывания частотности, то получим следующую последовательность: ASRXJILPWMCYOUEQNTHBFZGKVD. Чаще всего в шифротексте встречается буква 'A', на втором месте — буква 'S' и т.д. вплоть до буквы 'D', которая встречается реже всех остальных букв.

Из шести наиболее часто встречающихся букв в этом примере ('A', 'S', 'R', 'X', 'J' и 'I') две буквы ('A' и 'I') также входят в список шести наиболее употребительных букв английского языка, которыми являются 'E', 'T', 'A', 'O', 'I' и 'N'. Соответственно мы добавляем к оценке частотного соответствия два балла.

Шестью наименее часто встречающимися буквами в шифротексте являются буквы 'F', 'Z', 'G', 'K', 'V' и 'D'. Три из них ('Z', 'K' и 'V') появляются в списке наименее часто встречающихся букв английского языка ('V', 'K', 'J', 'X', 'Q' и 'Z'). Поэтому мы увеличиваем оценку еще на три балла. На основании упорядоченности букв шифротекста по частотности (ASRXJILPWMCYOUEQNTHBFZGKVD) мы получаем суммарную оценку частотного соответствия, равную 5 (рис. 19.3).

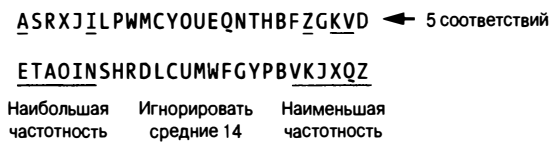


Рис. 19.3. *Вычисление оценки частотного соответствия букв для простого подстановочного шифра*

Для шифротекста, полученного с помощью простого подстановочного шифра, оценка частотного соответствия не будет очень высокой по той причине, что частотность букв в этом шифре не согласуется с частотностью

букв в типичных текстах на английском языке, так как буквы открытого текста поочередно заменяются шифробуквами. Например, если буква 'T' в результате шифрования заменяется буквой 'J', то мы можем ожидать частого появления буквы 'J' в шифротексте, несмотря на то что она относится к числу наименее часто встречающихся букв в английском языке.

Вычисление оценки частотного соответствия букв для перестановочного шифра

Давайте вычислим оценку частотного соответствия для букв шифротекста, полученного с помощью перестановочного шифра.

```
"I rc ascwuiluhnviwuetnh,osgaa ice tipeeeee slnatsfietgi tittynecenisl. e
fo f fnc isltn sn o a yrs sd onisli ,l erglei trhfmwfrogotn,l stcofiit.
aea  wesn,lnc ee w,l eIh eeehoer ros  iol er snh nl oahsts  ilasvih  tvfeh
rtira id thatnie.im ei-dlmf i  thszonsisehroe, aiehcdsanahiec  gv gyedsB
affcahiecesd d lee   onsdihsoc nin cethiTitx  eRneahgin r e teom fbiotd  n
ntacscwevhtdhnhpiwru"
```

Ряд букв, расположенных в порядке убывания частотности, в данном шифротексте выглядит так: EISNTHAOCLRFDGWVMUYBPZXQJK. Буква 'E' встречается наиболее часто, на втором месте — буква 'I' и т.д.

Четыре буквы, которые встречаются в этом шифротексте чаще других ('E', 'I', 'N' и 'T'), также входят в список букв, наиболее часто встречающихся в типичных текстах на английском языке (ETAOIN). Точно так же пять наименее часто встречающихся букв в шифротексте ('Z', 'X', 'Q', 'J' и 'K') одновременно входят в список VKJXQZ, в результате чего суммарная оценка частотного соответствия достигает значения 9 (рис. 19.4).

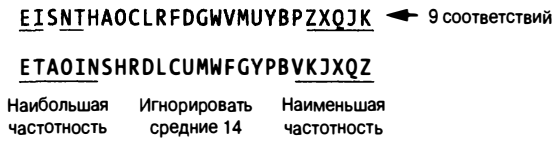


Рис. 19.4. *Вычисление оценки частотного соответствия букв для перестановочного шифра*

Шифротекст, полученный с помощью перестановочного шифра, должен иметь намного более высокую оценку частотного соответствия по сравнению с простым перестановочным шифром. Причина заключается в том, что в перестановочном шифре, в отличие от простого подстановочного шифра, используются те же буквы, что и в исходном незашифрованном тексте, но расположенные в другом порядке. Поэтому частотность каждой буквы остается одной и той же.

Использование частотного анализа в случае шифра Виженера

Пытаясь взломать шифр Виженера, мы должны дешифровать каждый из подключей по отдельности. Это означает, что мы не можем полагаться на метод обнаружения английских слов, поскольку не в состоянии дешифровать достаточную часть сообщения с помощью только одного подключа.

Вместо этого мы дешифруем буквы, зашифрованные с помощью одного подключа, и выполняем частотный анализ, чтобы определить, какой из вариантов дешифрования приводит к частотности, наиболее близкой к частотности букв в типичных текстах на английском языке. Другими словами, мы должны выяснить, какой из вариантов дешифрования характеризуется наиболее высокой оценкой частотного соответствия букв. Это служит хорошим показателем того, что мы нашли правильный подключ.

Описанный процесс повторяется для второго, третьего, четвертого и пятого подключей. На данном этапе предположение о том, что длина ключа составляет пять букв, является всего лишь догадкой. (В главе 20 вы узнаете об использовании метода Касиски для определения длины ключа.) Поскольку в шифре Виженера для каждого подключа существует 26 вариантов дешифрования (число букв в алфавите), то в случае пятибуквенного ключа компьютер должен перепробовать 26 + 26 + 26 + 26 + 26, т.е. 156 вариантов дешифрования. Это намного меньше, чем если бы испытывались все возможные комбинации подключей, общее количество которых составляет 11 881 376 ($26 \cdot 26 \cdot 26 \cdot 26 \cdot 26$)!

Взлом шифра Виженера включает ряд дополнительных этапов, рассмотренных в главе 20, в которой мы напишем собственно программу взлома. А пока что мы создадим модуль частотного анализа, включающий следующие вспомогательные функции.

- **`getLetterCount()`.** Получает строковый аргумент и возвращает словарь, содержащий счетчики частотности каждой буквы в строке.
- **`getFrequencyOrder()`.** Получает строковый аргумент и возвращает строку из 26 букв алфавита, расположенных в порядке убывания частотности.
- **`englishFreqMatchScore()`.** Получает строковый аргумент и возвращает целое число в интервале от 0 до 12, определяющее оценку частотного соответствия букв.

Исходный код программы `Frequency Analysis`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *freqAnalysis.py*. Запустите программу, нажав клавишу <F5>.

freqAnalysis.py

```
 1. # Определение частотности букв
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'
 5. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 6.
 7. def getLetterCount(message):
 8.     # Возвращает словарь, ключами которого являются буквы,
 9.     # а значениями - частотность каждой буквы в строке message
10.     letterCount = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0,
          'G': 0, 'H': 0, 'I': 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}
11.
12.     for letter in message.upper():
13.         if letter in LETTERS:
14.             letterCount[letter] += 1
15.
16.     return letterCount
17.
18.
19. def getItemAtIndexZero(items):
20.     return items[0]
21.
22.
23. def getFrequencyOrder(message):
24.     # Возвращает строку букв алфавита, расположенных в порядке
25.     # убывания их частотности в строке message.
26.
27.     # Во-первых, получаем словарь частотности букв
28.     letterToFreq = getLetterCount(message)
29.
30.     # Во-вторых, создаем словарь счетчиков частотности
31.     # со списком букв по каждому счетчику
32.     freqToLetter = {}
33.     for letter in LETTERS:
34.         if letterToFreq[letter] not in freqToLetter:
35.             freqToLetter[letterToFreq[letter]] = [letter]
36.         else:
37.             freqToLetter[letterToFreq[letter]].append(letter)
38.
39.     # В-третьих, изменяем порядок букв в каждом списке на
```

```
    # обратный порядку "ETAOIN" и превращаем списки в строки
    for freq in freqToLetter:
        freqToLetter[freq].sort(key=ETAOIN.find, reverse=True)
        freqToLetter[freq] = ''.join(freqToLetter[freq])

    # В-четвертых, преобразуем словарь freqToLetter в список
    # кортежей (ключ, значение) и сортируем его
    freqPairs = list(freqToLetter.items())
    freqPairs.sort(key=getItemAtIndexZero, reverse=True)

    # В-пятых, после того как буквы были упорядочены по частотности,
    # извлекаем все буквы для формирования окончательной строки
    freqOrder = []
    for freqPair in freqPairs:
        freqOrder.append(freqPair[1])

    return ''.join(freqOrder)


def englishFreqMatchScore(message):
    # Возвращает оценку частотного соответствия для строки
    # message. Совпадения проверяются по шести наиболее
    # и наименее часто встречающимся буквам в строке
    # и в английском языке в целом.

    freqOrder = getFrequencyOrder(message)

    matchScore = 0
    # Число совпадений для шести наиболее часто встречающихся букв
    for commonLetter in ETAOIN[:6]:
        if commonLetter in freqOrder[:6]:
            matchScore += 1
    # Число совпадений для шести наименее часто встречающихся букв
    for uncommonLetter in ETAOIN[-6:]:
        if uncommonLetter in freqOrder[-6:]:
            matchScore += 1

    return matchScore
```

Сохранение букв алфавита в порядке ETAOIN

В строке 4 создается переменная ETAOIN, в которой сохраняются 26 букв алфавита, расположенных в порядке убывания их частотности.

```
1. # Определение частотности букв
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'
```

Разумеется, не все тексты на английском языке отражают именно такое частотное распределение. Не составит труда найти книгу, в которой, скажем, буква 'Z' встречается чаще буквы 'Q'. Например, в романе "Гадсби" американского писателя Эрнеста Винсента Райта буква 'E' вообще не используется, что приводит к необычному распределению частотности букв. Однако в большинстве случаев, включая и наш модуль, порядок ETAOIN будет соблюдаться достаточно точно.

Кроме того, для некоторых функций, входящих в состав нашего модуля, понадобится также строка, которая содержит все буквы в верхнем регистре, расположенные в алфавитном порядке. Для этого в строке 5 создается константа `LETTERS`.

```
5. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Константа `LETTERS` служит той же цели, что и константа `SYMBOLS` в предыдущих программах: она позволяет установить соответствие между буквами и целочисленными индексами.

Теперь рассмотрим, как функция `getLettersCount()` подсчитывает частотность каждой буквы в заданной строке.

Подсчет букв в сообщении

Функция `getLetterCount()` получает строку `message` и возвращает словарь, ключами которого являются буквы, а значениями — количество появлений каждой буквы в строке.

В строке 10 создается словарь `letterCount`, для всех ключей которого установлены нулевые начальные значения.

```
 7. def getLetterCount(message):
 8.     # Возвращает словарь, ключами которого являются буквы,
 9.     # а значениями - число появлений каждой буквы в строке message
10.     letterCount = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0,
          'G': 0, 'H': 0, 'I': 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}
```

Проверяя каждый символ сообщения в цикле `for` (строка 12), мы инкрементируем значения ключей до тех пор, пока они не будут представлять частотность каждой буквы.

```
12.     for letter in message.upper():
13.         if letter in LETTERS:
14.             letterCount[letter] += 1
```

В цикле `for` мы проходим по всем символам строки `message`, преобразованным в верхний регистр. Текущий символ записывается в переменную `letter`. В строке 13 мы проверяем, содержится ли символ в строке `LETTERS`, поскольку счетчики небуквенных символов нас не интересуют. Если буква встречается в строке `LETTERS`, в строке 14 инкрементируется значение `letterCount[letter]`.

По завершении цикла `for` функция `getLetterCount()` возвращает словарь `letterCount`, содержащий счетчики появлений каждой буквы в строке `message`.

```
16.     return letterCount
```

В этой главе мы будем использовать следующую строку (`https://en.wikipedia.org/wiki/Alan_Turing`).

```
"""Alan Mathison Turing was a British mathematician, logician,
cryptanalyst, and computer scientist. He was highly influential in
the development of computer science, providing a formalisation of the
concepts of "algorithm" and "computation" with the Turing machine.
Turing is widely considered to be the father of computer science and
artificial intelligence. During World War II, Turing worked for the
Government Code and Cypher School (GCCS) at Bletchley Park, Britain's
codebreaking centre. For a time he was head of Hut 8, the section
responsible for German naval cryptanalysis. He devised a number of
techniques for breaking German ciphers, including the method of the
bombe, an electromechanical machine that could find settings for the
Enigma machine. After the war he worked at the National Physical
Laboratory, where he created one of the first designs for a stored-
program computer, the ACE. In 1948 Turing joined Max Newman's Computing
Laboratory at Manchester University, where he assisted in the development
of the Manchester computers and became interested in mathematical
biology. He wrote a paper on the chemical basis of morphogenesis, and
predicted oscillating chemical reactions such as the Belousov-Zhabotinsky
reaction, which were first observed in the 1960s. Turing's homosexuality
resulted in a criminal prosecution in 1952, when homosexual acts were
still illegal in the United Kingdom. He accepted treatment with female
hormones (chemical castration) as an alternative to prison. Turing died
in 1954, just over two weeks before his 42nd birthday, from cyanide
poisoning. An inquest determined that his death was suicide; his mother
and some others believed his death was accidental. On 10 September 2009,
following an Internet campaign, British Prime Minister Gordon Brown made
an official public apology on behalf of the British government for "the
appalling way he was treated." As of May 2012 a private member's bill was
before the House of Lords which would grant Turing a statutory pardon if
enacted."""
```

Для этой строки, содержащей 135 вхождений буквы 'A', 30 вхождений буквы 'B' и т.д., функция `getLetterCount()` возвращает следующий словарь.

```
{'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}
```

Получение первого элемента кортежа

Функция `getItemAtIndexZero()` в строке 19 возвращает элемент кортежа с индексом 0.

```
19. def getItemAtIndexZero(items):
20.     return items[0]
```

Эта функция будет передаваться методу `sort()` для сортировки списка букв по частотности. Более подробно этот вопрос будет рассмотрен в разделе "Преобразование элементов словаря в сортируемый список".

Упорядочение букв, встречающихся в сообщении, в соответствии с частотностью

Функция `getFrequencyOrder()` получает строку `message` и возвращает строку, которая содержит 26 букв алфавита в верхнем регистре, расположенных в порядке убывания их частотности в сообщении. Если строка `message` представляет собой осмысленный текст, а не случайное скопление букв, то, вероятнее всего, она будет аналогична, если не идентична, строке, содержащейся в константе `ETAOIN`. Эта функция выполняет основную часть работы по вычислению оценки частотного соответствия, которую мы используем в программе взлома шифра Виженера в главе 20.

Например, если передать функции `getFrequencyOrder()` строку `"""Alan Mathison Turing..."""`, то она вернет строку `'ETIANORSHCLMDGFUPBWYVKXQJZ'`, поскольку в исходной строке чаще всего встречается буква 'E', второе место занимает буква 'T', затем 'I', 'A' и т.д.

Алгоритм работы функции `getFrequencyOrder()` можно представить в виде следующей последовательности действий:

1) подсчет числа букв в строке;
2) создание словаря счетчиков частотности со списком букв по каждому счетчику;

3) сортировка списков букв в порядке, обратном порядку ETAOIN;
4) преобразование этих данных в список кортежей;
5) преобразование списка кортежей в окончательную строку, возвращаемую функцией `getFrequencyOrder()`.

Рассмотрим каждый из этапов по отдельности.

Подсчет букв с помощью функции `getLetterCount()`

На первом этапе функция `getFrequencyOrder()` вызывает функцию `getLetterCount()` в строке 28, передавая ей параметр `message` для получения словаря `letterToFreq`, содержащего счетчики по каждой букве сообщения.

```
23. def getFrequencyOrder(message):
24.     # Возвращает строку букв алфавита, расположенных в порядке
25.     # убывания их частотности в параметре message
26.
27.     # Во-первых, получаем словарь частотности букв
28.     letterToFreq = getLetterCount(message)
```

Если передать этой функции строку `"""Alan Mathison Turing..."""`, то в строке 28 в переменную `letterToFreq` будет записан следующий словарь.

```
{'A': 135, 'C': 74, 'B': 30, 'E': 196, 'D': 58, 'G': 39, 'F': 37,
'I': 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}
```

Создание словаря счетчиков частотности со списками букв

На втором этапе функция `getFrequencyOrder()` создает словарь `freqToLetter`, ключами которого являются счетчики частотности, а значениями — списки букв, соответствующих счетчикам частотности. В то время как словарь `letterToFreq` сопоставляет буквенные ключи значениям частотности, словарь `freqToLetter` сопоставляет значения частотности спискам букв, поэтому необходимо поменять местами ключи и значения в словаре `letterToFreq`, так как несколько букв могут иметь одну и ту же частотность. В данном примере у букв `'B'` и `'W'` одинаковые счетчики (30), и мы создаем для них словарь наподобие `{30: ['B', 'W']}`, поскольку ключи словаря должны быть уникальными.

В строке 32 сначала создается пустой словарь.

```
30.     # Во-вторых, создаем словарь счетчиков частотности
31.     # со списком букв по каждому счетчику
32.     freqToLetter = {}
33.     for letter in LETTERS:
34.         if letterToFreq[letter] not in freqToLetter:
35.             freqToLetter[letterToFreq[letter]] = [letter]
36.         else:
37.             freqToLetter[letterToFreq[letter]].append(letter)
```

В строке 33 начинается цикл по всем буквам строки `LETTERS`. Инструкция `if` в строке 34 проверяет, содержится ли показатель частотности буквы, т.е. `letterToFreq[letter]`, в виде ключа в словаре `freqToLetter`. Если это не так, то в строке 35 ключ добавляется в словарь, а в качестве спискового значения задается буква. Если же ключ уже содержится в словаре `freqToLetter`, то соответствующая буква добавляется в конец списка.

Если взять в качестве примера словарь `letterToFreq`, созданный для строки `"""Alan Mathison Turing..."""`, то словарь `freqToLetter` будет выглядеть так.

```
{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',
'U'], 39: ['G'], 58: ['D', 'M'], 62: ['L'], 196: ['E'], 74: ['C'],
87: ['H'], 89: ['S'], 106: ['R'], 113: ['O'], 122: ['N']}
```

Обратите внимание на то, что теперь ключи словаря содержат счетчики частотности, а значения превратились в списки букв, имеющих данную частотность.

Сортировка списков букв в порядке, обратном порядку ETAOIN

Третий этап, выполняемый функцией `getFrequencyOrder()`, включает сортировку букв в каждом списке `freqToLetter`. Вспомните, что значением `freqToLetter[freq]` является *список* букв, соответствующих значению частотности `freq`. Мы используем список, поскольку одну и ту же частотность могут иметь несколько букв.

Когда у нескольких букв одинаковая частотность, мы сортируем их в порядке, обратном порядку их появления в строке `ETAOIN`. Это делает их сортировку согласованной и минимизирует вероятность случайного увеличения оценки частотного соответствия.

Предположим, например, что в оцениваемой строке у букв 'V', 'I', 'N' и 'K' одинаковая частотность. Также предположим, что у четырех букв в этой строке выше частотность, а у восемнадцати букв — меньше. Используем

вместо таких букв заполнитель x. Расположение наших четырех букв в соответствии с порядком ETAOIN представлено на рис. 19.5.

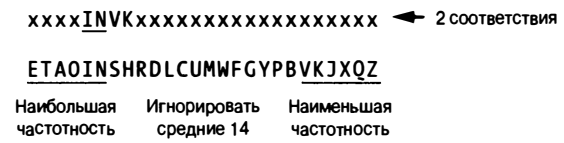


Рис. 19.5. *Увеличение оценки частотного соответствия на два балла в том случае, если указанные четыре буквы расположены в порядке ETAOIN*

В данном случае буквы 'I' и 'N' добавляют два балла к оценке частотного соответствия, так как они входят в число шести наиболее часто встречающихся букв, хотя в этом примере они не встречаются чаще букв 'V' и 'K'. Поскольку оценка частотного соответствия определяется в диапазоне от 0 до 12, эти два балла могут сыграть огромную роль! Но если мы расположим буквы, имеющие одинаковую частотность, в порядке, обратном порядку ETAOIN, то тем самым минимизируем вероятность переоценки роли одной из этих букв (рис. 19.6).

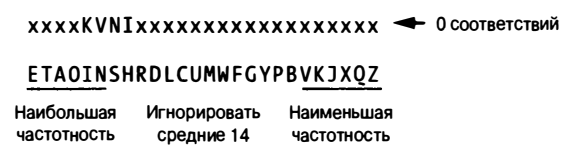


Рис. 19.6. *Если указанные четыре буквы расположить в порядке, обратном порядку ETAOIN, то оценка частотного соответствия не увеличивается*

Располагая буквы в порядке, обратном порядку ETAOIN, мы избегаем искусственного завышения оценки частотного соответствия вследствие случайного выбора порядка 'I', 'N', 'V' и 'K'. Это утверждение останется в силе и в том случае, когда более высокие значения частотности имеют восемнадцать букв, а более низкие – четыре (рис. 19.7).

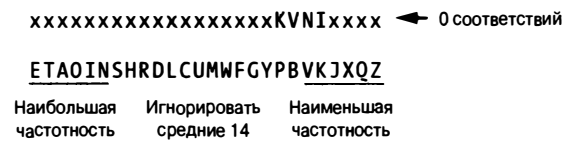


Рис. 19.7. *Обращение порядка ETAOIN для наименее часто встречающихся букв также позволяет избежать увеличения оценки частотного соответствия*

Обращение порядка сортировки гарантирует, что буквы 'K' и 'V' не будут соответствовать никакой из шести букв, наименее часто встречающихся в типичных текстах на английском языке, что также позволяет избежать увеличения оценки частотного соответствия на два балла.

Чтобы отсортировать каждый из списков, содержащихся в словаре `freqToLetter`, в порядке, обратном порядку ETAOIN, понадобится передать метод функции `sort()`, о чем пойдет речь в следующем разделе.

Передача функций как значений

В строке 42, вместо того чтобы вызвать метод `find()`, мы передаем его имя функции `sort()`.

```
42. freqToLetter[freq].sort(key=ETAOIN.find, reverse=True)
```

Мы вправе это сделать, поскольку в Python с функциями можно работать как со значениями. Фактически объявление функции `spam` равносильно сохранению ее определения в переменной `spam`. Чтобы убедиться в этом, введите в интерактивной оболочке следующий код.

```
>>> def spam():
...   print('Hello!')
...
>>> spam()
Hello!
>>> eggs = spam
>>> eggs()
Hello!
```

В данном случае мы объявляем функцию `spam()`, которая выводит на экран строку `'Hello!'`. Но это также означает, что в переменной `spam` хранится определение функции. Затем мы копируем функцию, сохраненную в переменной `spam`, в переменную `eggs`. Теперь мы можем вызывать функцию `eggs()` точно так же, как и функцию `spam()`! Обратите внимание на то, что скобки после имени `spam` в инструкции присваивания не ставятся. Если бы они стояли, то в результате была бы вызвана функция `spam()`, а возвращенное ею значение было бы присвоено переменной `eggs`.

Поскольку функции трактуются как значения, их можно передавать в качестве аргументов при вызове других функций. Введите в интерактивной оболочке следующий код.

```
>>> def doMath(func):
...   return func(10, 5)
...
>>> def adding(a, b):
...   return a + b
...
>>> def subtracting(a, b):
...   return a - b
```

```
  ...
❶ >>> doMath(adding)
  15
  >>> doMath(subtracting)
  5
```

Здесь мы объявляем три функции: `doMath()`, `adding()` и `subtracting()`. Передавая функцию, хранящуюся в переменной `adding`, функции `doMath()` (❶), мы тем самым присваиваем значение переменной `adding` параметру `func`, и поэтому вызов `func(10, 5)` эквивалентен вызову функции `adding()` с аргументами `10` и `5`. Вот почему функция `doMath(adding)` возвращает значение `15`. Точно так же, когда мы передаем переменную `subtracting` функции `doMath()` в качестве аргумента, вызов `doMath(subtracting)` возвращает значение `5`, поскольку `func(10, 5)` — это то же самое, что и `subtracting(10, 5)`.

Передача функции методу `sort()`

Передача функции (или другого метода) методу `sort()` позволяет реализовать нестандартный вариант сортировки. Обычно метод `sort()` сортирует элементы списка по алфавиту.

```
>>> spam = ['C', 'B', 'A']
>>> spam.sort()
>>> spam
['A', 'B', 'C']
```

Но если передать функцию (или метод) в качестве аргумента `key`, то элементы списка сортируются в соответствии со значениями, возвращаемыми этой функцией при передаче ей каждого из элементов списка. Например, методу `sort()` можно передать строковый метод `ETAOIN.find()` в качестве аргумента `key`.

```
>>> ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'
>>> spam.sort(key=ETAOIN.find)
>>> spam
['A', 'C', 'B']
```

При получении аргумента `ETAOIN.find` метод `sort()` не выполняет сортировку в алфавитном порядке. Вместо этого он сначала вызывает метод `find()` для каждого элемента списка, и в результате вызовы `ETAOIN.find('A')`, `ETAOIN.find('B')` и `ETAOIN.find('C')` возвращают индексы `2`, `19` и `11`, соответствующие позиции каждой буквы в строке `ETAOIN`. Для сортировки элементов списка `spam` метод `sort()` использует не исход-

ные строки 'A', 'B' и 'C', а полученные индексы. Именно поэтому строки 'A', 'B' и 'C' после сортировки располагаются в очередности 'A', 'C' и 'B', отражающей порядок их появления в ряду ETAOIN.

Обращение списков букв с помощью метода `sort()`

Чтобы выполнить сортировку букв в порядке, обратном порядку ETAOIN, необходимо сначала отсортировать их на основании строки ETAOIN, присвоив аргументу `key` метода `sort()` метод `ETAOIN.find`. После того как этот метод вызывается для всех букв и находит их индексы, метод `sort()` выполняет сортировку букв на основании их числовых индексов.

Обычно метод `sort()` сортирует элементы списка в алфавитном или числовом порядке *по возрастанию*. Чтобы выполнить сортировку *по убыванию*, необходимо присвоить аргументу `reverse` метода `sort()` значение `True`.

Это выполняется в строке 42.

```
39.     # В-третьих, изменяем порядок букв в каждом списке на
40.     # обратный порядку "ETAOIN" и превращаем списки в строки
41.     for freq in freqToLetter:
42.         freqToLetter[freq].sort(key=ETAOIN.find, reverse=True)
43.         freqToLetter[freq] = ''.join(freqToLetter[freq])
```

Вспомните, что на данном этапе `freqToLetter` — это словарь, ключами которого являются целочисленные значения счетчиков, а значениями — списки букв. Мы сортируем строковые значения, соответствующие ключу `freq`, а не сам словарь `freqToLetter`. Словари нельзя сортировать, поскольку и них нет порядка элементов: не существует первой или последней пары "ключ: значение", в отличие от элементов списка.

Использовав строку `"""Alan Mathison Turing..."""` в качестве значения переменной `freqToLetter`, мы получим следующий вид словаря по завершении цикла.

```
{1: 'Z', 2: 'QJ', 3: 'X', 135: 'A', 8: 'K', 139: 'I', 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'}
```

Обратите внимание на то, что строки при ключах 30, 37 и 58 отсортированы в порядке, обратном порядку ETAOIN. До начала цикла эти пары "ключ: значение" выглядели так: `{30: ['B', 'W'], 37: ['F', 'U'], 58: ['D', 'M'], ...}`. По окончании цикла они приобретают следующий вид: `{30: 'BW', 37: 'FU', 58: 'MD', ...}`.

Вызов метода `join()` в строке 43 превращает список букв в одиночную строку. Например, элемент `freqToLetter[30]` содержит список `['B', 'W']`, который превращается в строку `'BW'`.

Сортировка списков словаря по частотности

Четвертым этапом, выполняемым функцией `getFrequencyOrder()`, является сортировка строк, хранящихся в словаре `freqToLetter`, по частотности и преобразование этих строк в список. Имейте в виду, что в силу неупорядоченности пар "ключ: значение" в словарях список всех ключей или всех значений словаря будет представлять собой список элементов, расположенных в случайном порядке. Отсюда следует, что этот список также нуждается в сортировке.

Использование словарных методов `keys()`, `values()` и `items()`

Каждый из методов `keys()`, `values()` и `items()` словаря преобразует отдельные его элементы в типы данных, не являющиеся словарями. После того как словарь преобразован в другой тип данных, его можно преобразовать в список с помощью функции `list()`.

В качестве примера введите в интерактивной оболочке следующие инструкции.

```
>>> spam = {'cats': 10, 'dogs': 3, 'mice': 3}
>>> spam.keys()
dict_keys(['mice', 'cats', 'dogs'])
>>> list(spam.keys())
['mice', 'cats', 'dogs']
>>> list(spam.values())
[3, 10, 3]
```

Для получения списка всех ключей словаря применяется метод `keys()`, возвращающий объект `dict_keys`, который затем можно передать функции `list()`. Аналогичный метод `values()` возвращает объект `dict_values`. В приведенных выше примерах мы получили с помощью этих методов списки ключей и значений словаря соответственно.

Если нужно одновременно получить и ключи, и значения словаря, используйте метод `items()`, возвращающий объект `dict_items`, который превращает пары "ключ: значение" в кортежи. Далее этот объект можно передать функции `list()`. Введите в интерактивной оболочке следующий код.

```
>>> spam = {'cats': 10, 'dogs': 3, 'mice': 3}
>>> list(spam.items())
[('mice', 3), ('cats', 10), ('dogs', 3)]
```

С помощью вызовов `items()` и `list()` мы преобразуем пары "ключ: значение" словаря в список кортежей. Это именно то, что необходимо предпринять в отношении словаря `freqToLetter` для того, чтобы можно было отсортировать буквенные строки по частотности.

Преобразование элементов словаря в сортируемый список

В словаре `freqToLetter` ключами являются целочисленные счетчики частотности букв, а значениями — однобуквенные строки. Чтобы отсортировать строки по частотности, мы вызываем метод `items()` и функцию `list()` для создания списка кортежей "ключ: значение". В строке 47 список кортежей сохраняется в переменной `freqPairs`.

```
45.     # В-четвертых, преобразуем словарь freqToLetter в список
46.     # кортежей (ключ, значение) и сортируем его
47.     freqPairs = list(freqToLetter.items())
```

В строке 48 мы передаем методу `sort()` функцию `getItemAtIndexZero()`, которая была создана ранее.

```
48.     freqPairs.sort(key=getItemAtIndexZero, reverse=True)
```

Функция `getItemAtIndexZero()` получает первый элемент кортежа, которым в данном случае является целочисленный счетчик частотности. Это означает, что элементы списка `freqPairs` сортируются по числовым значениям счетчиков частотности. В строке 48 мы передаем методу `sort()` значение `True` в качестве аргумента `reverse`, поэтому кортежи сортируются по убыванию счетчиков частотности.

Взяв строку `"""Alan Mathison Turing..."""` в качестве примера, мы должны получить после выполнения строки 48 следующий список `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')]
```

Теперь переменная `freqPairs` содержит список кортежей, расположенных в порядке убывания частотности букв. В каждом кортеже первым значением является целое число, представляющее собой счетчик частотности, а вторым — строка, содержащая буквы с соответствующей частотностью.

Создание списка отсортированных букв

Пятый этап, выполняемый функцией `getFrequencyOrder()`, — создание списка всех строк на основе отсортированного списка кортежей, хранящегося в переменной `freqPairs`. Мы хотим получить единую строку, в которой буквы расположены по убыванию частотности, поэтому нам не нужны целочисленные значения, хранящиеся в кортежах. Сначала переменной `freqOrder` присваивается пустой список в строке 52, а затем в цикле `for` в конец списка `freqOrder` присоединяется строка с индексом 1 из каждого кортежа, хранящегося в переменной `freqPairs`.

```
50.     # В-пятых, после того как буквы были упорядочены по частотности,
51.     # извлекаем все буквы для формирования окончательной строки
52.     freqOrder = []
53.     for freqPair in freqPairs:
54.         freqOrder.append(freqPair[1])
```

По завершении цикла переменная `freqOrder` должна содержать список `['E', 'T', 'I', 'A', 'N', 'O', 'R', 'S', 'H', 'C', 'L', 'MD', 'G','FU', 'P', 'BW', 'Y', 'V', 'K', 'X', 'QJ', 'Z']`.

Далее элементы списка `freqOrder` объединяются в строку с помощью метода `join()`:

```
56.     return ''.join(freqOrder)
```

В примере со строкой `"""Alan Mathison Turing..."""` функция `getFrequencyOrder()` возвращает строку `'ETIANORSHCLMDGFUPBWYVKXQJZ'`. Таким образом, в данном примере чаще всего встречается буква 'E', на втором месте — буква 'T', на третьем — буква 'I' и т.д.

Получив частотное распределение букв в виде строки, мы можем сравнить его с аналогичным распределением для типичных текстов на английском языке (`'ETAOINSHRDLCUMWFGYPBVKJXQZ'`), чтобы оценить степень их близости.

Вычисление оценки частотного соответствия букв в сообщении

Функция `englishFreqMatchScore()` получает строку `message` и возвращает целое число в интервале от 0 до 12, являющееся оценкой частотного соответствия для данной строки. Чем выше эта оценка, тем больше частотность букв сообщения соответствует частотности букв в типичных текстах на английском языке.

```
59. def englishFreqMatchScore(message):
60.     # Возвращает оценку частотного соответствия для строки
61.     # message. Совпадения проверяются по шести наиболее
62.     # и наименее часто встречающимся буквам в строке
63.     # и в английском языке в целом.
64.
65.     freqOrder = getFrequencyOrder(message)
```

Первое, что необходимо сделать для вычисления оценки частотного соответствия букв, — это упорядочить буквы в соответствии с их частотностью, что и делается в строке 65 с помощью функции `getFrequencyOrder()`. Полученная последовательность сохраняется в виде строки в переменной `freqOrder`.

Переменная `matchScore` инициализируется значением `0` в строке 67 и инкрементируется в цикле `for` (строка 69). В этом цикле первые шесть букв строки `ETAOIN` сравниваются с первыми шестью буквами строки `freqOrder`, и за каждую общую букву начисляется дополнительный балл.

```
67.     matchScore = 0
68.     # Число соответствий для шести наиболее часто встречающихся букв
69.     for commonLetter in ETAOIN[:6]:
70.         if commonLetter in freqOrder[:6]:
71.             matchScore += 1
```

Вспомните о том, что срез `[:6]` — это то же самое, что срез `[0:6]`, поэтому в строках 69 и 70 извлекаются первые шесть букв строк `ETAOIN` и `freqOrder` соответственно. Если любая из букв 'E', 'T', 'A', 'O', 'I' или 'N' также встречается среди первых шести букв строки `freqOrder`, то условие в строке 70 становится равно `True`, и тогда в строке 71 инкрементируется значение `matchScore`.

Строки 73–75 аналогичны строкам 69–71, за исключением того, что в данном случае проверяется наличие последних шести букв строки `ETAOIN` ('V', 'K', 'J', 'X', 'Q' и 'Z') среди последних шести букв строки `freqOrder` и соответствующим образом инкрементируется значение `matchScore`.

```
72.     # Число соответствий для шести наименее часто встречающихся букв
73.     for uncommonLetter in ETAOIN[-6:]:
74.         if uncommonLetter in freqOrder[-6:]:
75.             matchScore += 1
```

В строке 77 функция возвращает целое число, содержащееся в переменной `matchScore`:

```
77.     return matchScore
```

При вычислении оценки частотного соответствия мы игнорируем 14 букв, находящихся в середине упорядоченного списка букв. Частотности этих букв слишком близки между собой, чтобы они могли дать нам сколь-нибудь полезную информацию.

Резюме

В этой главе вы узнали о том, как применять функцию `sort()` для сортировки списков в алфавитном или числовом порядке и как использовать ее аргументы `reverse` и `key` для изменения параметров сортировки. Мы рассмотрели, как преобразовывать словари в списки с помощью словарных методов `keys()`, `values()` и `items()`. Вы также узнали о том, что функцию можно передать в качестве аргумента другой функций.

В главе 20 мы воспользуемся написанным в этой главе модулем частотного анализа для взлома шифра Виженера.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Что такое частотный анализ?
2. Какие шесть букв чаще всего встречаются в текстах на английском языке?
3. Что будет содержать переменная `spam` после выполнения следующего кода?

```
spam = [4, 6, 2, 8]
spam.sort(reverse=True)
```

4. Если переменная `spam` содержит словарь, то как получить список его ключей?

20

ВЗЛОМ ШИФРА ВИЖЕНЕРА

"Неприкосновенность частной жизни – неотъемлемое право человека и обязательное условие того, чтобы он мог жить с чувством самоуважения и собственного достоинства".

Брюс Шнайер, криптограф, 2006 г.

Существуют два метода взлома шифра Виженера. Один из них — *перебор по словарю*, или *словарная атака* (dictionary attack), предполагающий полный перебор всех слов из файла словаря в качестве ключа. Это сработает только в том случае, если искомый ключ является английским словом, например "RAVEN" или "DESK". Второй, более сложный метод, открытый в XIX веке математиком Чарльзом Бэббиджем, дает результат, даже если ключ представляет собой случайный набор букв, такой как "VUWFE" или "PNFJ". В этой главе мы напишем программы для взлома шифра Виженера обоими методами.

В этой главе...

- Перебор по словарю
- Метод Касиски
- Нахождение множителей
- Тип данных `set` и функция `set()`
- Списковый метод `extend()`
- Функция `itertools.product()`

Использование перебора по словарю для взлома шифра Виженера методом грубой силы

Сначала мы используем метод перебора по словарю для взлома шифра Виженера. Файл словаря *dictionary.txt* (доступный на сайте издательства; см. введение) насчитывает примерно 45 000 английских слов. На компьютере автора дешифрование сообщения размером с длинный абзац с использованием каждого из этих слов в качестве ключа заняло менее пяти минут. Это означает, что в тех случаях, когда шифротекст зашифрован с помощью английского слова, он уязвим для перебора по словарю. Рассмотрим исходный код программы, реализующей перебор по словарю для взлома шифра Виженера.

Исходный код программы `Vigenere Dictionary Hacker`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *vigenereDictionaryHacker.py*. Убедитесь в том, что файлы *detectEnglish.py*, *vigenereCipher.py* и *pyperclip.py* находятся в той же самой папке. Запустите программу, нажав клавишу <F5>.

vigenere.Dictionary.Hacker.py

```
1 # Взлом шифра Виженера методом перебора по словарю
2 # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import detectEnglish, vigenereCipher, pyperclip
5.
6. def main():
```

```
 7.     ciphertext = """Tzx isnz eccjxkg nfq lol mys bbqq I lxcz."""
 8.     hackedMessage = hackVigenereDictionary(ciphertext)
 9.
10.     if hackedMessage != None:
11.         print('Copying hacked message to clipboard:')
12.         print(hackedMessage)
13.         pyperclip.copy(hackedMessage)
14.     else:
15.         print('Failed to hack encryption.')
16.
17.
18. def hackVigenereDictionary(ciphertext):
19.     fo = open('dictionary.txt')
20.     words = fo.readlines()
21.     fo.close()
22.
23.     for word in words:
24.         word = word.strip()  # удалить завершающий символ новой строки
25.         decryptedText = vigenereCipher.decryptMessage(word, ciphertext)
26.         if detectEnglish.isEnglish(decryptedText, wordPercentage=40):
27.             # Спросить у пользователя, найден ли ключ дешифрования
28.             print()
29.             print('Possible encryption break:')
30.             print('Key ' + str(word) + ': ' + decryptedText[:100])
31.             print()
32.             print('Enter D for done, or just press Enter to continue
                        breaking:')
33.             response = input('> ')
34.
35.             if response.upper().startswith('D'):
36.                 return decryptedText
37.
38. if __name__ == '__main__':
39.     main()
```

Пример выполнения программы Vigenere Dictionary Hacker

Выполнив программу, вы должны получить следующий результат.

```
Possible encryption break:
Key ASTROLOGY: The recl yecrets crk not the qnks I tell.

Enter D for done, or just press Enter to continue breaking:
>

Possible encryption break:
```

```
Key ASTRONOMY: The real secrets are not the ones I tell.

Enter D for done, or just press Enter to continue breaking:
> d
Copying hacked message to clipboard:
The real secrets are not the ones I tell.
```

Первый ключ, предложенный программой (ASTROLOGY), не подходит, поэтому пользователь нажимает клавишу <Enter>, позволяя программе выполняться до тех пор, пока не будет найден корректный ключ (ASTRONOMY).

О структуре программы

Исходный код программы *vigenereDictionaryHacker.py* в основном содержит уже знакомые нам инструкции, поэтому мы не будем рассматривать его построчно. Общий алгоритм таков: функция `hackVigenereDictionary()` пытается использовать каждое слово из файла словаря в качестве ключа для дешифрования шифротекста, и если результат дешифрования представляет собой осмысленный текст на английском языке (в соответствии с критериями модуля `detectEnglish`), то она выводит его на экран и предлагает пользователю завершить процесс дешифрования или продолжить его.

Обратите внимание на вызов метода `readlines()` для файлового объекта, возвращаемого функцией `open()`:

```
20.     words = fo.readlines()
```

В отличие от метода `read()`, который возвращает все содержимое файла в виде одной строки, метод `readlines()` возвращает список, каждый элемент которого является строкой из файла. Поскольку каждая строка словаря содержит по одному слову, переменная `words` будет содержать список всех английских слов от *Aarhus* до *Zurich*.

Остальная часть функции (строки 23–36) аналогична коду, который использовался в программе взлома перестановочного шифра, рассмотренной в главе 12. В цикле `for` мы перебираем все слова, содержащиеся в списке `words`, дешифруем шифротекст с использованием очередного слова в качестве ключа, а затем вызываем функцию `detectEnglish.isEnglish()` для проверки того, что результат представляет собой осмысленный текст на английском языке.

Теперь рассмотрим, как взломать шифр Виженера даже в том случае, когда ключом служит случайный набор букв, а не слово из словаря.

Использование метода Касиски для определения длины ключа

Метод Касиски предназначен для определения длины ключа шифра Виженера. Это дает возможность применить частотный анализ для независимого взлома каждого из подключей. Первым человеком, которому удалось взломать шифр Виженера, был Чарльз Бэббидж, но об этом никто не узнал, так как он не опубликовал свои результаты. Позже аналогичное открытие сделал немецкий математик Фридрих Касиски, в честь которого и был назван метод. Рассмотрим этапы алгоритма, которые будут реализованы нашей программой взлома шифра Виженера.

Нахождение повторяющихся сегментов

В соответствии с методом Касиски первое, что необходимо сделать, — это найти в шифротексте повторяющиеся сегменты, содержащие по крайней мере три буквы. Такие последовательности могут быть повторениями одних и тех же наборов символов в исходном тексте, зашифрованных с помощью одних и тех же подключей. Например, если зашифровать открытый текст "THE CAT IS OUT OF THE BAG" с помощью ключа "SPILLTHEBEANS", то получим следующее:

```
THECATISOUTOFTHEBAG
SPILLTHEBEANSSPILLT
LWMNLMPWPYTBXLWMMLZ
```

Обратите внимание на то, что сочетание "LWM" повторяется дважды. Причина заключается в том, что группа букв "LWM" в шифротексте — это слово "THE", зашифрованное с помощью одних и тех же букв ключа, "SPI", поскольку так получилось, что ключ повторяется на втором слове "THE". Количество букв от начала первого буквосочетания "LWM" до начала второго такого же буквосочетания, которое мы будем называть *интервалом повторения*, равно 13. Это дает основания полагать, что длина ключа, применяемого в данном шифротексте, составляет 13 букв. Таким образом, мы выявили длину ключа путем простого анализа повторяющихся последовательностей.

Однако в большинстве шифротекстов ключ не будет удобным образом выравниваться по повторяющимся последовательностям букв, или же ключ может встречаться между повторяющимися последовательностями более одного раза, т.е. интервал повторения будет кратным длине ключа, а не равным ей. Чтобы попытаться найти пути решения этой проблемы,

рассмотрим более длинный пример, в котором нам не известно, что собой представляет ключ.

Если удалить из шифротекста "PPQCA XQVEKGYBNKMAZU YBNGBAL JON I TSZM JYIM. VRAG VOHT VRAU C TKSG.DDWUO XITLAZU VAVV RAZ C VKB QP IWPOU" все небуквенные символы, то он примет вид строки, приведенной на рис. 20.1. На этом рисунке также выделены повторяющиеся последовательности — "VRA", "AZU" и "YBN" — и указано количество букв между каждой парой последовательностей.

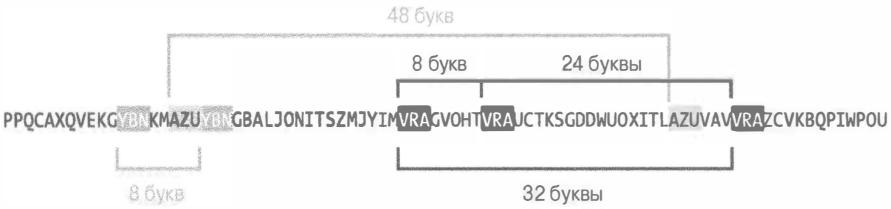


Рис. 20.1. *Повторяющиеся последовательности в шифротексте*

В этом примере существует несколько возможных вариантов длины ключа. Следующий шаг метода Касиски заключается в вычислении множителей всех интервалов повторения для сужения круга возможных вариантов.

Определение множителей в интервалах повторения

В нашем случае промежутки между последовательностями, т.е. интервалы повторения, равны 8, 8, 24, 32 и 48. Но для нас важны не они, а их множители.

Чтобы понять, почему это так, рассмотрим сообщение "THEDOGANDTHECAT" (табл. 20.1) и попытаемся зашифровать его с помощью девятибуквенного ключа ABCDEFGHI, а также трехбуквенного ключа XYZ. Каждый из этих ключей повторяется на протяжении всей длины сообщения.

Таблица 20.1. Шифрование сообщения "THEDOGANDTHECAT" двумя различными ключами

	Шифрование ключом ABCDEFGHI	**Шифрование ключом XYZ**
Исходное сообщение	**THE**DOGAND**THE**CAT	**THE**DOGAND**THE**CAT
Ключ (повторяющийся)	**ABC**DEFGHI**ABC**DEF	**XYZ**XYZXYZ**XYZ**XYZ
Шифротекст	**TIG**GSLGUL**TIG**FEY	**QFD**AMFXLC**QFD**ZYS

Как и ожидалось, два ключа приводят к двум разным шифротекстам. Разумеется, ни исходное сообщение, ни ключ не будут известны взломщи-

ку шифра, но он заметит, что в шифротексте "**TIG**GSLGUL**TIG**FEY" последовательность "TIG" встречается при значениях индекса 0 и 9. Поскольку 9 – 0 = 9, интервал повторения для данной последовательности равен 9, а это может свидетельствовать о том, что истинный ключ является девятибуквенным, и в рассматриваемом случае он именно такой.

Однако и в шифротексте "**QFD**AMFXLC**QFD**ZYS" имеется своя повторяющаяся последовательность, "QFD", которая встречается при значениях индекса 0 и 9. Интервал повторения здесь тоже равен 9, и это, на первый взгляд, указывает на то, что ключ, использованный для получения данного шифротекста, насчитывает девять букв. Однако мы знаем, что в действительности ключ содержит всего три буквы: "XYZ".

Повторяющиеся последовательности встречаются тогда, когда одни и те же буквы в исходном сообщении ("THE" в нашем примере) шифруются одними и теми же буквами ключа (в данном примере это "ABC" и "XYZ"), что происходит в том случае, если аналогичные группы в сообщении и ключе "выравниваются" и шифруются в одну и ту же последовательность. Подобное выравнивание может встречаться в позициях, кратных длине реального ключа (таких, как 3, 6, 9, 12 и т.д.), и именно поэтому трехбуквенный ключ может порождать последовательности с интервалом повторения, равным 9.

Таким образом, возможная длина ключа может быть обусловлена не только интервалом повторения, но и его множителями. Множителями числа 9 являются числа 9, 3 и 1. Поэтому, если вы нашли последовательности с интервалом повторения 9, то должны рассмотреть два возможных варианта длины ключа: 9 и 3. Ключ длиной 1 можно игнорировать, поскольку шифр Виженера с ключом такой длины сводится к шифру Цезаря.

Второй этап метода Касиски включает нахождение множителей каждого из интервалов повторения (см. рис. 20.1 и табл. 20.2).

Таблица 20.2. Множители выявленных интервалов повторения

Интервал повторения	Множители
8	2, 4, 8
24	2, 4, 6, 8, 12, 24
32	2, 4, 8, 16
48	2, 4, 6, 8, 12, 24, 48

Числа 8, 8, 24, 32 и 48 в совокупности имеют следующие множители: 2, 2, 2, 2, 4, 4, 4, 4, 6, 6, 8, 8, 8, 8, 12, 12, 16, 24, 24 и 48.

С наибольшей вероятностью ключами окажутся чаще всего встречающиеся множители. Поскольку в данном примере чаще остальных встречаются множители 2, 4 и 8, то именно они являются наиболее вероятными вариантами длины ключа Виженера.

Получение каждой n-й буквы строки

Теперь, когда нам известны возможные значения длины ключа, мы можем воспользоваться этой информацией для дешифрования сообщения по одному подключу за раз. Предположим, что длина ключа в нашем примере равна 4. Если нам не удастся взломать шифротекст, мы попытаемся использовать ключи длиной 2 или 8.

Поскольку ключ применяется циклично, использование ключа длиной 4 означает, что каждая четвертая буква открытого текста шифруется первым подключом, каждая четвертая буква исходного текста, начиная со второй, шифруется вторым подключом и т.д. Мы сформируем строки из букв шифротекста, шифруемых одним и тем же подключом. Сначала мы определим, какой будет каждая четвертая буква строки, если начинать с разных букв, а затем объединим все такие буквы в одну строку. В рассматриваемых примерах каждая четвертая буква выделяется полужирным шрифтом.

Идентифицируем каждую четвертую букву, начиная с первой:

```
PPQCAXQVEKGYBNKMAZUYBNGBALJONITSZMJYIMVRAGVOHTVRAUCTKSGDDWUOXITLAZUVAVVRAZCV
KBQPIWPOU
```

Далее находим каждую четвертую букву, начиная со второй:

```
PPQCAXQVEKGYBNKMAZUYBNGBALJONITSZMJYIMVRAGVOHTVRAUCTKSGDDWUOXITLAZUVAVVRAZCV
KBQPIWPOU
```

Затем проделываем то же самое, начиная с третьей и четвертой буквы, пока не исчерпаем всю проверяемую длину ключа. В табл. 20.3 приведены объединенные строки, составленные из букв, выделенных полужирным шрифтом на каждой итерации.

Таблица 20.3. Строки, составленные из каждой четвертой буквы

Начало отсчета	Строка
Первая буква	PAEBABANZIAHAKDXAAAKIU
Вторая буква	PXKNZNLIMMGTUSWIZVZBW
Третья буква	QQGKUGJTJVVVCGUTUVCQP
Четвертая буква	CVYMYBOSYRORTDOLVRVPO

Применение частотного анализа для взлома каждого подключа

Если мы угадали длину ключа, то каждая из четырех строк, созданных нами в предыдущем разделе, должна быть зашифрована с помощью одного ключа. Это означает, что если строка, зашифрованная корректным ключом, подвергается частотному анализу, то дешифрованные буквы, вероятнее всего, будут иметь высокую оценку частотного соответствия. Рассмотрим, как это работает, используя в качестве примера первую строку: `PAEBABANZIAHAKDXAAAKIU`.

Прежде всего, дешифруем строку 26 раз (по одному разу для каждого из 26 возможных подключей), используя функцию `vigenereCipher.decryptMessage()` (см. главу 18). Затем протестируем каждую дешифрованную строку, используя функцию частотного анализа для английского языка `freqAnalysis.englishFreqMatchScore()` (см. главу 19). Введите в интерактивной оболочке следующий код.

```
>>> import freqAnalysis, vigenereCipher
>>> for subkey in 'ABCDEFGHJIJKLMNOPQRSTUVWXYZ':
...   decryptedMessage = vigenereCipher.decryptMessage(subkey,
        'PAEBABANZIAHAKDXAAAKIU')
      print(subkey, decryptedMessage,
        freqAnalysis.englishFreqMatchScore(decryptedMessage))
...
A PAEBABANZIAHAKDXAAAKIU 2
B OZDAZAZMYHZGZJCWZZZJHT 1
--опущено--
```

Результаты представлены в табл. 20.4.

Таблица 20.4. Оценки частотного соответствия для каждого варианта дешифрования

Подключ	Дешифрование	Оценка частотного соответствия
'A'	'PAEBABANZIAHAKDXAAAKIU'	2
'B'	'OZDAZAZMYHZGZJCWZZZJHT'	1
'C'	'NYCZYZYLXGYFYIBVYYYIGS'	1
'D'	'MXBYXYXKWFXEXHAUXXXHFR'	0
'E'	'LWAXWXWJVEWDWGZTWWWGEQ'	1
'F'	'KVZWVWVIUDVCVFYSVVVFDP'	0
'G'	'JUYVUVUHTCUBUEXRUUUECO'	1

Окончание табл. 20.4

Подключ	Дешифрование	Оценка частотного соответствия
'H'	'ITXUTUTGSBTATDWQTTTDBN'	1
'I'	'HSWTSTSFRASZSCVPSSSCAM'	2
'J'	'GRVSRSREQZRYRBUORRRBZL'	0
'K'	'FQURQRQDPYQXQATNQQQAYK'	1
'L'	'EPTQPQPCOXPWPZSMPPPZXJ'	0
'M'	'DOSPOPOBNWOVOYRLOOOYWI'	1
'N'	'CNRONONAMVNUNXQKNNNXVH'	2
'O'	'BMQNMNMZLUMTMWPJMMMWUG'	1
'P'	'ALPMLMLYKTLSLVOILLLVTF'	1
'Q'	'ZKOLKLKXJSKRKUNHKKKUSE'	0
'R'	'YJNKJKJWIRJQJTMGJJJTRD'	1
'S'	'XIMJIJIVHQIPISLFIIISQC'	1
'T'	'WHLIHIHUGPHOHRKEHHHRPB'	1
'U'	'VGKHGHGTFOGNGQJDGGGQOA'	1
'V'	'UFJGFGFSENFMFPICFFFPNZ'	1
'W'	'TEIFEFERDMELEOHBEEEOMY'	2
'X'	'SDHEDEDQCLDKDNGADDDNLX'	2
'Y'	'RCGDCDCPBKCJCMFZCCCMKW'	0
'Z'	'QBFCBCBOAJBIBLEYBBBLJV'	0

Подключи, которые приводят к получению вариантов дешифрования с наибольшей оценкой, являются самыми вероятными кандидатами на роль реального ключа. В табл. 20.4 подключи `'A'`, `'I'`, `'N'`, `'W'` и `'X'` дают наивысшую оценку частотного соответствия для первой строки. Обратите внимание на то, что оценки в целом имеют низкие значения, поскольку объем шифротекста не позволяет получить достаточно длинную тестовую строку, но для данного примера этого вполне достаточно.

Следующий шаг заключается в повторении этого процесса для остальных трех строк с целью выявления наиболее вероятных вариантов подключей. Окончательные результаты приведены в табл. 20.5.

Поскольку для первой строки существуют пять возможных подключей, для второй — два, для третьей — один и для четвертой — пять, общее количество возможных комбинаций составляет 50 ($5 \cdot 2 \cdot 1 \cdot 5$). Другими слова-

ми, нам нужно перебрать 50 возможных ключей. Это намного лучше, чем полный перебор 26 · 26 · 26 · 26 (или 456 976) возможных ключей, к которому пришлось бы прибегнуть, если бы мы не сузили список кандидатов. Разница становится еще более существенной в случае ключей Виженера большей длины.

Таблица 20.5. Наиболее вероятные подключи для тестируемых строк

Строка шифротекста	Наиболее вероятные подключи
PAEBABANZIAHAKDXAAAKIU	A, I, N, W, X
PXKNZNLIMMGTUSWIZVZBW	I, Z
QQGKUGJTJVVVCGUTUVCQP	C
CVYMYBOSYRORTDOLVRVPO	K, N, R, V, Y

Перебор возможных подключей методом грубой силы

Метод грубой силы предполагает полный перебор всех возможных комбинаций подключей. Все 50 возможных комбинаций приведены ниже.

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

Завершающим этапом алгоритма, реализуемого нашей программой взлома шифра Виженера, будет тестирование всех 50 возможных вариантов дешифрования на полном шифротексте, чтобы увидеть, какие из них приводят к получению осмысленного текста на английском языке. В ходе этого процесса мы должны обнаружить, что ключом для дешифрования шифротекста "PPQCA XQVEKG..." является WICK.

Исходный код программы `Vigenere Hacker`

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New. Введите в этом окне приведенный ниже код и сохраните его в файле *vigenereHacker.py*. Убедитесь в том, что файлы *detectEnglish.py*, *freqAnalysis.py*, *vigenereCipher.py* и *pyperclip.py* находятся в той же самой папке. Запустите программу, нажав клавишу <F5>.

Шифротекст, заданный в строке 17 программы, неудобно вводить вручную. Чтобы избежать возможных опечаток, скопируйте его из файла исходного кода, доступного на сайте издательства (см. введение). Для выявления возможных различий между текстом вашей программы и текстом программы, приведенным в книге, воспользуйтесь онлайн-утилитой `diff` (`http://inventwithpython.com/hacking/diff/`).

vigenere.Hacker.py

```
 1. # Программа взлома шифра Виженера
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import itertools, re
 5. import vigenereCipher, pyperclip, freqAnalysis, detectEnglish
 6.
 7. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 8. MAX_KEY_LENGTH = 16        # ограничение длины проверяемых ключей
 9. NUM_MOST_FREQ_LETTERS = 4  # ограничение количества букв на подключ
10. SILENT_MODE = False        # True - отключение вывода
11. NONLETTERS_PATTERN = re.compile('[^A-Z]')
12.
13.
14. def main():
15.     # Этот шифротекст можно скопировать из
16.     # файла исходного кода (см. введение)
17.     ciphertext = """Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi,
          lgouqdaf, kdmktsvmztsl, izr xoexghzr kkusitaaf. Vz wsa twbhdg
          ubalmmzhdad qz
          --опущено--
          azmtmd'g widt ion bwnafz tzm Tcpsw wr Zjrva ivdcz eaigd yzmbo
          Tmzubb a kbmhptgzk dvrvwz wa efiohzd."""
18.       hackedMessage = hackVigenere(ciphertext)
19.
20.     if hackedMessage != None:
21.         print('Copying hacked message to clipboard:')
22.         print(hackedMessage)
23.         pyperclip.copy(hackedMessage)
24.     else:
25.         print('Failed to hack encryption.')
26.
27.
```

```
28. def findRepeatSequencesSpacings(message):
29.     # Находит в сообщении любые 3-, 4- и 5-буквенные повторяющиеся
30.     # последовательности. Возвращает словарь, в котором ключи - это
31.     # последовательности, а значения - списки интервалов повторения.
32.
33.     # Используем регулярное выражение для удаления небуквенных символов
34.     message = NONLETTERS_PATTERN.sub('', message.upper())
35.
36.     # Получение списка последовательностей, найденных в сообщении
37.     seqSpacings = {}  # ключи - последовательности, значения -
                          списки интервалов повторения
38.     for seqLen in range(3, 6):
39.         for seqStart in range(len(message) - seqLen):
40.             # Получение очередной последовательности
41.             seq = message[seqStart:seqStart + seqLen]
42.
43.             # Поиск этой последовательности в остальной части сообщения
44.             for i in range(seqStart + seqLen, len(message) - seqLen):
45.                 if message[i:i + seqLen] == seq:
46.                     # Найдена повторяющаяся последовательность
47.                     if seq not in seqSpacings:
48.                         seqSpacings[seq] = []  # пустой список
49.
50.                         # Добавить интервал повторения между исходной
51.                         # и повторившейся последовательностями
52.                         seqSpacings[seq].append(i - seqStart)
53.     return seqSpacings
54.
55.
56. def getUsefulFactors(num):
57.     # Возвращает список полезных множителей параметра num. Полезными
58.     # считаются множители от 2 до MAX_KEY_LENGTH. Например, вызов
59.     # getUsefulFactors(144) вернет [2, 3, 4, 6, 8, 9, 12, 16].
60.
61.     if num < 2:
62.         return []  # числа, меньшие 2, не имеют полезных множителей
63.
64.     factors = []  # список найденных множителей
65.
66.     # При поиске множителей необходимо проверять лишь целые
67.     # числа вплоть до MAX_KEY_LENGTH
68.     for i in range(2, MAX_KEY_LENGTH + 1):  # множитель 1 бесполезен
69.         if num % i == 0:
70.             factors.append(i)
71.             otherFactor = int(num / i)
72.             if otherFactor < MAX_KEY_LENGTH + 1 and otherFactor != 1:
73.                 factors.append(otherFactor)
74.     return list(set(factors))  # удаляем дубликаты
75.
76.
```

```
77. def getItemAtIndexOne(x):
78.     return x[1]
79.
80.
81. def getMostCommonFactors(seqFactors):
82.     # Во-первых, подсчитываем повторы множителей в словаре seqFactors
83.     factorCounts = {}  # ключ - множитель; значение - число повторений
84.
85.     # Ключи словаря seqFactors - это цепочки букв, а значения - списки
86.     # множителей интервалов повторения. Словарь выглядит примерно так:
87.     # {'GFD': [2, 3, 4, 6, 9, 12, 18, 23, 36, 46, 69, 92, 138, 207],
          'ALW': [2, 3, 4, 6, ...], ...}
88.     for seq in seqFactors:
89.         factorList = seqFactors[seq]
90.         for factor in factorList:
91.             if factor not in factorCounts:
92.                 factorCounts[factor] = 0
93.             factorCounts[factor] += 1
94.
95.     # Во-вторых, объединяем множители и счетчики повторений в кортежи
96.     # и создаем список таких кортежей, чтобы их можно было сортировать
97.     factorsByCount = []
98.     for factor in factorCounts:
99.         # Исключаем множители, которые больше, чем MAX_KEY_LENGTH
100.        if factor <= MAX_KEY_LENGTH:
101.            # factorsByCount - список кортежей (множитель, счетчик).
102.            # Типичный вид: [(3, 497), (2, 487), ...]
103.            factorsByCount.append( (factor, factorCounts[factor]) )
104.
105.    # Сортировка списка по счетчикам повторений
106.    factorsByCount.sort(key=getItemAtIndexOne, reverse=True)
107.
108.    return factorsByCount
109.
110.
111. def kasiskiExamination(ciphertext):
112.    # Находим последовательности длиной от 3 до 5 букв, встречающиеся
113.    # в шифротексте неоднократно. Словарь repeatedSeqSpacings выглядит
114.    # примерно так: {'EXG': [192], 'NAF': [339, 972, 633], ... }
115.    repeatedSeqSpacings = findRepeatSequencesSpacings(ciphertext)
116.
117.    # См. описание словаря seqFactors в функции getMostCommonFactors()
118.    seqFactors = {}
119.    for seq in repeatedSeqSpacings:
120.        seqFactors[seq] = []
121.        for spacing in repeatedSeqSpacings[seq]:
122.            seqFactors[seq].extend(getUsefulFactors(spacing))
123.
124.    # См. описание factorsByCount в функции getMostCommonFactors()
125.    factorsByCount = getMostCommonFactors(seqFactors)
```

```
126.
127.     # Извлекаем множители из списка factorsByCount
128.     # и помещаем их в список allLikelyKeyLengths, чтобы
129.     # с ними было проще работать
130.     allLikelyKeyLengths = []
131.     for twoIntTuple in factorsByCount:
132.         allLikelyKeyLengths.append(twoIntTuple[0])
133.
134.     return allLikelyKeyLengths
135.
136.
137. def getNthSubkeysLetters(nth, keyLength, message):
138.     # Возвращает каждую n-ю букву из каждого набора длиной keyLength.
139.     # Так, getNthSubkeysLetters(1, 3, 'ABCABCABC') возвращает 'AAA',
140.     #       getNthSubkeysLetters(2, 3, 'ABCABCABC') возвращает 'BBB',
141.     #       getNthSubkeysLetters(3, 3, 'ABCABCABC') возвращает 'CCC',
142.     #       getNthSubkeysLetters(1, 5, 'ABCDEFGHI') возвращает 'AF'.
143.
144.     # Используем регулярное выражение для удаления небуквенных символов
145.     message = NONLETTERS_PATTERN.sub('', message)
146.
147.     i = nth - 1
148.     letters = []
149.     while i < len(message):
150.         letters.append(message[i])
151.         i += keyLength
152.     return ''.join(letters)
153.
154.
155. def attemptHackWithKeyLength(ciphertext, mostLikelyKeyLength):
156.     # Определяем наиболее вероятные буквы для каждого подключа
157.     ciphertextUp = ciphertext.upper()
158.     # allFreqScores - список длиной mostLikelyKeyLength,
159.     # элементами которого являются списки freqScores
160.     allFreqScores = []
161.     for nth in range(1, mostLikelyKeyLength + 1):
162.         nthLetters = getNthSubkeysLetters(nth, mostLikelyKeyLength,
              ciphertextUp)
163.
164.         # freqScores - список кортежей вида [(<буква>,
165.         # <оценка частотного соответствия>), ... ]. Список сортируется
166.         # по оценкам: чем выше, тем лучше. См. комментарии к функции
167.         # englishFreqMatchScore() в модуле freqAnalysis.py.
168.         freqScores = []
169.         for possibleKey in LETTERS:
170.             decryptedText = vigenereCipher.decryptMessage(possibleKey,
                  nthLetters)
171.             keyAndFreqMatchTuple = (possibleKey,
                  freqAnalysis.englishFreqMatchScore(decryptedText))
172.             freqScores.append(keyAndFreqMatchTuple)
```

```
        # Сортировка по оценкам частотного соответствия
        freqScores.sort(key=getItemAtIndexOne, reverse=True)

        allFreqScores.append(freqScores[:NUM_MOST_FREQ_LETTERS])

    if not SILENT_MODE:
        for i in range(len(allFreqScores)):
            # Используем i + 1, чтобы первая буква не считалась 0-й
            print('Possible letters for letter %s of the key:' %
                  (i + 1), end='')
            for freqScore in allFreqScores[i]:
                print('%s ' % freqScore[0], end='')
            print()  # переход на новую строку

    # Проверяем все комбинации наиболее вероятных букв
    # для каждой позиции в ключе
    for indexes in itertools.product(range(NUM_MOST_FREQ_LETTERS),
      repeat=mostLikelyKeyLength):
        # Создаем возможный ключ из букв в списке allFreqScores
        possibleKey = ''
        for i in range(mostLikelyKeyLength):
            possibleKey += allFreqScores[i][indexes[i]][0]

        if not SILENT_MODE:
            print('Attempting with key: %s' % (possibleKey))

        decryptedText = vigenereCipher.decryptMessage(possibleKey,
          ciphertextUp)

        if detectEnglish.isEnglish(decryptedText):
            # Задаем исходный регистр букв во взломанном шифротексте
            origCase = []
            for i in range(len(ciphertext)):
                if ciphertext[i].isupper():
                    origCase.append(decryptedText[i].upper())
                else:
                    origCase.append(decryptedText[i].lower())
            decryptedText = ''.join(origCase)

            # Спросить у пользователя, найден ли ключ дешифрования
            print('Possible encryption hack with key %s:' %
                  (possibleKey))
            print(decryptedText[:200])  # выводим первые 200 символов
            print()
            print('Enter D if done, anything else to continue
                  hacking:')
            response = input('> ')

            if response.strip().upper().startswith('D'):
                return decryptedText
```

```
218.
219.     # Получить осмысленный текст не удалось, поэтому возвращаем None
220.     return None
221.
222.
223. def hackVigenere(ciphertext):
224.     # Прежде всего, необходимо применить метод Касиски
225.     # для выяснения возможной длины ключа
226.     allLikelyKeyLengths = kasiskiExamination(ciphertext)
227.     if not SILENT_MODE:
228.         keyLengthStr = ''
229.         for keyLength in allLikelyKeyLengths:
230.             keyLengthStr += '%s ' % (keyLength)
231.         print('Kasiski examination results say the most likely
                key lengths are: ' + keyLengthStr + '\n')
232.     hackedMessage = None
233.     for keyLength in allLikelyKeyLengths:
234.         if not SILENT_MODE:
235.             print('Attempting hack with key length %s
                  (%s possible keys) ...' % (keyLength,
                  NUM_MOST_FREQ_LETTERS ** keyLength))
236.         hackedMessage = attemptHackWithKeyLength(ciphertext, keyLength)
237.         if hackedMessage != None:
238.             break
239.
240.     # Если ни один из найденных с помощью метода Касиски вариантов
241.     # длины ключа не сработал, начать атаку методом грубой силы
242.     if hackedMessage == None:
243.         if not SILENT_MODE:
244.             print('Unable to hack message with likely key length(s).
                    Brute-forcing key length...')
245.         for keyLength in range(1, MAX_KEY_LENGTH + 1):
246.             # Не перепроверять длину ключа, уже опробованную
                  методом Касиски
247.             if keyLength not in allLikelyKeyLengths:
248.                 If not SILENT_MODE:
249.                     print('Attempting hack with key length %s
                          (%s possible keys)...' % (keyLength,
                          NUM_MOST_FREQ_LETTERS ** keyLength))
250.             hackedMessage = attemptHackWithKeyLength(ciphertext,
                  keyLength)
251.             if hackedMessage != None:
252.                 break
253.     return hackedMessage
254.
255.
256. # Если файл vigenereHacker.py выполняется как программа
257. # (а не импортируется как модуль), вызвать функцию main():
258. if __name__ == '__main__':
259.     main()
```

Пример выполнения программы Vigenere Hacker

Выполнив программу, вы должны получить примерно следующие результаты.

```
Kasiski examination results say the most likely key lengths are: 3 2 6 4
12 8 9 16 5 11 10 15 7 14 13

Attempting hack with key length 3 (64 possible keys)...
Possible letters for letter 1 of the key: A L M E
Possible letters for letter 2 of the key: S N O C
Possible letters for letter 3 of the key: V I Z B
Attempting with key: ASV
Attempting with key: ASI
--опущено--
Attempting with key: ECZ
Attempting with key: ECB
Attempting hack with key length 2 (16 possible keys)...
Possible letters for letter 1 of the key: O A E Z
Possible letters for letter 2 of the key: M S I D
Attempting with key: OM
Attempting with key: OS
--опущено--
Attempting with key: ZI
Attempting with key: ZD
Attempting hack with key length 6 (4096 possible keys)...
Possible letters for letter 1 of the key: A E O P
Possible letters for letter 2 of the key: S D G H
Possible letters for letter 3 of the key: I V X B
Possible letters for letter 4 of the key: M Z Q A
Possible letters for letter 5 of the key: O B Z A
Possible letters for letter 6 of the key: V I K Z
Attempting with key: ASIMOV
Possible encryption hack with key ASIMOV:
Alan Mathison Turing was a British mathematician, logician, cryptanalyst, and
computer scientist. He was highly influential in the development of computer
science, providing a formalisation of the con

Enter D if done, anything else to continue hacking:
> d
Copying hacked message to clipboard:
Alan Mathison Turing was a British mathematician, logician, cryptanalyst, and
computer scientist. He was highly influential in the development of computer
--опущено--
```

Импорт модулей и функция `main()`

Рассмотрим исходный код программы для взлома шифра Виженера. Данная программа импортирует различные модули, в том числе и новый модуль `itertools`, о котором вскоре будет рассказано более подробно.

```
1. # Программа взлома шифра Виженера
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import itertools, re
5. import vigenereCipher, pyperclip, freqAnalysis, detectEnglish
```

В строках 7–11 задается несколько констант, о назначении которых мы поговорим, когда будем обсуждать код, в котором они используются.

Функция `main()` напоминает аналогичные функции предыдущих программ взлома.

```
14. def main():
15.     # Этот шифротекст можно скопировать из
16.     # файла исходного кода (см. введение)
17.     ciphertext = """Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi,
          lgouqdaf, kdmktsvmztsl, izr xoexghzr kkusitaaf. Vz wsa twbhdg
          ubalmmzhdad qz
           --опущено--
          azmtmd'g widt ion bwnafz tzm Tcpsw wr Zjrva ivdcz eaigd yzmbo
          Tmzubb a kbmhptgzk dvrvwz wa efiohzd."""
18.       hackedMessage = hackVigenere(ciphertext)
19.
20.     if hackedMessage != None:
21.         print('Copying hacked message to clipboard:')
22.         print(hackedMessage)
23.         pyperclip.copy(hackedMessage)
24.     else:
25.         print('Failed to hack encryption.')
```

Шифротекст передается функции `hackVigenere()`, которая возвращает либо дешифрованную строку, если попытка взлома оказалась успешной, либо значение `None` в случае неудачи. В случае успеха программа выводит взломанное сообщение на экран и копирует его в буфер обмена.

Нахождение повторяющихся последовательностей

Функция `findRepeatSequencesSpacings()` реализует первый этап метода Касиски, обнаруживая все повторяющиеся последовательности букв в строке сообщения и вычисляя длину интервалов между ними.

```
28. def findRepeatSequencesSpacings(message):
        --опущено--
33.     # Используем регулярное выражение для удаления небуквенных символов
34.     message = NONLETTERS_PATTERN.sub('', message.upper())
35.
36.     # Получение списка последовательностей, найденных в сообщении
37.     seqSpacings = {}  # ключи - последовательности, значения -
                          списки интервалов повторения
```

В строке 34 сообщение преобразуется в верхний регистр, и с помощью метода `sub()` из него удаляются любые небуквенные символы.

Словарь `seqSpacings`, создаваемый в строке 37, предназначен для хранения повторяющихся последовательностей в качестве ключей. Значениями словаря являются списки целочисленных значений, соответствующих количеству букв между повторными вхождениями данной последовательности. Например, если в параметре `message` будет передана наша тестовая строка `'PPQCAXQV...'`, то функция `findRepeatSequenceSpacings()` вернет словарь `{'VRA': [8, 24, 32], 'AZU': [48], 'YBN': [8]}`.

В цикле `for`, который начинается в строке 38, перебираются все последовательности, встречающиеся в строке, и выявляются повторы.

```
38.     for seqLen in range(3, 6):
39.         for seqStart in range(len(message) - seqLen):
40.             # Получение очередной последовательности
41.             seq = message[seqStart:seqStart + seqLen]
```

На первой итерации цикла ищутся повторяющиеся последовательности, которые содержат ровно три буквы. На следующей итерации ищутся последовательности из четырех букв и т.д. Изменяя границы диапазона `(3, 6)` в строке 38, вы сможете управлять тем, последовательности какой длины будет проверять программа. В то же время в большинстве шифротекстов вполне можно ограничиться поиском повторяющихся последовательностей длиной три, четыре и пять букв. Дело в том, что, с одной стороны, такие последовательности достаточно длинные для того, чтобы не считать повторения случайными, а с другой стороны — достаточно короткие, что увеличивает вероятность их повторений. Текущая длина последовательности, проверяемая в цикле `for`, хранится в переменной `seqLen`.

Во вложенном цикле `for`, который начинается в строке 39, с помощью срезов из строки `message` извлекаются все возможные подстроки длиной `seqLen`. Начало среза хранится в переменной `seqStart`. Например, если переменная `seqLen` равна 3, а параметр `message` содержит

строку 'PPQCAXQ', то на первой итерации (seqStart = 0) извлекаются первые три символа строки, и мы получаем подстроку 'PPQ'. Следующим значением seqStart будет 1, и мы извлекаем подстроку 'PQC'. Эти действия повторяются для всех последующих индексов, пока не будут извлечены последние три символа строки, для которых индекс начала среза определяется выражением len(message) - seqLen. В результате мы получаем последовательности, приведенные на рис. 20.2.

Рис. 20.2. *Значения переменной* `seq` *в зависимости от значения* `seqStart`

Цикл выполняется для всех индексов вплоть до len(message) - seqLen, а текущий индекс начала среза присваивается переменной seqStart. В строке 41 полученный срез записывается в переменную seq.

В строке 44 начинается еще один вложенный цикл for, в котором в строке сообщения ищутся повторные вхождения данного среза.

```
43.             # Поиск этой последовательности в остальной части сообщения
44.             for i in range(seqStart + seqLen, len(message) - seqLen):
45.                 if message[i:i + seqLen] == seq:
```

Переменная i поочередно принимает значения индексов всех возможных последовательностей длиной seqLen, содержащихся в строке message. Индексы изменяются в диапазоне от seqStart + seqLen (позиция сразу за концом текущей последовательности seq) и до len(message) - seqLen (последний индекс, при котором еще может быть обнаружена последовательность длиной seqLen). Например, если параметр message содержит строку 'PPQCAXQVEKGYBNKMAZUYBN', переменная seqStart равна 11, а переменная seqLen равна 3, то в строке 41 в переменную seq будет записано значение 'YBN', и в цикле for сообщение будет просматриваться начиная с индекса 14.

Выражение message[i:i + seqLen] в строке 45 возвращает подстроку, которая сравнивается с переменной seq, чтобы выяснить, не является ли она ее повторением. Если обнаружен повтор, то в строках 46–52 вычисляется интервал повторения, который добавляется в словарь seqSpacings. На первой итерации цикла переменная seq в строке 45 сравнивается с подстрокой 'KMA', на второй — с подстрокой 'MAZ', затем — с подстрокой 'AZU' и т.д. Когда переменная i достигнет значения 19, в строке 45 обнаружится, что подстрока 'YBN' совпадает с переменной seq, и тогда будут выполнены строки с 46 по 52.

```
46.                     # Найдена повторяющаяся последовательность
47.                     if seq not in seqSpacings:
48.                         seqSpacings[seq] = []  # пустой список
49.
50.                         # Добавить интервал повторения между исходной
51.                         # и повторившейся последовательностями
52.                         seqSpacings[seq].append(i - seqStart)
```

В строке 47 проверяется, содержится ли подстрока `seq` в качестве ключа в словаре `seqSpacings`. Если такого ключа еще нет, то в строке 48 он добавляется в словарь, а его значением становится пустой список.

Интервал между последовательностью `message[i:i + seqLen]` и исходной последовательностью `message[seqStart:seqStart + seqLen]` равен разности `i - seqStart`. Обратите внимание на то, что `i` и `seqStart` — это начальные индексы срезов, указанные перед двоеточием. Следовательно, выражение `i - seqStart` дает нам количество букв, разделяющих обе последовательности, и мы присоединяем это значение к списку, хранящемуся в позиции `seqSpacings[seq]`.

По завершении всех циклов словарь `seqSpacings` будет содержать все повторяющиеся последовательности длиной 3, 4 и 5 букв, а также значения всех интервалов между ними. Словарь `seqSpacings` возвращается функцией `findRepeatSequencesSpacings()` в строке 53:

```
53.     return seqSpacings
```

Итак, вы узнали, как реализуется первый этап метода Касиски, на котором мы находим повторяющиеся последовательности букв в шифротексте и подсчитываем количество букв между ними. Теперь можно перейти к следующему этапу.

Вычисление множителей интервалов повторения

Вспомните, что следующий этап метода Касиски заключается в нахождении множителей интервалов повторения. Мы ищем множители, выражаемые числами в диапазоне от 2 до `MAX_KEY_LENGTH`. Для этого предназначена функция `getUsefulFactors()`, которая получает параметр `num` и возвращает список, содержащий только множители, удовлетворяющие этому критерию.

```
56. def getUsefulFactors(num):
        --опущено--
61.     if num < 2:
```

```
        return []  # числа, меньшие 2, не имеют полезных множителей

    factors = []  # список найденных множителей
```

В строке 61 проверяется особая ситуация, когда значение num меньше 2. В таком случае в строке 62 возвращается пустой список, поскольку параметр num не имеет полезных множителей.

Если же значение num больше 2, то необходимо вычислить все множители num и сохранить их в виде списка. В строке 64 создается пустой список factors, предназначенный для хранения таких множителей.

В цикле for, который начинается в строке 68, перебираются целые числа в диапазоне от 2 до MAX_KEY_LENGTH включительно. Вспомните, что функция range() возвращает диапазон, не включающий верхнее граничное значение, поэтому мы передаем ей MAX_KEY_LENGTH + 1 в качестве второго аргумента. Цикл предназначен для нахождения всех множителей числа num.

```
    for i in range(2, MAX_KEY_LENGTH + 1):  # множитель 1 бесполезен
        if num % i == 0:
            factors.append(i)
            otherFactor = int(num / i)
```

В строке 69 проверяется равенство выражения num % i нулю. Если это условие выполняется, то мы знаем, что num делится на i без остатка, а значит, i является множителем num. В таком случае в строке 70 значение i присоединяется к списку множителей, хранящемуся в переменной factors. Поскольку частное num / i тоже является множителем числа num, его целочисленная форма сохраняется в переменной otherFactor в строке 71. (Вспомните, что результатом операции / всегда будет число с плавающей точкой. Например, выражение 21 / 7 возвращает не целое число 3, а вещественное число 3.0.) Если результирующее значение равно 1, то оно не включается в список factors; этот случай проверяется в строке 72.

```
            if otherFactor < MAX_KEY_LENGTH + 1 and otherFactor != 1:
                factors.append(otherFactor)
```

Мы исключаем значение 1 по той причине, что в случае ключа длиной 1 шифр Виженера ничем не отличается от шифра Цезаря.

Удаление дубликатов с помощью функции set()

В методе Касиски мы должны определить наиболее часто встречающийся множитель интервалов повторения, поскольку длина ключа Виженера,

скорее всего, будет равна именно ему. Но прежде чем мы сможем приступить к анализу частотности каждого множителя, необходимо воспользоваться функцией `set()` для удаления дубликатов множителей из списка `factors`. Например, если передать функции `getUsefulFactors()` аргумент `9`, то проверка `9 % 3 == 0` вернет значение `True`, и к списку `factors` будет присоединено как значение `i`, так и значение `int(num / i)`. Но оба выражения равны `3`, поэтому `3` будет включено в список дважды. Во избежание появления дубликатов мы передаем список функции `set()`, которая преобразует список в *множество*. Тип данных `set` аналогичен типу данных `list`, за исключением того, что множество может включать лишь уникальные значения.

Передав список функции `set()`, мы получим множество, содержащее лишь уникальные элементы. В то же время, если передать множество функции `list()`, мы преобразуем его в список. Чтобы убедиться в этом, введите в интерактивной оболочке следующие инструкции.

```
>>> set([1, 2, 3, 3, 4])
{1, 2, 3, 4}
>>> spam = list(set([2, 2, 2, 'cats', 2, 2]))
>>> spam
[2, 'cats']
```

Преобразование списка в множество приводит к удалению из него дубликатов. После обратного преобразования множества в список последний тоже не будет содержать повторяющихся элементов.

Удаление дублирующихся множителей и сортировка списка

В строке 74 список `factors` передается функции `set()` для удаления дубликатов:

```
74.     return list(set(factors))  # удаляем дубликаты
```

Функция `getItemAtIndexOne()` (строка 77) почти идентична функции `getItemAtIndexZero()` программы *freqAnalysis.py*, которую мы написали в главе 19.

```
77. def getItemAtIndexOne(x):
78.     return x[1]
```

Впоследствии эта функция будет передаваться методу `sort()` для сортировки списка на основании элемента с индексом 1.

Нахождение наиболее часто встречающихся множителей

Для нахождения наиболее часто встречающихся множителей, являющихся самыми вероятными значениями длины ключа, нам понадобится функция `getMostCommonFactors()`, определяемая в строке 81.

```
81. def getMostCommonFactors(seqFactors):
82.     # Во-первых, подсчитываем повторы множителей в словаре seqFactors
83.     factorCounts = {}  # ключ - множитель; значение - число повторений
```

Параметр `seqFactors` — это словарь, созданный с помощью функции `kasiskiExamination()`, которую мы рассмотрим далее. В данном словаре ключами служат цепочки букв, а значениями — списки множителей интервалов повторения, возвращаемые функцией `findRepeatSequencesSpacings()`. Например, переменная `seqFactors` может содержать словарь следующего вида.

```
{'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]}
```

Функция `getMostCommonFactors()` сортирует множители, содержащиеся в словаре `seqFactors`, в порядке убывания частотности и возвращает список кортежей, включающих два целочисленных элемента. Первым элементом кортежа является множитель, вторым — количество вхождений этого множителя в словаре `seqFactors`.

Например, функция `getMostCommonFactors()` может вернуть такой список.

```
[(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)]
```

Как следует из этого списка, в словаре `seqFactors`, который был передан функции `getMostCommonFactors()`, множитель 3 встречается 556 раз, множитель 2 — 541 раз, множитель 6 — 529 раз и т.д. Заметьте, что первым в списке появляется множитель 3, поскольку он встречается чаще других. Множитель 13 встречается реже, чем любой другой множитель, потому он оказался последним в списке.

В функции `getMostCommonFactors()` сначала создается пустой словарь `factorCounts` (строка 83), который будет использоваться для хранения счетчиков числа вхождений каждого множителя. В данном словаре ключами будут множители, а значениями — счетчики этих множителей.

В цикле for, который начинается в строке 88, функция проходит по всем последовательностям, содержащимся в словаре seqFactors, сохраняя каждую из них в переменной seq на каждой итерации. Список множителей, содержащийся в словаре seqFactors для последовательности seq, сохраняется в переменной factorList (строка 89).

```
 88.     for seq in seqFactors:
 89.         factorList = seqFactors[seq]
 90.         for factor in factorList:
 91.             if factor not in factorCounts:
 92.                 factorCounts[factor] = 0
 93.             factorCounts[factor] += 1
```

Все множители из этого списка просматриваются во вложенном цикле for, который начинается в строке 90. Если множитель не существует в качестве ключа в словаре factorCounts, то он добавляется в него в строке 92 со значением 0. В противном случае значение, соответствующее данному ключу в словаре factorCounts, инкрементируется в строке 93.

Далее необходимо отсортировать множители, содержащиеся в словаре factorCounts, в соответствии со значениями их счетчиков. Но поскольку понятие упорядоченности неприменимо к словарям, мы должны преобразовать словарь в список кортежей, каждый из которых содержит два целочисленных элемента. (Аналогичное преобразование мы применяли в функции getFrequencyOrder() программы *freqAnalaysis.py* в главе 19.) Список будет храниться в переменной factorsByCount, которая инициализируется в строке 97.

```
 97.     factorsByCount = []
 98.     for factor in factorCounts:
 99.         # Исключаем множители, которые больше, чем MAX_KEY_LENGTH
100.         if factor <= MAX_KEY_LENGTH:
101.             # factorsByCount - список кортежей (множитель, счетчик).
102.             # Типичный вид: [(3, 497), (2, 487), ...]
103.             factorsByCount.append( (factor, factorCounts[factor]) )
```

В цикле for, который начинается в строке 98, программа проходит по всем множителям списка factorCounts и присоединяет кортеж (factor, factorCounts[factor]) к списку factorsByCount лишь в том случае, если множитель меньше или равен MAX_KEY_LENGTH.

После того как все кортежи добавлены в список factorsByCount, выполняется последний этап функции getMostCommonFactors() — сортировка полученного списка в строке 106.

```
106.     factorsByCount.sort(key=getItemAtIndexOne, reverse=True)
107.
108.     return factorsByCount
```

Поскольку в качестве аргумента `key` передается функция `getItemAtIndexOne`, а в качестве аргумента `reverse` — значение `True`, список сортируется в порядке убывания счетчиков множителей. В строке 108 возвращается отсортированный список `factorsByCount`, содержащий информацию о том, какие множители встречаются чаще всего и потому являются наиболее вероятными вариантами длины ключа Виженера.

Нахождение наиболее вероятной длины ключа

Прежде чем начинать искать возможные подключи, необходимо выяснить, сколько всего подключей должно быть, т.е. какова длина ключа. Функция `kasiskiExamination()` возвращает список наиболее вероятных вариантов длины ключа для заданного аргумента `ciphertext`.

```
111. def kasiskiExamination(ciphertext):
         --опущено--
115.     repeatedSeqSpacings = findRepeatSequencesSpacings(ciphertext)
```

Возможные значения длины ключа будут представлены списком целых чисел. Первое число в списке — это наиболее вероятная длина ключа, второе — следующая по вероятности длина ключа и т.д.

Первым шагом является нахождение интервалов между повторяющимися последовательностями в шифротексте. Эта информация возвращается функцией `findRepeatSequencesSpacings()` в виде словаря, в котором ключами являются строковые последовательности, а значениями — списки с целочисленными значениями интервалов повторения. Функция `findRepeatSequencesSpacings()` была рассмотрена в разделе "Нахождение повторяющихся последовательностей".

Прежде чем двигаться дальше, необходимо познакомиться со списковым методом `extend()`.

Списковый метод `extend()`

Для того чтобы добавить сразу несколько значений в конец списка, не обязательно вызывать метод `append()` в цикле. Даже при получении аргумента-списка метод `append()` делает не то, что нам нужно: он добавляет его в качестве единого элемента в конец другого списка.

```
>>> spam = ['cat', 'dog', 'mouse']
>>> eggs = [1, 2, 3]
>>> spam.append(eggs)
>>> spam
['cat', 'dog', 'mouse', [1, 2, 3]]
```

В отличие от этого метод `extend()` добавляет в конец списка каждый из элементов списка, переданного ему в качестве аргумента. Введите в интерактивной оболочке следующие инструкции.

```
>>> spam = ['cat', 'dog', 'mouse']
>>> eggs = [1, 2, 3]
>>> spam.extend(eggs)
>>> spam
['cat', 'dog', 'mouse', 1, 2, 3]
```

Как видите, все значения из списка `eggs` (`1`, `2` и `3`) присоединяются к списку `spam` как отдельные элементы.

Расширение словаря `repeatedSeqSpacings`

Словарь `repeatedSeqSpacings` сопоставляет последовательности строк со списками целочисленных интервалов повторения, но фактически нам нужен словарь, в котором последовательности строк сопоставляются со списками множителей данных интервалов. (Чтобы понять, почему это так, см. раздел "Определение множителей в интервалах повторения".) Данная задача решается в строках 118–122.

```
118.     seqFactors = {}
119.     for seq in repeatedSeqSpacings:
120.         seqFactors[seq] = []
121.         for spacing in repeatedSeqSpacings[seq]:
122.             seqFactors[seq].extend(getUsefulFactors(spacing))
```

В строке 118 создается пустой словарь `seqFactors`. В цикле `for`, который начинается в строке 119, программа проходит по всем ключам словаря `repeatedSeqSpacings`, представляющим собой строковые последовательности. В строке 120 для каждого ключа создается пустой список, который будет храниться в словаре `seqFactors`.

Во вложенном цикле `for` (строки 121–122) программа проходит по всем интервалам повторения, передавая каждый из них функции `getUsefulFactors()`. Каждый элемент списка, возвращаемый этой функцией, добавляется в качестве значения ключа `seqFactors[seq]` с помощью метода `extend()`. По завершении циклов переменная `seqFactors` будет содер-

жать словарь, сопоставляющий строковые последовательности со списками множителей интервалов повторения. Благодаря этому мы получаем информацию не только об интервалах повторения, но и об их множителях.

В строке 125 словарь `seqFactors` передается функции `getMostCommonFactors()`, возвращающей список кортежей из двух целых чисел, первое из которых представляет множитель, а второе — количество появлений этого множителя в словаре `seqFactors`. Данный список сохраняется в переменной `factorsByCount`.

```
125. factorsByCount = getMostCommonFactors(seqFactors)
```

Но нам все-таки нужно, чтобы функция `kasiskiExamination()` возвращала список множителей, а не список кортежей. Поскольку множители хранятся в первых элементах кортежей списка `factorsByCount`, мы должны извлечь их из кортежей и сформировать из них отдельный список.

Извлечение множителей из списка `factorsByCount`

В строках 130–134 список множителей сохраняется в переменной `allLikelyKeyLengths`.

```
130.     allLikelyKeyLengths = []
131.     for twoIntTuple in factorsByCount:
132.         allLikelyKeyLengths.append(twoIntTuple[0])
133.
134.     return allLikelyKeyLengths
```

В цикле `for`, который начинается в строке 131, программа проходит по всем кортежам списка `factorsByCount` и добавляет элемент кортежа с индексом `0` в конец списка `allLikelyKeyLengths`. По завершении цикла переменная `allLikelyKeyLengths` будет содержать все целочисленные множители из словаря `factorsByCount`, которые возвращаются в виде списка функцией `kasiskiExamination()`.

Теперь мы можем определить наиболее вероятную длину ключа, с помощью которого было зашифровано сообщение, но нам еще нужно иметь возможность выделить из сообщения буквы, зашифрованные одним и тем же подключом. Вспомните: в случае шифрования сообщения `'THEDOGANDTHECAT'` ключом `'XYZ'` подключ `'X'` будет применяться для шифрования букв сообщения с индексами 0, 3, 6, 9 и 12. Поскольку буквы исходного сообщения шифруются одним и тем же подключом (`'X'`), частотность этих букв в дешифрованном тексте должна быть близка к частотности букв в типичных английских текстах. Данную информацию можно задействовать для нахождения подключа.

Получение букв, зашифрованных одним и тем же подключом

Чтобы извлечь из шифротекста буквы, зашифрованные одним и тем же подключом, необходимо написать функцию, которая создает строку, используя 1-е, 2-е или *n*-е буквы сообщения. Первое, что должна сделать такая функция, получив начальный индекс, длину ключа и сообщение, — это удалить из сообщения небуквенные символы, используя объект регулярного выражения и метод `sub()` (строка 145).

Примечание

Регулярные выражения обсуждались в главе 17.

Строка, содержащая лишь буквы, сохраняется в виде нового значения переменной `message`.

```
137. def getNthSubkeysLetters(nth, keyLength, message):
         --опущено--
145.     message = NONLETTERS_PATTERN.sub('', message)
```

Далее мы формируем список, присоединяя к нему буквы, а затем используем метод `join()` для объединения элементов списка в одну строку.

```
147.     i = nth - 1
148.     letters = []
149.     while i < len(message):
150.         letters.append(message[i])
151.         i += keyLength
152.     return ''.join(letters)
```

Переменная `i` задает индекс той буквы в строке `message`, которую требуется присоединить к списку `letters`. Начальным значением переменной `i` является `nth - 1` (строка 147), а начальным значением переменной `letters` — пустой список (строка 148).

Цикл `while`, начинающийся в строке 149, продолжает выполняться до тех пор, пока значение `i` остается меньшим, чем длина строки `message`. На каждой итерации цикла буква `message[i]` добавляется в список `letters`. Затем значение `i` увеличивается на `keyLength` в строке 151, что позволяет перейти к следующей букве подключа.

По завершении цикла однобуквенные строки, содержащиеся в списке `letters`, объединяются в одну строку, и эта строка возвращается функцией `getNthSubkeysLetters()`.

Теперь, когда мы научились извлекать буквы, зашифрованные одним и тем же подключом, мы сможем использовать функцию `getNthSubkeysLetters()` для того, чтобы попытаться дешифровать сообщение с помощью ключей наиболее вероятной длины.

Попытки дешифрования с помощью ключей вероятной длины

Вспомните о том, что функция `kasiskiExamination()` возвращает не гарантированно верную длину ключа Виженера, а список, включающий несколько возможных вариантов, отсортированных в порядке убывания их вероятности. Если длина ключа была определена неверно, программа попытается использовать ключ другой длины. Эту работу выполняет функция `attemptHackWithKeyLength()`, которая получает шифротекст и предполагаемую длину ключа. В случае успеха функция возвращает строку взломанного сообщения, а в случае неудачи — значение `None`.

Процедура взлома применяется к буквам, преобразованным в верхний регистр, но мы хотим, чтобы любая дешифрованная строка возвращалась с соблюдением исходного регистра, а для этого нужно сохранить строку в ее первоначальном виде. Чтобы обеспечить это, мы сохраняем строку шифротекста, буквы которой преобразованы в верхний регистр, в переменной `ciphertextUp` (строка 157).

```
155. def attemptHackWithKeyLength(ciphertext, mostLikelyKeyLength):
156.     # Определяем наиболее вероятные буквы для каждого подключа
157.     ciphertextUp = ciphertext.upper()
```

Исходя из предположения, что значение `mostLikelyKeyLength` представляет собой корректную длину ключа, программа взлома вызывает функцию `getNthSubkeysLetters()` для каждого подключа, а затем применяет метод грубой силы, выполняя полный перебор всех 26 возможных букв для нахождения той из них, которая приводит к получению дешифрованного текста с частотностью, наиболее близко соответствующей частотности букв в типичных английских текстах.

Сначала в строке 160 создается пустой список `allFreqScores`, в котором будут храниться оценки частотного соответствия, возвращаемые функцией `freqAnalysis.englishFreqMatchScore()`.

```
160.     allFreqScores = []
161.     for nth in range(1, mostLikelyKeyLength + 1):
162.         nthLetters = getNthSubkeysLetters(nth, mostLikelyKeyLength,
               ciphertextUp)
```

Переменная nth в цикле for, который начинается в строке 161, поочередно принимает значения от 1 до mostLikelyKeyLength. Вспомните, что диапазон, возвращаемый функцией range(), не включает верхнюю границу, задаваемую вторым аргументом. В связи с этим мы увеличиваем верхнюю границу диапазона на 1, чтобы охватить значение mostLikelyKeyLength.

Буквы *n*-го подключа возвращаются функцией getNthSubkeysLetters() в строке 162.

Далее необходимо дешифровать буквы *n*-го подключа, используя все 26 возможных подключей, чтобы увидеть, какой из них приводит к частотности букв, близкой к частотности в типичных английских текстах. Список оценок частотного соответствия будет храниться в переменной freqScores, которая инициализируется пустым списком в строке 168. В цикле for, который начинается в строке 169, программа проходит по всем 26 буквам строки LETTERS.

```
168.         freqScores = []
169.         for possibleKey in LETTERS:
170.             decryptedText = vigenereCipher.decryptMessage(possibleKey,
                   nthLetters)
```

Значение possibleKey используется для дешифрования шифротекста путем вызова функции vigenereCipher.decryptMessage() в строке 170. Подключ, содержащийся в переменной possibleKey, является однобуквенным, тогда как строка nthLetters формируется только из тех букв шифротекста, которые могут быть зашифрованы данным подключом, если программа правильно определила длину ключа.

Далее дешифрованный текст передается функции freqAnalysis.englishFreqMatchScore(), чтобы проверить, насколько близко частотность букв в строке decryptedText соответствует частотности букв в типичных английских текстах. Вспомните из главы 19, что данная функция возвращает целое число в интервале от 0 до 12, причем чем больше это значение, тем ближе соответствие.

В строке 171 полученная оценка частотного соответствия и подключ, использованный для дешифрования, заносятся в переменную keyAndFreqMatchTuple. В строке 172 этот кортеж добавляется в конец списка freqScores.

```
171.             keyAndFreqMatchTuple = (possibleKey,
                   freqAnalysis.englishFreqMatchScore(decryptedText))
172.             freqScores.append(keyAndFreqMatchTuple)
```

Когда цикл `for`, начатый в строке 169, завершится, список `freqScores` будет содержать 26 кортежей, представляющих собой пары "ключ, оценка частотного соответствия" — по одному кортежу для каждого из 26 подключей. Список должен быть отсортирован таким образом, чтобы первыми шли кортежи с наибольшими оценками. Это означает, что кортежи необходимо отсортировать по значениям с индексом 1 и в обратном порядке (по убыванию).

Мы применяем к списку `freqScores` метод `sort()`, передавая ему функцию `getItemAtIndexOne()` в качестве аргумента `key`. (Отсутствие круглых скобок после имени функции означает, что она не вызывается, а передается как значение.) Аргумент `reverse` равен `True`, что означает сортировку по убыванию.

```
174.         freqScores.sort(key=getItemAtIndexOne, reverse=True)
```

В строке 9 для константы `NUM_MOST_FREQ_LETTERS` было задано значение 4. После того как кортежи, содержащиеся в списке `freqScores`, были отсортированы по убыванию, в строке 176 в переменную `allFreqScores` добавляется список, содержащий только первые четыре кортежа, т.е. кортежи с четырьмя наиболее высокими оценками частотного соответствия. Таким образом, элемент `allFreqScores[0]` содержит оценки для первого подключа, элемент `allFreqScores[1]` — оценки для второго подключа и т.д.

```
176.         allFreqScores.append(freqScores[:NUM_MOST_FREQ_LETTERS])
```

Когда цикл `for`, начатый в строке 161, завершится, переменная `allFreqScores` будет содержать списки в количестве, равном `mostLikelyKeyLength`. Например, если параметр `mostLikelyKeyLength` равен 3, то переменная `allFreqScores` будет включать три списка. Первый список содержит кортежи для первых четырех букв с наивысшими оценками частотного соответствия, относящихся к первому подключу полного ключа шифра Виженера. Второй список содержит аналогичные кортежи, относящиеся ко второму подключу полного ключа шифра Виженера, и т.д.

Первоначально, если бы мы хотели применить метод грубой силы к полному ключу шифра Виженера, то количество возможных ключей было бы равно 26 в степени, равной длине ключа. Например, в случае ключа `ROSEBUD` длиной 7 букв мы имели бы 26^7, или 8 031 810 176, возможных ключей.

Однако проверка частотного соответствия букв помогает определить четыре наиболее вероятные буквы для каждого подключа. Применительно к примеру с ключом `ROSEBUD` это означает, что необходимо проверить только

4^7, или 16 384, возможных ключей, что является огромным шагом вперед по сравнению с прежним их количеством, составляющим около 8 млрд!

Аргумент `end` функции `print()`

Следующее, что необходимо сделать, — вывести информацию для пользователя. Для этого мы воспользуемся функцией `print()`, но передадим ей необязательный аргумент, с которым ранее не сталкивались. Всякий раз, когда вызывается функция `print()`, она выводит на экран переданную ей строку вместе с завершающим символом новой строки. Мы можем отменить переход на новую строку и вывести другую информацию в текущей строке, указав аргумент `end` при вызове функции `print()`. Чтобы увидеть, как это работает, введите в интерактивной оболочке показанный ниже код.

```
>>> def printStuff():
❶ ...     print('Hello', end='\n')
❷ ...     print('Howdy', end='')
❸ ...     print('Greetings', end='XYZ')
...     print('Goodbye')
...
>>> printStuff()
Hello
HowdyGreetingsXYZGoodbye
```

Передача аргумента `end='\n'` (❶) эквивалентна обычному вызову функции. Однако передача аргумента `end=''` (❷) или `end='XYZ'` (❸) приводит к замене символа новой строки, поэтому последующие вызовы функции `print()` отображают результаты в текущей строке.

Запуск программы в "тихом" режиме или с выводом информации для пользователя

На данном этапе мы хотим знать, какие буквы являются четырьмя наиболее вероятными кандидатами для каждого из подключей. Если ранее в программе для константы `SILENT_MODE` было установлено значение `False`, то содержимое переменной `allFreqScores` будет выведено на экран.

```
178.     if not SILENT_MODE:
179.         for i in range(len(allFreqScores)):
180.             # Используем i + 1, чтобы первая буква не считалась 0-й
181.             print('Possible letters for letter %s of the key:' %
                       (i + 1), end='')
182.             for freqScore in allFreqScores[i]:
183.                 print('%s ' % freqScore[0], end='')
184.             print()  # переход на новую строку
```

Если же для константы SILENT_MODE было установлено значение True, то блок инструкции if будет пропущен.

На данном этапе нам удалось уменьшить количество возможных подключей до уровня, позволяющего применить к ним метод грубой силы. Далее вы узнаете о том, каким образом можно сгенерировать все возможные комбинации подключей с помощью функции itertools.product() для их последующего перебора методом грубой силы.

Нахождение возможных комбинаций подключей

Теперь, когда у нас имеются все возможные подключи, мы должны объединить их для формирования целого ключа. Проблема в том, что даже наиболее вероятные буквы могут оказаться некорректными. В реальности корректной может быть вторая или третья буква из числа наиболее вероятных. Это означает, что мы не можем ограничиться простым объединением наиболее вероятных букв каждого подключа в единый ключ. Для нахождения истинного ключа мы должны испытать все возможные комбинации.

Для тестирования всех возможных комбинаций подключей в программе *vigenereHacker.py* применяется функция itertools.product().

Функция itertools.product()

Функция itertools.product() создает все возможные комбинации элементов списка или значений схожего типа, такого как строка или кортеж. Подобную комбинацию элементов называют *декартовым произведением*, что и дало название функции. Данная функция возвращает объект, который можно преобразовать в список, передав его функции list(). Чтобы увидеть, как это работает, введите в интерактивной оболочке следующий код.

```
>>> import itertools
>>> itertools.product('ABC', repeat=4)
❶ <itertools.product object at 0x02C40170>
>>> list(itertools.product('ABC', repeat=4))
[('A', 'A', 'A', 'A'), ('A', 'A', 'A', 'B'), ('A', 'A', 'A', 'C'),
 ('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'),
--опущено--
 ('C', 'B', 'C', 'B'), ('C', 'B', 'C', 'C'), ('C', 'C', 'A', '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')]
```

Получив строку 'ABC' и целое число 4 в качестве аргумента repeat, функция itertools.product() возвращает объект декартового произведения (❶), который после преобразования в список содержит кортежи из четырех значений, представляющих возможные комбинации букв 'A', 'B' и 'C'. В конечном итоге мы получаем список, содержащий в общей сложности 3^4, или 81, кортеж.

Также допускается передавать в функцию itertools.product() списки и им подобные значения, например объекты range, возвращаемые функцией range(). Чтобы увидеть это на конкретном примере, введите в интерактивной оболочке следующий код.

```
>>> 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),
--опущено--
(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)]
```

Получив объект диапазона, создаваемый вызовом range(8), а также число 5 в качестве аргумента repeat, функция itertools.product() возвращает список, элементами которого являются кортежи из пяти значений, представляющих собой целые числа в диапазоне от 0 до 7.

Мы не можем просто передать список потенциальных подключей в функцию itertools.product(), так как она создает комбинации одних и тех же значений, а для каждого из подключей, вероятнее всего, будут существовать разные варианты потенциальных букв. Вместо этого, поскольку наши подключи хранятся в виде кортежей в списке allFreqScores, мы будем получать доступ к буквам с помощью индексов, имеющих значения от 0 до желаемого количества букв минус 1. Мы знаем, что количество букв в каждом кортеже равно NUM_MOST_FREQ_LETTERS, поскольку именно это значение задавалось для каждого кортежа в строке 176. Соответственно, константу NUM_MOST_FREQ_LETTERS нужно будет передать функции itertools.product(). Вместе с ней мы передадим также вероятную длину ключа в качестве второго аргумента, чтобы длина создаваемых кортежей была равна длине потенциального ключа.

Например, если мы хотим испытать только первые четыре наиболее вероятные буквы каждого подключа (их количество определяется константой NUM_MOST_FREQ_LETTERS), в то время как сам ключ, вероятнее

всего, состоит из пяти букв, то первым значением, созданным функцией `itertools.product()`, будет кортеж `(0, 0, 0, 0, 0)`. Следующими значениями будут кортежи `(0, 0, 0, 0, 1)`, затем `(0, 0, 0, 0, 2)` и т.д. до тех пор, пока не будет создан кортеж `(3, 3, 3, 3, 3)`. Каждое из пяти целочисленных значений кортежа представляет собой индекс элемента в списке `allFreqScores`.

Доступ к подключам, хранящимся в списке `allFreqScores`

Значением переменной `allFreqScores` является список наиболее вероятных букв каждого подключа вместе с их оценками частотного соответствия. Чтобы увидеть, как это выглядит, создайте гипотетический список `allFreqScores` в IDLE. Например, в случае шестибуквенного ключа, для которого мы нашли четыре наиболее вероятные буквы каждого из подключей, список может иметь следующий вид.

```
>>> allFreqScores = [[('A', 9), ('E', 5), ('O', 4), ('P', 4)], [('S', 10),
('D', 4), ('G', 4), ('H', 4)], [('I', 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), ('I', 5), ('K', 5), ('Z', 4)]]
```

Список кажется слишком сложным, но мы можем добраться до конкретных значений с помощью индексов. Если обратиться к списку `allFreqScores` по индексу элемента, то мы получим список кортежей возможных букв для этого подключа и их оценки частотного соответствия. Например, элемент `allFreqScores[0]` содержит список кортежей для первого подключа вместе с оценками для каждого потенциального подключа, элемент `allFreqScores[1]` — аналогичные данные для второго подключа и т.д.

```
>>> allFreqScores[0]
[('A', 9), ('E', 5), ('O', 4), ('P', 4)]
>>> allFreqScores[1]
[('S', 10), ('D', 4), ('G', 4), ('H', 4)]
```

Можно также получить доступ к отдельным кортежам возможных букв для каждого подключа, добавив дополнительный индекс. Например, обратившись к элементу `allFreqScores[1][0]`, мы получим кортеж, включающий наиболее вероятную букву второго подключа и ее оценку частотного соответствия. Обратившись к элементу `allFreqScores[1][1]`, получим кортеж, включающий вторую наиболее вероятную букву того же подключа, и т.д.

```
>>> allFreqScores[1][0]
('S', 10)
>>> allFreqScores[1][1]
('D', 4)
```

Поскольку эти значения являются кортежами, для получения самой буквы без оценки ее частотного соответствия потребуется обратиться к первому элементу кортежа. Таком образом, для доступа к наиболее вероятной букве второго подключа мы должны использовать элемент `allFreqScores[1][0][0]`, для доступа ко второй наиболее вероятной букве этого же подключа — элемент `allFreqScores[1][1][0]` и т.д.

```
>>> allFreqScores[1][0][0]
'S'
>>> allFreqScores[1][1][0]
'D'
```

Разобравшись с доступом к потенциальным подключам, содержащимся в списке `allFreqScores`, мы должны скомбинировать их для нахождения потенциальных ключей.

Создание комбинаций подключей с помощью функции `itertools.product()`

Каждый из кортежей, созданных с помощью функции `itertools.product()`, представляет один ключ, причем позиция в кортеже соответствует первому индексу, по которому мы получаем доступ к списку `allFreqScores`, а целые числа в кортеже представляют второй индекс в списке `allFreqScores`.

Поскольку ранее мы установили для константы `NUM_MOST_FREQ_LETTERS` значение 4, вызов `itertools.product(range(NUM_MOST_FREQ_LETTERS), repeat=mostLikelyKeyLength)` в строке 188 означает, что для каждого значения переменной `indexes` в цикле `for` мы получаем кортеж целых чисел (от 0 до 3), представляющих четыре наиболее вероятные буквы для каждого подключа.

```
188.     for indexes in itertools.product(range(NUM_MOST_FREQ_LETTERS),
           repeat=mostLikelyKeyLength):
189.         # Создаем возможный ключ из букв в списке allFreqScores
190.         possibleKey = ''
191.         for i in range(mostLikelyKeyLength):
192.             possibleKey += allFreqScores[i][indexes[i]][0]
```

Мы конструируем полные ключи Виженера, используя переменную `indexes`, которая на каждой итерации получает значение в виде одного кортежа, созданного с помощью функции `itertools.product()`. Ключ инициализируется пустой строкой в строке 190, а во вложенном цикле `for` (строки 191 и 192) программа проходит по целым числам в диапазоне от `0` до значения `mostLikelyKeyLength` (но не включая его) для каждого кортежа, формируя ключ.

Поскольку значение переменной `i` изменяется на каждой итерации цикла, значение `indexes[i]` становится индексом кортежа, который мы выбираем в элементе `allFreqScores[i]`. Именно поэтому, обращаясь к элементу `allFreqScores[i][indexes[i]]`, мы получаем нужный кортеж. Далее необходимо обратиться к его элементу с индексом `0`, чтобы получить букву подключа.

Если для константы `SILENT_MODE` установлено значение `False`, то в строке 195 выводится сформированный ключ.

```
194.         if not SILENT_MODE:
195.             print('Attempting with key: %s' % (possibleKey))
```

Теперь, когда в нашем распоряжении имеется полный ключ Виженера, мы можем дешифровать шифротекст (строки 197–208) и проверить, получен ли в результате осмысленный текст на английском языке. В случае положительного исхода такой проверки программа выводит дешифрованный текст на экран, чтобы дать пользователю возможность принять окончательное решение.

Вывод дешифрованного текста с сохранением корректного регистра букв

Поскольку переменная `decryptedText` содержит буквы в верхнем регистре, мы создаем новую строку путем добавления в список `origCase` букв из строки `decryptedText`, предварительно преобразуя их в нужный регистр (строки 201–207).

```
197.         decryptedText = vigenereCipher.decryptMessage(possibleKey,
               ciphertextUp)
198.
199.         if detectEnglish.isEnglish(decryptedText):
200.             # Задаем исходный регистр букв во взломанном шифротексте
201.             origCase = []
202.             for i in range(len(ciphertext)):
203.                 if ciphertext[i].isupper():
204.                     origCase.append(decryptedText[i].upper())
```

```
205.                 else:
206.                     origCase.append(decryptedText[i].lower())
207.             decryptedText = ''.join(origCase)
```

Цикл `for`, начинающийся в строке 202, проходит по индексам строки `ciphertext`, которая, в отличие от строки `ciphertextUp`, сохраняет исходную версию регистра букв шифротекста. Если `ciphertext[i]` — буква в верхнем регистре, то буква `decryptedText[i]` преобразуется в верхний регистр и добавляется в список `origCase`. В противном случае буква `decryptedText[i]` добавляется в список `origCase` без изменений. По окончании цикла элементы списка `origCase` объединяются (строка 207) в одну строку `decryptedText`.

В следующем фрагменте результат дешифрования отображается на экране, чтобы пользователь мог проверить, действительно ли найден истинный ключ.

```
210.             print('Possible encryption hack with key %s:' %
                       (possibleKey))
211.             print(decryptedText[:200])  # выводим первые 200 символов
212.             print()
213.             print('Enter D if done, anything else to continue
                       hacking:')
214.             response = input('> ')
215.
216.             if response.strip().upper().startswith('D'):
217.                 return decryptedText
```

Дешифрованный текст в корректном регистре выводится на экран, чтобы пользователь мог подтвердить, что является ли он осмысленным текстом на английском языке. Если пользователь вводит букву `'D'`, то функция возвращает строку `decryptedText`.

В противном случае, если ни один из дешифрованных вариантов не был принят пользователем, попытка взлома считается неудачной, и функция возвращает значение `None`:

```
220.     return None
```

Получение взломанного сообщения

Все созданные нами функции будут вызываться в функции `hackVigenere()`, которая получает строку шифротекста в качестве аргумента и возвращает взломанное сообщение (если взлом оказался успешным) или значение `None` (если попытка взлома оказалась неудачной). Вначале

мы определяем возможные варианты длины ключа с помощью функции `kasiskiExamination()`.

```
223. def hackVigenere(ciphertext):
224.     # Прежде всего, необходимо применить метод Касиски
225.     # для выяснения возможной длины ключа
226.     allLikelyKeyLengths = kasiskiExamination(ciphertext)
```

Список возможных вариантов выводится на экран, если для константы `SILENT_MODE` было установлено значение `False`.

```
227.     if not SILENT_MODE:
228.         keyLengthStr = ''
229.         for keyLength in allLikelyKeyLengths:
230.             keyLengthStr += '%s ' % (keyLength)
231.         print('Kasiski examination results say the most likely
                  key lengths are: ' + keyLengthStr + '\n')
```

Далее необходимо найти наиболее вероятные буквы подключей для каждой длины ключа. Это делается с помощью еще одного цикла, в котором функция пытается взломать шифр, используя каждую найденную длину ключа.

Выход из цикла, если найден потенциальный ключ

Мы хотим, чтобы цикл тестирования длины ключа продолжался до тех пор, пока не будет найдена корректная длина ключа. Если функция `attemptHackWithKeyLength()` сумела подобрать ключ, то цикл принудительно завершается с помощью инструкции `break`.

По аналогии с тем, как инструкция `continue` применяется в теле цикла для возврата в его начало, инструкция `break` предназначена для немедленного выхода из цикла. Встретив данную инструкцию, программа немедленно переходит к выполнению строки, стоящей сразу за телом цикла. Мы прервем цикл, когда программа найдет корректный ключ, подтвержденный пользователем.

```
232.     hackedMessage = None
233.     for keyLength in allLikelyKeyLengths:
234.         if not SILENT_MODE:
235.             print('Attempting hack with key length %s
                      (%s possible keys) ...' % (keyLength,
                      NUM_MOST_FREQ_LETTERS ** keyLength))
236.         hackedMessage = attemptHackWithKeyLength(ciphertext, keyLength)
237.         if hackedMessage != None:
238.             break
```

В строке 236 для каждой возможной длины ключа вызывается функция `attemptHackWithKeyLength()`. Если она не вернула значение `None`, то взлом считается успешным, и программа выходит из цикла в строке 238.

Тестирование всех остальных вариантов длины ключа методом грубой силы

Если попытки взлома с использованием всех возможных вариантов длины ключа, возвращаемых функцией `kasiskiExamination()`, оказались неудачными, то в строке 242 переменная `hackedMessage` окажется равна `None`. В этом случае проверяются все остальные варианты длины ключа вплоть до `MAX_KEY_LENGTH`. Другими словами, если с помощью метода Касиски не удалось вычислить корректную длину ключа, то мы выполняем полный перебор вариантов в цикле `for` (строка 245).

```
242.     if hackedMessage == None:
243.         if not SILENT_MODE:
244.             print('Unable to hack message with likely key length(s).
                       Brute-forcing key length...')
245.         for keyLength in range(1, MAX_KEY_LENGTH + 1):
246.             # Не перепроверять длину ключа, уже опробованную
                   методом Касиски
247.             if keyLength not in allLikelyKeyLengths:
248.                 if not SILENT_MODE:
249.                     print('Attempting hack with key length %s (%s
                           possible keys)...' % (keyLength,
                           NUM_MOST_FREQ_LETTERS ** keyLength))
250.                 hackedMessage = attemptHackWithKeyLength(ciphertext,
                       keyLength)
251.                 if hackedMessage != None:
252.                     break
```

В строке 245 начинается цикл `for`, в котором для каждого значения переменной `keyLength` (имеющей значения от `1` до `MAX_KEY_LENGTH`) вызывается функция `attemptHackWithKeyLength()`, при условии, что это значение не содержится в списке `allLikelyKeyLengths`. Причина заключается в том, что значения длины ключа, содержащиеся в переменной `allLikelyKeyLengths`, ранее уже были проверены (строки 233–238).

Наконец, значение `hackedMessage` возвращается в строке 253:

```
253.     return hackedMessage
```

Вызов функции main()

Функция main() будет вызвана в том случае, если программа была запущена непосредственно, а не импортируется в виде модуля другой программой.

```
256. # Если файл vigenereHacker.py выполняется как программа
257. # (а не импортируется как модуль), вызвать функцию main():
258. if __name__ == '__main__':
259.     main()
```

На этом рассмотрение программы взлома шифра Виженера можно считать завершенным. Будет ли взлом успешным, зависит от характеристик шифротекста. Чем ближе частотность букв исходного текста к частотности букв типичных английских текстов и чем длиннее исходный текст, тем выше вероятность успеха.

Изменение значений констант, используемых в программе

Если программа взлома не справилась с задачей, то можно попытаться изменить ряд настроек. На работу программы оказывают влияние три константы, которые задаются в строках 8–10.

```
 8. MAX_KEY_LENGTH = 16         # ограничение длины проверяемых ключей
 9. NUM_MOST_FREQ_LETTERS = 4   # ограничение количества букв на подключ
10. SILENT_MODE = False         # True - отключение вывода
```

Если длина ключа шифра Виженера превышает значение константы MAX_KEY_LENGTH (строка 8), то программе взлома заведомо не удастся определить корректный ключ. В случае неудачи попробуйте увеличить это значение и повторно выполнить программу.

Попытки взлома с использованием коротких некорректных ключей занимают мало времени. Но если для константы MAX_KEY_LENGTH установлено слишком большое значение и функция kasiskiExamination() ошибочно решит проверять ключи чрезмерно большой длины, то она может выполняться часами или даже месяцами, пытаясь взломать шифротекст с помощью ключей ошибочной длины.

Чтобы этого избежать, константа NUM_MOST_FREQ_LETTERS (строка 9) ограничивает возможное количество букв, проверяемых для каждого подключа. С увеличением данного значения программа взлома будет тестировать намного большее количество ключей. Это может потребо-

ваться, если оценки, полученные с помощью функции `freqAnalysis.englishFreqMatchScore()`, оказались неточными, но учитывайте, что работа программы существенно замедлится. А установка значения `26` для константы `NUM_MOST_FREQ_LETTERS` полностью исключит сокращение числа возможных букв для каждого подключа!

Уменьшение значений констант `MAX_KEY_LENGTH` и `NUM_MOST_FREQ_LETTERS` ускоряет работу программы, но снижает вероятность успешного взлома шифра, тогда как их увеличение замедляет выполнение, но повышает вероятность успеха.

Наконец, для увеличения быстродействия программы можно установить для константы `SILENT_MODE` значение `True`, чтобы программа не тратила понапрасну время на вывод информации на экран. Бывает так, что компьютер сам по себе работает очень быстро, но отображение сообщений на экране проходит относительно медленно. Недостатком отказа от вывода информации на экран является то, что вам ничего не будет известно о ходе процесса, и вы узнаете о результате только после того, как программа полностью завершит свою работу.

Резюме

Взлом шифра Виженера требует соблюдения определенной последовательности действий. Кроме того, многие компоненты программы взлома могут не сработать. Например, ключ шифра Виженера, использованный для шифрования текста, может иметь длину, превышающую значение `MAX_KEY_LENGTH`, или же функция, выдающая оценку частотного соответствия, может получить неточные результаты, поскольку частотность букв в открытом тексте отличается от типичной. Кроме того, открытый текст может содержать слишком много слов, которые отсутствуют в файле словаря, и функция `isEnglish()` не распознает его как английский текст.

Проверяя различные причины, по которым программа взлома не сработала, вы сможете вносить в код соответствующие изменения для обработки подобных случаев. Тем не менее представленный в книге вариант программы взлома отлично справляется с задачей снижения количества возможных ключей, исчисляемых миллиардами или триллионами, всего лишь до нескольких тысяч.

Вместе с тем существует одна методика, которая делает взлом шифра Виженера математически невозможным, каким бы мощным ни был компьютер и какой бы изощренной ни была программа взлома. Об этой методике, получившей название *одноразовый шифроблокнот*, речь пойдет в главе 21.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Что такое перебор по словарю?
2. Какую информацию о шифротексте предоставляет метод Касиски?
3. Какие два изменения происходят при преобразовании списка в множество с помощью функции `set()`?
4. Переменная `spam` представляет собой список `['cat', 'dog', 'mouse', 'dog']`, содержащий четыре элемента. Сколько элементов будет содержать список, возвращаемый функцией `list(set(spam))`?
5. Что выведет следующий код?

```
print('Hello', end='')
print('World')
```

21

ОДНОРАЗОВЫЙ ШИФРОБЛОКНОТ

"Я тысячу раз прокручивал это в голове, – произносит Уотерхауз, – и вижу единственное объяснение: они переводят свои сообщения в большие двоичные числа и комбинируют их с другими большими двоичными числами – скорее всего, одноразовыми шифроблокнотами. Тогда ты ничего не добьешься, – говорит Алан. – Одноразовый шифроблокнот взломать нельзя".

Нил Стивенсон, "Криптономикон"

В этой главе вы узнаете о шифре, который невозможно взломать, каким бы мощным ни был компьютер, сколько бы времени вы ни тратили на взлом и каким бы гениальным хакером вы ни были. Речь идет о так называемом *одноразовом шифроблокноте* (one-time pad cipher). К счастью, нам не придется писать для него новую программу! Программа для работы с шифром Виженера, которую мы создали в главе 18, позволяет реализовать этот шифр без внесения в нее каких-либо изменений. Однако шифрование с помощью одноразового блокнота настолько неудобно для регулярного применения, что к нему в основном прибегают лишь с целью кодирования сообщений особой важности.

В этой главе...

- Не поддающийся взлому одноразовый шифроблокнот
- Двухразовый шифроблокнот как эквивалент шифра Виженера

Не поддающийся взлому одноразовый шифроблокнот

Одноразовый шифроблокнот — это шифр Виженера, который приобретает абсолютную криптографическую стойкость, если ключ удовлетворяет следующим критериям:

1) длина ключа совпадает с длиной открытого сообщения;
2) символы ключа выбираются абсолютно случайным образом;
3) ключ используется всего один раз и больше не применяется ни к каким другим сообщениям.

Придерживаясь этих трех правил, можно сделать зашифрованное сообщение неуязвимым для любых видов криптоанализа. Такой ключ невозможно взломать, даже располагая неограниченными вычислительными ресурсами.

Свое название шифр получил благодаря тому, что обычно его ключи записывали в блокноте. После использования ключа верхний лист блокнота отрывали, чтобы перейти к следующему ключу. Как правило, в блокноте сразу записывали большое количество ключей, помечая ключи конкретными датами, а сам блокнот передавали из рук в руки. Например, если зашифрованное сообщение было получено 31 октября, то следовало пролистать блокнот и найти ключ, соответствующий этой дате.

Выравнивание длины ключа по размеру сообщения

Чтобы понять, почему шифротекст, зашифрованный с помощью одноразового шифроблокнота, невозможно взломать, порассуждаем над тем, что делает обычный шифр Виженера уязвимым для атак. Вспомните, что в основе нашей программы взлома шифра Виженера лежит частотный анализ. Но если размер ключа равен размеру сообщения, то подключ каждой буквы открытого текста является уникальным в том смысле, что каждая такая буква может быть зашифрована в любую букву шифротекста с равной вероятностью.

Например, чтобы зашифровать сообщение "IF YOU WANT TO SURVIVE OUT HERE, YOU'VE GOT TO KNOW WHERE YOUR TOWEL IS", мы удаляем из него пробелы и знаки препинания, получая в результате сообщение длиной 55 букв. Для применения одноразового шифроблокнота необходимо получить ключ, длина которого тоже составляет 55 букв. Выбрав в качестве примера ключ KCQYZHEPXAUTIQEKXEJMORETZHZTRWWQDYLBTTVEJMEDBSANYBPXQIK, мы получим шифротекст "SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC" (рис. 21.1).

Открытый текст	IFYOUWANTTOSURVIVEOUTHEREYOUVEGOTTOKNOWWHEREYOURTOWELIS
Ключ	KCQYZHEPXAUTIQEKXEJMORETZHZTRWWQDYLBTTVEJMEDBSANYBPXQIK
Шифротекст	SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

Рис. 21.1. *Шифрование сообщения с помощью одноразового шифроблокнота*

А теперь представьте, что криптоаналитик завладел шифротекстом ("SHOMTDEC..."). Как взломать такой шифр? Полный перебор ключей методом грубой силы не сработает, так как это чересчур много даже для компьютера. Количество ключей будет равно 26 в степени, равной числу букв в сообщении. Если длина сообщения — 55 букв, как в нашем примере, то общее число возможных ключей составит 26^{55}, или 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.

Но, даже располагая компьютером, достаточно мощным для того, чтобы испытать все ключи, криптоаналитик все равно не сможет взломать одноразовый шифроблокнот, *поскольку для любого шифротекста все возможные варианты открытых текстов одинаково вероятны.*

Шифротекст "SHOMTDEC..." может оказаться результатом шифрования совершенно иного сообщения с тем же количеством букв, например текста "THE MYTH OF OSIRIS WAS OF IMPORTANCE IN ANCIENTEGYPTIAN RELIGION", зашифрованного с помощью ключа ZAKAVKXOLFQDLZHWSQJBZMTWMMNAKWURWEXDCUYWKSGORGHNNEDVTCP (рис. 21.2).

Открытый текст	THEMYTHOFOSIRISWASOFIMPORTANCEINANCIENTEGYPTIANRELIGION
Ключ	ZAKAVKXOLFQDLZHWSQJBZMTWMMNAKWURWEXDCUYWKSGORGHNNEDVTCP
Шифротекст	SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

Рис. 21.2. *Шифрование другого сообщения с использованием другого ключа, приводящее к точно такому же шифротексту*

Причина, по которой мы вообще способны взломать какой-либо шифр, заключается в том, что обычно существует лишь один ключ, применение которого для дешифрования сообщения позволяет получить осмысленный текст на английском языке. Но, как только что было показано на конкретном примере, *один и тот же* шифротекст может быть получен для двух совершенно *разных* сообщений. В случае одноразового шифроблокнота у криптоаналитика нет возможности определить, какое из них является истинным. По сути, *любое* сообщение на английском языке, насчитывающее ровно 55 букв, может с *равной вероятностью* оказаться истинным. Одного лишь факта получения осмысленного текста на английском языке недостаточно для определения исходного ключа шифрования.

Поскольку для получения шифротекста с равной вероятностью мог быть использован любой текст на английском языке, сообщение, зашифрованное с помощью одноразового шифроблокнота, не поддается взлому.

Создание истинно случайного ключа

Как обсуждалось в главе 9, встроенный модуль `random` не позволяет получать истинно случайные числа. Применяемый алгоритм генерирует так называемые *псевдослучайные* числа, которые лишь кажутся случайными, чего вполне достаточно в большинстве задач. Однако в случае одноразового шифроблокнота ключ должен быть истинно случайным числом, в противном случае шифр теряет свою абсолютную криптографическую стойкость.

В Python версии 3.6 и выше имеется модуль `secrets`, который в качестве источника истинно случайных чисел использует операционную систему (чаще всего таким источником служат случайные события, например промежуток времени между последовательными нажатиями клавиш). Функция `secrets.randbelow()` возвращает истинно случайное число в диапазоне от 0 до значения, определяемого аргументом (не включая его самого).

```
>>> import secrets
>>> secrets.randbelow(10)
2
>>> secrets.randbelow(10)
0
>>> secrets.randbelow(10)
6
```

Функции модуля `secrets` работают медленнее по сравнению с функциями модуля `random`, поэтому последние предпочтительнее в ситуациях, ког-

да истинная случайность не нужна. Можно также воспользоваться функцией `secrets.choice()`, которая возвращает случайно выбранный элемент из переданной ей строки или списка.

```
>>> import secrets
>>> secrets.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
'R'
>>> secrets.choice(['cat', 'dog', 'mouse'])
'dog'
```

Для создания истинно случайного одноразового шифроблокнота размером 55 символов можно использовать следующий код.

```
>>> import secrets
>>> otp = ''
>>> for i in range(55):
...         otp += secrets.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
...
>>> otp
'MVOVAAYDPELIRNRUZNNQHDNSOUWWNWPJUPIUAIMKFKNHQANIIYCHHDC'
```

Кроме того, существует еще одно обстоятельство, о котором никогда нельзя забывать при работе с одноразовым шифроблокнотом. Рассмотрим, почему следует избегать повторного использования одноразовых шифроблокнотов.

Избегайте двухразовых шифроблокнотов

Под *двухразовым шифроблокнотом* понимают использование одного и того же ключа одноразового шифроблокнота для шифрования двух разных сообщений. Это создает критическую уязвимость.

Как уже говорилось ранее, тот факт, что результат расшифровки сообщения, зашифрованного с помощью одноразового шифроблокнота, представляет собой осмысленный текст на английском языке, еще не означает, что найден правильный ключ. Но если вы используете один и тот же ключ для шифрования двух разных сообщений, то тем самым предоставите взломщику ценнейшую информацию. В этом случае взломщик, получив осмысленный текст в результате дешифрования первого шифротекста с помощью найденного ключа и увидев, что применение этого же ключа ко второму шифротексту дает случайный набор букв, сразу же поймет, что ключ не подходит. Как будет показано в следующем разделе, существует очень высокая вероятность того, что лишь один ключ позволяет получить осмысленные тексты при дешифровании обоих сообщений.

Если в распоряжении взломщика имеется только одно из двух сообщений, то расшифровать его по-прежнему невозможно. Но мы всегда должны исходить из того, что *все* наши зашифрованные сообщения могут перехватываться хакерами и шпионами. В противном случае незачем переживать о шифровании сообщений. Никогда не забывайте о максиме Шеннона: враг знает систему! Это относится ко всему, что связано с шифротекстами.

Почему двухразовый шифроблокнот эквивалентен шифру Виженера

Вы уже знаете, как взламывать шифр Виженера. Если мы сможем показать, что двухразовый шифроблокнот аналогичен шифру Виженера, то сможем и взломать его теми же способами.

Чтобы понять, почему двухразовый шифроблокнот уязвим подобно шифру Виженера, рассмотрим работу шифра Виженера в том случае, когда размер сообщения превышает длину ключа. Исчерпав буквы ключа, мы возвращаемся к его первой букве и продолжаем процесс шифрования. Например, чтобы зашифровать сообщение "BLUE IODINE INBOUND CAT" длиной 20 букв с помощью ключа YZNMPZXYXY длиной 10 букв, мы шифруем первые 10 букв ("BLUE IODINE") ключом YZNMPZXYXY, после чего тем же ключом шифруем следующие 10 букв ("INBOUND CAT"). Эффект завертывания проиллюстрирован на рис. 21.3.

Открытый текст	BLUEIODINEINBOUNDCAT
Ключ Виженера	YZNMPZXYXYYZNMPZXYXY
Шифротекст Виженера	ZKHQXNAGKCGMOAJMAAXR

Рис. 21.3. *Эффект завертывания в шифре Виженера*

Предположим, что шифрование 10-буквенного сообщения "BLUE IODINE" осуществляется с использованием одноразового шифроблокнота и ключа YZNMPZXYXY. Тогда криптограф, шифрующий второе 10-буквенное сообщение "INBOUND CAT" с помощью того же ключа YZNMPZXYXY, совершает серьезную ошибку (рис. 21.4).

	Сообщение 1	**Сообщение 2**
Открытый текст	BLUEIODINE	INBOUNDCAT
Ключ одноразового шифроблокнота	YZNMPZXYXY	YZNMPZXYXY
Шифротекст одноразового шифроблокнота	ZKHQXNAGKC	GMOAJMAAXR

Рис. 21.4. *Шифрование открытого текста с помощью одноразового шифроблокнота дает тот же шифротекст, что и в случае шифра Виженера*

Если мы сравним шифротекст, полученный с помощью шифра Виженера ("ZKHQXNAGKCGMOAJMAAXR" на рис. 21.3), с шифротекстами, полученными с помощью одноразового шифроблокнота ("ZKHQXNAGKC" и "GMOAJMAAXR" на рис. 21.4), то увидим, что они в точности совпадают. Таким образом, двухразовый шифроблокнот обладает теми же свойствами, что и шифр Виженера, а значит, его можно взломать теми же самыми способами!

Резюме

Одноразовый шифроблокнот — это методика, позволяющая сделать шифр Виженера неуязвимым для взлома. Для этого необходимо, чтобы длина ключа совпадала с размером сообщения, а сам ключ был истинно случайным и использовался строго один раз. Соблюдение всех трех условий полностью исключает возможность взлома сообщения, зашифрованного с помощью одноразового шифроблокнота. Однако применять такую методику для регулярного шифрования сообщений не очень удобно. Обычно одноразовые шифроблокноты со списком ключей передаются из рук в руки. При этом вы должны быть уверены в том, что список не попадет в чужие руки!

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Почему в этой главе мы не создавали программу для одноразового шифроблокнота?
2. Эквивалентом какого шифра является двухразовый шифроблокнот?
3. Если использовать ключ, длина которого в два раза превышает размер исходного сообщения, то станет ли шифрование с помощью одноразового шифроблокнота в два раза более безопасным?

22

НАХОЖДЕНИЕ И ГЕНЕРИРОВАНИЕ ПРОСТЫХ ЧИСЕЛ

"До сегодняшнего дня математики безуспешно пытаются обнаружить некий порядок в последовательности простых чисел, и у нас есть все основания полагать, что это – тайна, в которую человеческому уму никогда не удастся проникнуть".

Леонард Эйлер, математик

Все рассмотренные нами шифры существуют уже на протяжении сотен лет. Они были хороши до тех пор, пока те, кто пытались их взломать, могли полагаться лишь на карандаш и бумагу, но с появлением компьютеров, способных манипулировать данными в триллионы раз быстрее человека, они стали уязвимыми. Другим недостатком классических шифров является то, что для кодирования и декодирования сообщений в них применяется один и тот же секретный ключ. Использование секретного ключа порождает проблемы при пересылке зашифрованных сообщений. Как организовать безопасную передачу секретного ключа, чтобы адресат смог расшифровать сообщение?

В главе 23 вы узнаете о том, как улучшить старые шифры, используя очень большие простые числа для создания двух ключей: открытого – для шифрования сообщений и закрытого – для их дешифрования. Чтобы научиться применять криптосистемы с открытым ключом, следует узнать о

некоторых важных свойствах простых чисел (и трудностях разложения больших чисел на множители), делающих возможным данный тип шифрования. В этой главе мы напишем модуль *primeNum.py*, позволяющий генерировать ключи на основе алгоритма быстрого определения простых чисел.

В этой главе...

- Простые и составные числа
- Проверка чисел на простоту методом перебора делителей
- Решето Эратосфена
- Тест Миллера — Рабина

Что такое простое число

Простым называют целое число, которое больше 1 и при этом делится без остатка только на 1 и на само себя, т.е. имеет ровно два *множителя*. Вспомните, что множителями называют числа, произведение которых равно исходному числу. Например, множителями числа 21 являются числа 3 и 7. У числа 12 имеются множители 2 и 6, а также 3 и 4.

Каждое число *x* имеет множители 1 и *x*, поскольку результатом умножения любого числа на 1 всегда будет само число. Например, 1 и 21 — множители числа 21, а 1 и 12 — множители числа 12. Если никаких других множителей числа не существует, то такое число является простым. Например, 2 — простое число, поскольку у него нет никаких других множителей, кроме 1 и 2.

Вот начало бесконечной последовательности простых чисел (обратите внимание на то, что число 1 не считается простым): 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 и т.д.

Простых чисел бесконечно много, а это означает, что такого понятия, как *наибольшее простое число*, не существует. Как и в случае любых натуральных чисел, для каждого простого числа всегда найдется простое число, которое больше его. В криптосистемах с открытым ключом большие простые числа применяются с целью получения ключа, слишком длинного для того, чтобы его можно было взломать методом грубой силы.

Нахождение простых чисел сопряжено с определенными вычислительными трудностями, а находить большие простые числа для криптосистем

с открытым ключом еще труднее. Чтобы получить открытый ключ, мы будем генерировать случайное большое число и проверять, является ли оно простым, с помощью *теста простоты*. Если тест подтверждает простоту числа, то мы используем его, в противном случае процесс повторяется.

Чтобы проиллюстрировать, насколько большими могут быть простые числа, рассмотрим ряд примеров.

Гугол — это число 10, возведенное в степень 100. Оно записывается как единица с последующими 100 нулями:

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

В миллиарде квинтиллионов гуголов содержится на 27 нулей больше:

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
000 000 000 000 000 000 000 000 000 000 000

Но даже эти числа являются ничтожно малыми по сравнению с теми простыми числами, которые применяются в криптосистемах с открытым ключом. Такие простые числа содержат сотни цифр, как в следующем примере:

112 829 754 900 439 506 175 719 191 782 841 802 172 556 768 253
593 054 977 186 235 584 979 780 304 652 423 405 148 425 447 063
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

Это настолько большое число, что даже страшно представить!

У простых чисел есть ряд интересных особенностей, о которых следует знать. Поскольку все четные числа кратны двум, 2 — единственное четное простое число. Кроме того, результатом умножения двух простых чисел будет число, единственными множителями которого являются единица, само это число и два исходных простых числа. (Например, при умножении простых чисел 3 и 7 получаем число 21, единственные множители которого — 1, 21, 3 и 7.)

Целые числа, не являющиеся простыми, называются *составными*, поскольку их можно разложить как минимум на два множителя, не равных единице и самому числу. Любое составное число можно разложить на произведение простых чисел (эту операцию называют *факторизацией*). Напри-

мер, составное число 1386 факторизуется на простые множители 2, 3, 7 и 11, поскольку $2 \cdot 3 \cdot 3 \cdot 7 \cdot 11 = 1386$. Любое составное число имеет единственное разложение на простые множители (данное утверждение называют *основной теоремой арифметики*).

Рассмотренные особенности простых чисел будут использованы для написания модуля *primeNum.py*, позволяющего определять, являются ли заданные числа простыми, а также генерировать простые числа. Модуль будет содержать следующие функции.

- Функция **`isPrimeTrialDiv()`** использует алгоритм перебора делителей, возвращая `True`, если аргумент — простое число, и `False` в противном случае.
- Функция **`primeSieve()`** использует алгоритм решета Эратосфена для генерирования простых чисел.
- Функция **`rabinMiller()`** использует тест Миллера — Рабина для проверки простоты числа. В отличие от алгоритма перебора делителей данный тест работает значительно быстрее в случае больших чисел. Эта функция вызывается не напрямую, а из функции `isPrime()`.
- Функция **`isPrime()`** позволяет проверить, является ли переданное ей большое целое число простым.
- Функция **`generateLargePrime()`** возвращает большое простое число, содержащее сотни цифр. Эта функция будет использоваться в программе *makePublicPrivateKeys.py* в главе 23.

Исходный код модуля `primeNum`

Как и в случае модуля *cryptomath.py*, рассмотренного в главе 13, предполагается, что программа *primeNum.py* будет импортироваться в виде модуля другими программами, а не выполняться самостоятельно. Программа импортирует модули `math` и `random`, задействуя их функции для генерирования простых чисел.

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *primeNum.py*.

primeNum.py

```
1. # Решето простых чисел
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import math, random
5.
6.
```

```
 7. def isPrimeTrialDiv(num):
 8.     # Возвращает True, если num - простое число, иначе - False
 9.
10.     # Использует алгоритм перебора делителей для проверки простоты числа
11.
12.     # Числа, меньшие 2, не являются простыми
13.     if num < 2:
14.         return False
15.
16.     # Проверяем делимость на целые числа вплоть до
          квадратного корня из num
17.     for i in range(2, int(math.sqrt(num)) + 1):
18.         if num % i == 0:
19.             return False
20.     return True
21.
22.
23. def primeSieve(sieveSize):
24.     # Возвращает список простых чисел, вычисленных
25.     # с помощью решета Эратосфена
26.
27.     sieve = [True] * sieveSize
28.     sieve[0] = False  # 0 и 1 не являются простыми числами
29.     sieve[1] = False
30.
31.     # Создаем решето
32.     for i in range(2, int(math.sqrt(sieveSize)) + 1):
33.         pointer = i * 2
34.         while pointer < sieveSize:
35.             sieve[pointer] = False
36.             pointer += i
37.
38.     # Формируем список простых чисел
39.     primes = []
40.     for i in range(sieveSize):
41.         if sieve[i] == True:
42.             primes.append(i)
43.
44.     return primes
45.
46. def rabinMiller(num):
47.     # Возвращает True, если num - простое число
48.     if num % 2 == 0 or num < 2:
49.         return False  # тест не работает для четных чисел
50.     if num == 3:
51.         return True
52.     s = num - 1
53.     t = 0
54.     while s % 2 == 0:
```

```
        # Делим s на 2 до тех пор, пока не получим нечетное
        # число (используя t для подсчета таких делений)
        s = s // 2
        t += 1
    for trials in range(5):  # пытаемся фальсифицировать проверку
                               простоты числа num 5 раз
        a = random.randrange(2, num - 1)
        v = pow(a, s, num)
        if v != 1:  # тест неприменим, если v равно 1
            i = 0
            while v != (num - 1):
                if i == t - 1:
                    return False
                else:
                    i = i + 1
                    v = (v ** 2) % num
    return True

# Обычно можно быстро определить, что num не является простым числом,
# разделив его на первые несколько десятков простых чисел. Это быстрее,
# чем вызывать функцию rabinMiller(), но годится не всегда.
LOW_PRIMES = primeSieve(100)


def isPrime(num):
    # Возвращает True, если num - простое число. Сначала выполняются
    # быстрые проверки перед вызовом функции rabinMiller().
    if (num < 2):
        return False  # 0, 1 и отрицательные числа не являются простыми
    # Проверяем, делится ли параметр num на простые числа из списка
    for prime in LOW_PRIMES:
        if (num == prime):
            return True
        if (num % prime == 0):
            return False
    # В остальных случаях вызываем функцию rabinMiller()
    return rabinMiller(num)


def generateLargePrime(keysize=1024):
    # Возвращает случайное простое число размером keysize бит
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num
```

Пример работы модуля primeNum

Чтобы увидеть, как работает модуль *primeNum.py*, введите в интерактивной оболочке следующие инструкции.

```
>>> import primeNum
>>> primeNum.generateLargePrime()
122881168342211041030523683515443239007484290600701555369488271748378054744009463751312511471291011945732413378446666809140502037003673211052153493607681619990563076859566835016382556518967124921538212397036345815983641146000671635019637218348455544435908428400192565849620509600312468757953899553441648428119
>>> primeNum.isPrime(45943208739848451)
False
>>> primeNum.isPrime(13)
True
```

Благодаря модулю *primeNum.py* мы получаем возможность генерировать очень большие простые числа с помощью функции `generateLargePrime()`. Кроме того, можно передать любое число функции `isPrime()`, чтобы определить, является ли оно простым.

Как работает алгоритм перебора делителей

Чтобы узнать, является ли заданное число простым, мы используем *алгоритм перебора делителей* (trial division algorithm), в соответствии с которым делим число на целые числа (2, 3 и т.д.), проверяя, делится ли оно на них без остатка. Например, чтобы выяснить, является ли число 49 простым, мы проверим его делимость на целые числа, начиная с 2:

49 ÷ 2 = 24, остаток 1,

49 ÷ 3 = 16, остаток 1,

49 ÷ 4 = 12, остаток 1,

49 ÷ 5 = 9, остаток 4,

49 ÷ 6 = 8, остаток 1,

49 ÷ 7 = 7, остаток 0.

Поскольку 49 делится на 7 без остатка, мы определяем 7 как множитель 49. Это означает, что число 49 не может быть простым, поскольку имеет по крайней мере один множитель, отличный от единицы и самого числа.

Процесс можно ускорить, выполняя деление только на простые числа и пропуская составные. Как уже говорилось ранее, составные числа есть

не что иное, как произведения простых чисел. Это означает, что если 49 не делится нацело на 2, то такое число, как 6, в число множителей которого входит 2, тоже не может быть делителем числа 49. Иными словами, любое число, которое делится на 6 без остатка, делится без остатка и на 2, поскольку 2 — это множитель числа 6 (рис. 22.1).

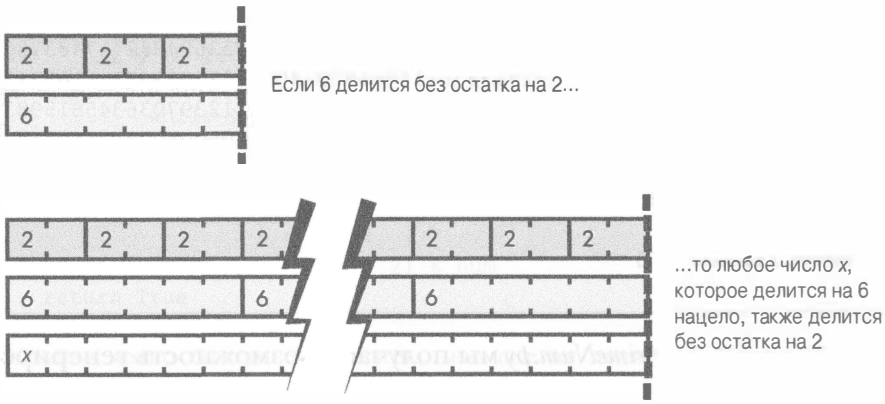


Рис. 22.1. *Любое число, которое делится без остатка на 6, также делится без остатка на 2*

В качестве еще одного примера проверим, является ли число 13 простым:

$13 \div 2 = 6$, остаток 1,

$13 \div 3 = 4$, остаток 1.

Мы должны проверять только числа вплоть до квадратного корня из числа, которое тестируем на простоту. *Квадратным корнем* называют число, результатом умножения которого на самого себя будет исходное число. Например, квадратный корень из 25 равен 5, поскольку $5 \cdot 5 = 25$. Так как число не может иметь двух множителей, превышающих квадратный корень из него, мы можем ограничить процедуру перебора делителей величиной квадратного корня. Квадратный корень из 13 равен приблизительно 3,6, поэтому для того, чтобы определить, является ли число 13 простым, мы проверить лишь делимость на 2 и 3.

В качестве другого примера рассмотрим число 16, квадратный корень из которого равен 4. В результате умножения двух чисел, каждое из которых больше 4, мы всегда получим число, превышающее 16, а любые множители числа 16, бо́льшие 4, всегда будут сочетаться с множителями, меньшими 4, как, например, в случае разложения $8 \cdot 2$. Поэтому в процессе нахождения множителей, не превышающих квадратный корень из тестируемого числа, вы также будете находить множители, превышающие это значение.

В Python для нахождения квадратного корня из заданного числа можно использовать функцию `math.sqrt()`. Чтобы увидеть, как это работает, введите в интерактивной оболочке следующие инструкции.

```
>>> import math
>>> 5 * 5
25
>>> math.sqrt(25)
5.0
>>> math.sqrt(10)
3.1622776601683795
```

Обратите внимание на то, что функция `math.sqrt()` всегда возвращает результат в виде вещественного числа (с плавающей точкой).

Реализация алгоритма перебора делителей

Функция `isPrimeTrialDiv()` получает число в качестве параметра `num` и проверяет его на простоту по методу перебора делителей. Если число оказывается составным, функция возвращает `False`, в противном случае возвращается значение `True`.

```
 7. def isPrimeTrialDiv(num):
 8.     # Возвращает True, если num - простое число, иначе - False
 9.
10.     # Использует алгоритм перебора делителей для проверки простоты числа
11.
12.     # Числа, меньшие 2, не являются простыми
13.     if num < 2:
14.         return False
```

Если в строке 13 выясняется, что значение `num` меньше 2, то функция возвращает `False`, поскольку числа, меньшие 2, не могут быть простыми.

В строке 17 начинается цикл `for`, реализующий алгоритм перебора делителей. Верхняя граница диапазона проверяемых целых чисел задается равной квадратному корню из `num`, который вычисляется с помощью функции `math.sqrt()`. Возвращаемое функцией вещественное значение преобразуется в ближайшее большее целое.

```
16.     # Проверяем делимость на целые числа вплоть до
          квадратного корня из num
17.     for i in range(2, int(math.sqrt(num)) + 1):
18.         if num % i == 0:
19.             return False
20.     return True
```

В строке 18 выполняется деление с остатком (`%`). Если остаток равен 0, значит, параметр `num` нацело делится на `i` и не может быть простым числом, вследствие чего функция сразу же возвращает значение `False`. Если же цикл, начатый в строке 17, все же завершился, то функция возвращает значение `True` (строка 20), указывающее на то, что `num` — простое число.

Реализованный в функции `isPrimeTrialDiv()` алгоритм перебора делителей — не единственный способ тестирования простоты чисел. Простые числа можно также находить с помощью решета Эратосфена.

Решето Эратосфена

Решето Эратосфена — это алгоритм нахождения всех простых чисел в пределах заданного диапазона. Чтобы понять, как работает алгоритм, представьте, что имеется группа ячеек. Ячейки содержат целые числа от 1 до 50, каждое из которых помечено как простое (рис. 22.2).

Простое 1	Простое 2	Простое 3	Простое 4	Простое 5	Простое 6	Простое 7	Простое 8	Простое 9	Простое 10
Простое 11	Простое 12	Простое 13	Простое 14	Простое 15	Простое 16	Простое 17	Простое 18	Простое 19	Простое 20
Простое 21	Простое 22	Простое 23	Простое 24	Простое 25	Простое 26	Простое 27	Простое 28	Простое 29	Простое 30
Простое 31	Простое 32	Простое 33	Простое 34	Простое 35	Простое 36	Простое 37	Простое 38	Простое 39	Простое 40
Простое 41	Простое 42	Простое 43	Простое 44	Простое 45	Простое 46	Простое 47	Простое 48	Простое 49	Простое 50

Рис. 22.2. *Начальная конфигурация решета Эратосфена для чисел от 1 до 50*

В процессе просеивания чисел мы последовательно исключаем из диапазона составные числа до тех пор, пока не останутся лишь простые числа. Поскольку единица не является простым числом, начнем с того, что пометим его как составное. Далее аналогичным образом пометим все четные числа (за исключением 2): 4 (2 · 2), 6 (2 · 3), 8 (2 · 4), 10, 12 и т.д. вплоть до 50 (рис. 22.3).

На следующем шаге мы повторяем весь процесс для чисел, кратных 3 (за исключением самого числа 3), и помечаем как составные числа 6, 9, 12, 15, 18, 21 и т.д. Затем все то же самое повторяется для чисел, кратных 4

(за исключением числа 4), для чисел, кратных 5 (за исключением числа 5), и т.д., пока мы не дойдем до 8. На этом числе следует остановиться, поскольку оно больше, чем 7,071, т.е. квадратный корень из 50. К этому моменту все числа, кратные 9, 10, 11 и т.д., уже будут помечены как составные, так как любой множитель, превышающий квадратный корень из 50, сочетается с одним из множителей, меньших этого значения.

Составное 1	Простое 2	Простое 3	Составное 4	Простое 5	Составное 6	Простое 7	Составное 8	Простое 9	Составное 10
Простое 11	Составное 12	Простое 13	Составное 14	Простое 15	Составное 16	Простое 17	Составное 18	Простое 19	Составное 20
Простое 21	Составное 22	Простое 23	Составное 24	Простое 25	Составное 26	Простое 27	Составное 28	Простое 29	Составное 30
Простое 31	Составное 32	Простое 33	Составное 34	Простое 35	Составное 36	Простое 37	Составное 38	Простое 39	Составное 40
Простое 41	Составное 42	Простое 43	Составное 44	Простое 45	Составное 46	Простое 47	Составное 48	Простое 49	Составное 50

Рис. 22.3. *Исключение единицы и всех четных чисел*

Итоговый вид решета Эратосфена приведен на рис. 22.4, где простые числа находятся в ячейках белого цвета.

Составное 1	Простое 2	Простое 3	Составное 4	Простое 5	Составное 6	Простое 7	Составное 8	Составное 9	Составное 10
Простое 11	Составное 12	Простое 13	Составное 14	Составное 15	Составное 16	Простое 17	Составное 18	Простое 19	Составное 20
Составное 21	Составное 22	Простое 23	Составное 24	Составное 25	Составное 26	Составное 27	Составное 28	Простое 29	Составное 30
Простое 31	Составное 32	Составное 33	Составное 34	Составное 35	Составное 36	Простое 37	Составное 38	Составное 39	Составное 40
Простое 41	Составное 42	Простое 43	Составное 44	Составное 45	Составное 46	Простое 47	Составное 48	Составное 49	Составное 50

Рис. 22.4. *Простые числа, найденные с помощью решета Эратосфена*

Используя решето Эратосфена, мы нашли все простые числа, не превышающие 50: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 и 47. Этот алгоритм наиболее пригоден в тех случаях, когда нужно быстро найти все простые числа в определенном диапазоне. Он работает намного быстрее, чем рассмотренный ранее алгоритм перебора делителей, когда каждое число проверяется по отдельности.

Генерирование простых чисел с помощью решета Эратосфена

Функция `primeSieve()` возвращает список всех простых чисел в диапазоне от 1 до `sieveSize`, используя решето Эратосфена.

```
23. def primeSieve(sieveSize):
24.     # Возвращает список простых чисел, вычисленных
25.     # с помощью решета Эратосфена
26.
27.     sieve = [True] * sieveSize
28.     sieve[0] = False  # 0 и 1 не являются простыми числами
29.     sieve[1] = False
```

В строке 27 создается список булевых значений `True` размером `sieveSize`. Индексам `0` и `1` соответствуют значения `False`, поскольку 0 и 1 не являются простыми числами.

В цикле `for`, который начинается в строке 32, мы проходим по всем целым числам в диапазоне от 2 до квадратного корня из `sieveSize`.

```
31.     # Создаем решето
32.     for i in range(2, int(math.sqrt(sieveSize)) + 1):
33.         pointer = i * 2
34.         while pointer < sieveSize:
35.             sieve[pointer] = False
36.             pointer += i
```

Переменная `pointer` инициализируется первым кратным `i` числом, каковым является число `i * 2` (строка 33). После этого в цикле `while` для элемента списка `sieve` с индексом `pointer` устанавливается значение `False`, а в строке 36 индекс `pointer` изменяется таким образом, чтобы он указывал на следующее число, кратное `i`.

По завершении цикла `for`, начатого в строке 32, список `sieve` будет содержать значения `True` для всех индексов, являющихся простыми числами. Далее мы создаем новый список `primes` (первоначально пустой), про-

ходим по всему списку `sieve` и добавляем в список `primes` значения `i` при условии, что элемент `sieve[i]` равен `True`, т.е. `i` — простое число.

```
38.     # Формируем список простых чисел
39.     primes = []
40.     for i in range(sieveSize):
41.         if sieve[i] == True:
42.             primes.append(i)
```

В строке 44 возвращается список простых чисел:

```
44.     return primes
```

Функция `primeSieve()` позволяет находить все простые числа в пределах небольшого диапазона, тогда как функция `isPrimeTrialDiv()` способна быстро определять, является ли заданное небольшое число простым. Но как быть с большими простыми числами, содержащими сотни цифр?

Даже если передать функции `isPrimeTrialDiv()` относительно большое целое число, то на его проверку уйдет несколько секунд. В случае простых чисел, содержащих сотни цифр, наподобие тех, с которыми мы будем работать в программе шифрования с открытым ключом в главе 23, функция `isPrimeTrialDiv()` будет выполняться более триллиона лет.

В следующем разделе вы узнаете о том, как определить, является ли очень большое число простым, используя тест Миллера — Рабина.

Тест Миллера — Рабина для проверки простоты числа

Основное преимущество теста Миллера — Рабина заключается в том, что на обычном компьютере он выполняется всего за несколько секунд. Но несмотря на относительно компактный код Python, математическое обоснование принципов его работы заняло бы слишком много места, поэтому мы не будем приводить его здесь. Тест Миллера — Рабина является вероятностным. Он позволяет находить *вероятно простые числа*, которые могут оказаться составными. Впрочем, риск ошибиться в данном случае относительно невелик, чего вполне достаточно для целей книги. Чтобы узнать больше о том, как работает этот алгоритм, обратитесь к Википедии (`https://ru.wikipedia.org/wiki/Тест_Миллера_—_Рабина`).

Тест Миллера — Рабина реализован в функции `rabinMiller()`.

```
46. def rabinMiller(num):
47.     # Возвращает True, если num - простое число
48.     if num % 2 == 0 or num < 2:
49.         return False  # тест не работает для четных чисел
50.     if num == 3:
51.         return True
52.     s = num - 1
53.     t = 0
54.     while s % 2 == 0:
55.         # Делим s на 2 до тех пор, пока не получим нечетное
56.         # число (используя t для подсчета таких делений)
57.         s = s // 2
58.         t += 1
59.     for trials in range(5):  # пытаемся фальсифицировать проверку
                                   простоты числа num 5 раз
60.         a = random.randrange(2, num - 1)
61.         v = pow(a, s, num)
62.         if v != 1:  # тест неприменим, если v равно 1
63.             i = 0
64.             while v != (num - 1):
65.                 if i == t - 1:
66.                     return False
67.                 else:
68.                     i = i + 1
69.                     v = (v ** 2) % num
70.     return True
```

Не обязательно понимать, как работает этот код. Важно знать лишь одно: если функция `rabinMiller()` возвращает `True`, то аргумент `num` является вероятно простым числом. Если же возвращаемое значение равно `False`, то число `num` является составным.

Проверка больших простых чисел

Мы создадим еще одну функцию, `isPrime()`, которая будет вызывать функцию `rabinMiller()`. Тест Миллера — Рабина не всегда будет самым эффективным способом проверки простоты числа, поэтому предварительно выполняются несколько быстрых проверок, позволяющих определить, является ли параметр `num` простым числом. Список всех простых чисел, не превышающих 100, хранится в константе `LOW_PRIMES`. Он генерируется с помощью функции `primeSieve()`.

```
72. # Обычно можно быстро определить, что параметр num - составное число,
73. # разделив его на первые несколько десятков простых чисел. Это быстрее,
74. # чем вызывать функцию rabinMiller(), но годится не всегда.
75. LOW_PRIMES = primeSieve(100)
```

Для начала отбрасываем любые числа, меньшие 2 (строки 81 и 82).

```
78. def isPrime(num):
79.     # Возвращает True, если num - простое число. Сначала выполняются
80.     # быстрые проверки перед вызовом функции rabinMiller().
81.     if (num < 2):
82.         return False  # 0, 1 и отрицательные числа не являются простыми
```

Далее можно выполнить быстрый тест, используя список LOW_PRIMES. Безусловно, проверка делимости параметра num на простые числа из этого списка не может служить окончательным критерием того, что параметр является простым числом, но она поможет нам выявлять составные числа. Около 90 процентов больших целых чисел, передаваемых функции isPrime(), могут быть распознаны как составные путем деления на простые числа, не превышающие 100. Это обусловлено тем, что если тестируемое число нацело делится на простое число, такое как 3, то нет необходимости проверять его делимость на составные числа 6, 9, 12, 15 и любые другие, кратные 3. Подобная проверка выполняется гораздо быстрее, чем тест Миллера — Рабина, что позволяет в 90 процентов случаев ускорить работу функции isPrime().

Цикл, начинающийся в строке 84, проходит по всем простым числам из списка LOW_PRIMES.

```
83.     # Проверяем, делится ли параметр num на простые числа из списка
84.     for prime in LOW_PRIMES:
85.         if (num == prime):
86.             return True
87.         if (num % prime == 0):
88.             return False
```

Если параметр num совпадает с любым значением из списка LOW_PRIMES, то очевидно, что он является простым числом, и тогда в строке 86 функция возвращает True.

Далее параметр num делится на каждый элемент списка LOW_PRIMES с помощью оператора деления с остатком (строка 87), и если результат равен 0, то это означает, что prime является делителем num, а значит, num — составное число. В таком случае в строке 88 функция возвращает False.

Это и есть те три быстрые проверки, о которых шла речь выше. Если же цикл завершается и выполнение функции продолжается после строки 88, то тогда вызывается функция rabinMiller(), и ее результат возвращается в качестве результата функции isPrime().

```
89.     # В остальных случаях вызываем функцию rabinMiller()
90.     return rabinMiller(num)
```

Теперь, когда вы знаете, как определить, является ли число простым, мы применим эту функцию для генерирования больших простых чисел, которые понадобятся нам в программе шифрования с открытым ключом в главе 23.

Генерирование больших простых чисел

Функция `generateLargePrime()` использует бесконечный цикл для получения простого числа. Для этого генерируется большое случайное число, сохраняемое в переменной `num`, которая затем передается функции `isPrime()` для проверки на простоту.

```
93. def generateLargePrime(keysize=1024):
94.     # Возвращает случайное простое число размером keysize бит
95.     while True:
96.         num = random.randrange(2**(keysize-1), 2**(keysize))
97.         if isPrime(num):
98.             return num
```

Если `num` — простое число, то в строке 98 функция возвращает `num`. В противном случае функция снова переходит к строке 96 для получения нового случайного числа. Цикл продолжается до тех пор, пока не будет найдено число, определяемое функцией `isPrime()` как простое.

По умолчанию параметр `keysize` функции `generateLargePrime()` равен 1024. Чем выше это значение, тем больше количество возможных ключей и тем труднее взломать шифр методом грубой силы. Обычно длина открытого ключа измеряется в битах, о чем пойдет речь в главах 23 и 24. А пока что достаточно знать, что 1024-битовый ключ — это очень большое число: в нем почти 300 цифр!

Резюме

Простые числа обладают рядом замечательных математических свойств. Как вы узнаете в главе 23, такие числа служат основой шифров, используемых в профессиональных криптографических приложениях. Определение простого числа звучит довольно тривиально: это число, которое нацело делится только на единицу и на само себя. В то же время программный код, позволяющий определять, какие числа являются простыми, далеко не так тривиален.

В этой главе мы написали функцию `isPrimeTrialDiv()`, реализующую тест простоты путем проверки делимости числа на каждое из чисел в интервале от 2 до квадратного корня из данного числа. Такой алгоритм назы-

вают перебором делителей. При делении простого числа на любое число, отличное от единицы и его самого, остаток всегда будет ненулевым. Поэтому получение нулевого остатка означает, что число не является простым.

Вы также узнали о том, что решето Эратосфена позволяет быстро находить все простые числа в заданном диапазоне, но для нахождения больших простых чисел алгоритму потребуется слишком много памяти.

Ввиду того, что решето Эратосфена и алгоритм перебора делителей, реализованные в модуле *primeNum.py*, недостаточно быстро справляются с нахождением больших простых чисел, для шифрования с открытым ключом нужен другой алгоритм, способный работать с числами, содержащими сотни цифр. В качестве одного из способов решения этой проблемы был предложен тест Миллера — Рабина, позволяющий определить простоту числа на основании сложных математических проверок.

В главе 23 мы воспользуемся модулем *primeNum.py* для написания программы шифрования с открытым ключом. Наконец-то мы создадим шифр, который проще в применении, чем одноразовый шифроблокнот, и при этом не может быть взломан с помощью тривиальных методов, рассмотренных в книге!

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. Сколько всего существует простых чисел?
2. Как называются целые числа, не являющиеся простыми?
3. Назовите два алгоритма нахождения простых чисел?

23

ГЕНЕРИРОВАНИЕ КЛЮЧЕЙ ДЛЯ КРИПТОСИСТЕМ С ОТКРЫТЫМ КЛЮЧОМ

"Стоит только начать использовать намеренно скомпрометированную криптографию с заранее предусмотренным «черным ходом», к которому, как предполагается, имеют ключи лишь «хорошие парни», как вы фактически лишаетесь всякой секретности. Зашифрованное таким способом сообщение можно с равным успехом выписать в небе самолетом для воздушной рекламы, так как оно уже заранее взломано".

Кори Доктороу, писатель-фантаст, 2015 г.

У всех рассмотренных ранее шифров есть одно общее свойство: для дешифрования сообщения применяется тот же ключ, которым оно было зашифровано. С этим связана очевидная проблема: как обмениваться сообщениями с людьми, с которыми вы прежде не встречались? Те, кто вас прослушивают, могут перехватить пересылаемый ключ так же легко, как и зашифрованные сообщения.

В этой главе вы познакомитесь с асимметричными криптосистемами, которые позволяют незнакомым людям обмениваться сообщениями, используя открытый и закрытый ключи. Вы узнаете о криптосистеме с открытым ключом, основанной на алгоритме RSA. Учитывая сложность и многоэтапность этого алгоритма, мы напишем две программы. В этой главе будет реализована программа для генерации открытых и закрытых ключей, а в следующей главе мы реализуем вторую программу, предназначенную для шифрования и дешифрования сообщений с использованием сгенерированных нами ключей. Но сначала необходимо разобраться с тем, как работают криптосистемы с открытым ключом.

В этой главе...

- Криптосистемы с открытым ключом
- Аутентификация
- Цифровые подписи
- Атака MITM
- Генерирование открытых и закрытых ключей
- Гибридные системы

Криптосистемы с открытым ключом

Представьте, что с вами хочет связаться кто-то, находящийся на другом конце земли. Вам обоим известно, что электронная почта, обычная переписка и телефонные звонки отслеживаются спецслужбами. Чтобы иметь возможность обмениваться зашифрованными сообщениями, вы оба должны прийти к соглашению относительно секретного ключа. Но если одна сторона вышлет секретный ключ другой стороне, то спецслужбы перехватят его и используют для дешифрования всех последующих сообщений. Тайная встреча с целью передачи ключа тоже невозможна. Можно попытаться зашифровать сам ключ, но для этого потребуется передать другому человеку *секретный ключ для шифрования секретного ключа*, который также будет перехвачен.

В *криптосистемах с открытым ключом* указанная проблема решается за счет использования двух ключей, один из которых служит для шифрования, а другой – для дешифрования. Это пример *асимметричного шифра.* Рассмотренные ранее шифры, в которых для шифрования и дешифрования применяется один и тот же секретный ключ, называются *симметричными.*

Важно понимать, что *сообщение, зашифрованное с помощью ключа шифрования, может быть расшифровано лишь с помощью ключа дешифрования.* Поэтому даже если кто-то завладеет ключом шифрования, он все равно не сможет прочитать исходное сообщение.

Ключ шифрования называют *открытым ключом* (public key), поскольку он доступен для всех. В отличие от этого *закрытый ключ* (private key), т.е. ключ дешифрования, должен храниться в тайне.

Например, если Алиса хочет послать сообщение Бобу, она берет открытый ключ Боба или он передает его ей. После этого Алиса шифрует свое сообщение Бобу, используя его открытый ключ. Поскольку расшифровать сообщение с помощью открытого ключа невозможно, перехват открытого ключа Боба не имеет никакого значения.

Когда Боб получит зашифрованное сообщение от Алисы, он дешифрует его с помощью своего закрытого ключа. Лишь Боб владеет закрытым ключом, с помощью которого можно расшифровать сообщение, зашифрованное с помощью его открытого ключа. Если Боб захочет ответить Алисе, он найдет ее открытый ключ и применит его для шифрования своего ответа. Поскольку закрытый ключ Алисы известен только ей одной, она единственный человек, который сможет дешифровать зашифрованный ответ Боба. Даже если Алиса и Боб находятся на разных концах земли, они смогут обмениваться сообщениями, не опасаясь того, что их перехватят.

Открытый ключ, который мы реализуем в этой главе, основан на алгоритме RSA (Rivest — Shamir — Adleman), предложенном в 1977 году. Название алгоритма образовано первыми буквами фамилий его изобретателей: Рональд Ривест, Ади Шамир и Леонард Адлеман.

В RSA используются большие простые числа, содержащие сотни цифр. Об арифметике простых чисел шла речь в главе 22. В алгоритме открытого ключа генерируются два случайных простых числа, которые затем сложным образом комбинируются (включая операцию модульного обращения, которая обсуждалась в главе 13) для создания открытого и закрытого ключей.

Опасность использования учебного варианта RSA

Несмотря на то что в книге мы не будем писать программу, предназначенную для взлома открытого ключа, имейте в виду, что программа *publicKeyCipher.py*, которую мы создадим в главе 24, не является безопасной. Криптография — непростая наука, и определение того, является ли шифр (вместе с реализующей его программой) действительно безопасным, требует большого опыта.

Рассмотренная в этой главе программа на основе RSA известна как *учебный вариант RSA* (textbook RSA), поскольку, несмотря на технически корректную реализацию RSA с помощью больших простых чисел, она уязвима для взлома. В частности, источником ее уязвимости является использование функций, генерирующих не истинно случайные, а псевдослучайные числа. Кроме того, тест Миллера — Рабина не всегда дает корректные результаты, как объяснялось в главе 22.

Так что, даже если вы сами не сумеете взломать шифротекст, созданный программой *publicKeyCipher.py*, это смогут сделать другие. Перефразируя эксперта по криптографии Брюса Шнайера, кто угодно способен создать криптографический алгоритм, кажущийся сверхнадежным для своего автора, но гораздо труднее разработать алгоритм, который никто не сможет взломать даже после многолетнего криптоанализа. Описанная в книге программа — это всего лишь занимательный проект, цель которого — обучить читателей основам RSA-шифрования. Для защиты файлов следует использовать только профессиональные криптосистемы. Список соответствующих программ (многие из которых бесплатны) доступен по следующему адресу:

`https://en.wikipedia.org/wiki/Comparison_of_disk_encryption_software`

Проблема аутентификации

Несмотря на оригинальность идеи шифрования с открытым ключом, с ней связана одна проблема. Представьте, например, что вы получаете письмо следующего содержания: "Привет! Меня зовут Эммануил Голдстейн, я лидер движения сопротивления. Хочу наладить с вами тайный канал связи. К письму прилагается мой открытый ключ".

Казалось бы, открытый ключ дает уверенность в том, что отправляемые вами сообщения не сможет прочитать никто, кроме того, кто передал вам этот ключ. Но откуда вам знать, что письмо было действительно написано Эммануилом Голдстейном? Вы можете даже не догадываться, что в действительности посылаете сообщения не Эммануилу Голдстейну, а тайному агенту, ведущему двойную игру, чтобы заманить вас в ловушку!

Несмотря на то что шифры с открытым ключом (а фактически все шифры, описанные в книге) обеспечивают *конфиденциальность* (сохранение содержания письма в тайне), они не всегда гарантируют *аутентификацию* отправителя (подтверждение того, что он действительно является личностью, за которую себя выдает).

В случае симметричных шифров это, как правило, не является проблемой, поскольку обмен ключами происходит во время личной встречи. Но в случае шифрования с открытым ключом вам вовсе не обязательно встречаться с владельцем ключа для того, чтобы получить ключ и начать отправлять зашифрованные сообщения. В криптосистемах с открытым ключом аутентификация критически важна.

Для решения задач аутентификации предназначена *инфраструктура открытых ключей* (Public Key Infrastructure — PKI), однако рассмотрение этой темы выходит за рамки книги. Дополнительную информацию можно получить по следующему адресу:

`https://ru.wikipedia.org/wiki/Инфраструктура_открытых_ключей`

Цифровые подписи

Цифровые подписи позволяют подписывать документы электронным способом при помощи шифрования. Чтобы понять, зачем они вообще нужны, рассмотрим пример сообщения электронной почты, которое Алиса посылает Бобу.

> Уважаемый Боб!
>
> Я обязуюсь купить ваш старый поломанный ноутбук за миллион долларов.

Искренне ваша,

Алиса

Для Боба это отличная новость, ведь он давно мечтал избавиться от своего бесполезного ноутбука за любые деньги. Но что если впоследствии Алиса заявит, что не брала на себя никаких обязательств, а полученное Бобом письмо — подделка, которую она не отправляла? В конце концов, автором письма вполне мог быть сам Боб.

Если бы они встретились лично, то Алиса могла бы просто подписать с Бобом договор купли-продажи. Подделать личную подпись не так-то просто, и она могла бы служить определенной гарантией того, что Алиса не нарушит взятых на себя обязательств. Но даже если бы Алиса подписала такой контракт, сфотографировала его на смартфон и выслала Бобу снимок, потенциально она могла бы впоследствии заявить, что изображение было изменено.

RSA (подобно другим криптосистемам с открытым ключом) позволяет не только шифровать послания, но и снабжать *цифровой подписью* файл или текст. Например, Алиса может зашифровать сообщение своим закрытым ключом, создав таким образом шифротекст, который можно дешифровать только с помощью открытого ключа Алисы. Этот шифротекст становится цифровой подписью к файлу. Фактически он даже не является секретным, так как любой человек в мире может прочитать его с помощью общедоступного открытого ключа Алисы. Суть в другом: *шифруя сообщение с помощью своего закрытого ключа, Алиса снабжает сообщение цифровой подписью способом, не допускающим ее подделку*. Поскольку единственным обладателем закрытого ключа является Алиса, только она могла создать такой шифротекст, а значит, она уже не сможет утверждать, что Боб подделал или изменил сообщение!

Гарантия того, что подписант сообщения впоследствии не сможет отрицать своего авторства, называется *невозможность отказа* (non-repudiation).

Далее вы узнаете, почему аутентификация не менее важна, чем стойкость самого шифра.

Остерегайтесь атак MITM

Не меньшую опасность, чем взлом зашифрованного сообщения, представляет собой так называемая *атака посредника* (man-in-the-middle — MITM). В атаках этого типа злоумышленник перехватывает отправленные вами сообщения и ретранслирует их без вашего ведома. Предположим, например, что Эммануил Голдстейн действительно хочет связаться с вами и отправляет незашифрованное сообщение со своим открытым ключом, но маршрутизаторы, контролируемые спецслужбами, перехватывают его.

Спецслужбы могут заменить открытый ключ Эммануила, прикрепленный к письму, своим открытым ключом и переслать сообщение вам. У вас не будет никакой возможности узнать, чей это ключ: Эммануила или спецслужб!

Впоследствии, когда вы будете шифровать свой ответ Эммануилу, вы воспользуетесь открытым ключ спецслужб, а не Эммануила. Спецслужбы смогут перехватить ваше сообщение и расшифровать его, а затем повторно зашифровать, но уже с помощью открытого ключа Эммануила. Аналогичный трюк будет проделан со всеми последующими ответами Эммануила на ваши письма. Суть атаки посредника проиллюстрирована на рис. 23.1.

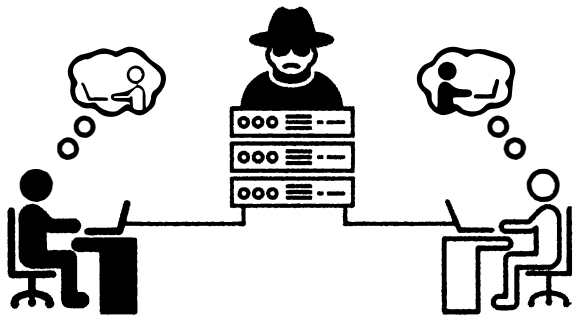


Рис. 23.1. *Атака посредника*

Для вас и Эммануила все будет выглядеть так, словно ваше общение осуществляется втайне от других. Однако спецслужбам будет известно содержимое всех ваших сообщений, поскольку и вы, и Эммануил шифруете свои сообщения открытым ключом, сгенерированным спецслужбами! Опять-таки, описанная проблема возникает лишь по той причине, что шифрование с открытым ключом обеспечивает конфиденциальность, но не аутентификацию. Более подробное обсуждение механизмов аутентификации и инфраструктуры открытых ключей выходит за рамки книги. А нам предстоит рассмотреть способы генерации ключей.

Порядок генерирования открытых и закрытых ключей

В криптосистемах с открытым ключом каждый ключ состоит из двух чисел. Открытым ключом служат числа n и e, закрытым — числа n и d. Эти числа создаются в три этапа.

1. Создайте два очень больших простых числа: p и q. Это должны быть два разных случайных числа. Перемножьте их для получения числа n.
2. Создайте случайное число e, взаимно простое с числом $(p-1)(q-1)$.
3. Вычислите модульное обращение числа e— это будет число d.

Обратите внимание на то, что число *n* используется в обоих ключах. Число *d* должно храниться в тайне, поскольку оно позволяет дешифровывать сообщения. Теперь все готово для написания программы, которая будет генерировать эти ключи.

Исходный код программы Make Public Private Keys

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *makePublicPrivateKeys.py*. Убедитесь в том, что модули *primeNum.py* и *cryptomath.py* находятся в той же самой папке.

makePublicPrivateKeys.py

```
 1. # Генератор ключей для программы шифрования с открытым ключом
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import random, sys, os, primeNum, cryptomath
 5.
 6.
 7. def main():
 8.     # Создаем пару "открытый/закрытый ключ", используя 1024-битовые ключи
 9.     print('Making key files...')
10.     makeKeyFiles('al_sweigart', 1024)
11.     print('Key files made.')
12.
13. def generateKey(keySize):
14.     # Создает открытый и закрытый ключи длиной keySize бит
15.     p = 0
16.     q = 0
17.     # Шаг 1: создаем два простых числа p and q и вычисляем n = p * q
18.     print('Generating p & q primes...')
19.     while p == q:
20.         p = primeNum.generateLargePrime(keySize)
21.         q = primeNum.generateLargePrime(keySize)
22.     n = p * q
23.
24.     # Шаг 2: создаем число e, взаимно простое с (p-1)*(q-1)
25.     print('Generating e that is relatively prime to (p-1)*(q-1)...')
26.     while True:
27.         # Перебираем случайные числа для e, пока не найдем допустимое
28.         e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
29.         if cryptomath.gcd(e, (p - 1) * (q - 1)) == 1:
30.             break
31.
32.     # Шаг 3: вычисляем d - модульное обращение e
```

```
33.     print('Calculating d that is mod inverse of e...')
34.     d = cryptomath.findModInverse(e, (p - 1) * (q - 1))
35.
36.     publicKey = (n, e)
37.     privateKey = (n, d)
38.
39.     print('Public key:', publicKey)
40.     print('Private key:', privateKey)
41.
42.     return (publicKey, privateKey)
43.
44.
45. def makeKeyFiles(name, keySize):
46.     # Создает два файла, 'x_pubkey.txt' и 'x_privkey.txt' (где
47.     # x - значение параметра name), записывая в них целые числа
48.     # keySize, n и e, разделенные запятыми.
49.
50.     # Проверка, предотвращающая перезапись старых файлов ключей
51.     if os.path.exists('%s_pubkey.txt' % (name)) or
           os.path.exists('%s_privkey.txt' % (name)):
52.         sys.exit('WARNING: The file %s_pubkey.txt or %s_privkey.txt
              already exists! Use a different name or delete these files
              and rerun this program.' % (name, name))
53.
54.     publicKey, privateKey = generateKey(keySize)
55.
56.     print()
57.     print('The public key is a %s and a %s digit number.' %
              (len(str(publicKey[0])), len(str(publicKey[1]))))
58.     print('Writing public key to file %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('The private key is a %s and a %s digit number.' %
              (len(str(privateKey[0])), len(str(privateKey[1]))))
65.     print('Writing private key to file %s_privkey.txt...' % (name))
66.     fo = open('%s_privkey.txt' % (name), 'w')
67.     fo.write('%s,%s,%s' % (keySize, privateKey[0], privateKey[1]))
68.     fo.close()
69.
70.
71. # Если файл makePublicPrivateKeys.py выполняется как программа
72. # (а не импортируется как модуль), вызвать функцию main()
73. if __name__ == '__main__':
74.     main()
```

Пример выполнения программы Make Public Private Keys

Запустив программу *makePublicPrivateKeys.py*, мы должны получить примерно следующие результаты (разумеется, числовые значения ключей будут другими, ведь они генерируются случайным образом).

```
Making key files...
Generating p & q primes...
Generating e that is relatively prime to (p-1)*(q-1)...
Calculating d that is mod inverse of e...
Public key: (21090240631670050240196849140657941740509039675461692613581
62121611619133808656784074598753554688979280723862705107204438273246714 35
89327485839374968506241167761472418211520269463228768694043944839222024 07
82167286424247892081318269900084735267117442965485638667684542514049519 60
80522468242549897523048895590808649185211634877784953627068508544697095 29
15640050522212204221803744494065881010331486468305317449607027888478777031
57299597899947132653113276637761677100770183400366683066126657594172078 45
82347990344057272406812521100232929833871861585954209372109725826359561 74
82450199200740185492044687913001143150561170 93, 17460230769175161021731 84
54592368335538324039108691290549542003736785809352476066222657643882357 52
17665473780584902300654473289630868551366950991745119582261139809895130 66
76600958889189564599581456460070270393693277683404354811575681605990659 14
53170741270845572335375041024799371425300216777273298110097435989)
Private key: (21090240631670050240196849140657941740509039675461692613581
06212161161913380865678407459875355468897928072386270510720443827324671 43
58932748583937496850624116776147241821152026946322876869404394483922202 40
78216728642424789208131826990008473526711744296548563866768454251404951 96
08052246824254989752304889559080864918521163487778495362706850854469709 52
91564005052221220422180374449406588101033148646830531744960702788478777 03
15729959789994713265311327663776167710077018340036668306612665759417207 84
58234799034405727240681252110023292983387186158595420937210972582635956 17
48245019920074018549204468791300114315056117093, 47676735798137710412166 8
84916983765043173120289416904341295971552286870991874666099933371008075 94
85490085512247606959426669624659681689954049939345083990142830537106767 60
83594890231288863993840268618707505236077306236416266427614496565255854 53
31166681735980981384493349313058750259417683727029633484451911396358260 00
81812237348621325648807719289311925724810779425681884603640028673273135 29
28311701786142068171658028122915283195622006250825572616804708456070635 96
01833919317974375031636011432177696164717000025430368269905397390574746 42
78541693387849989701477748140737132805300183808531444354584521908724954 46
63398589)

The public key is a 617 and a 309 digit number.
Writing public key to file al_sweigart_pubkey.txt...

The private key is a 617 and a 616 digit number.
Writing private key to file al_sweigart_privkey.txt...
Key files made.
```

Поскольку оба ключа очень большие, они записываются в отдельные файлы: *al_sweigart_pubkey.txt* и *al_sweigart_privkey.txt*. Оба файла будут использоваться в главе 24 для шифрования и дешифрования сообщений. Имена файлов формируются на основе строки `'al_sweigart'`, передаваемой функции `makeKeyFiles()` в строке 10. Вы сможете указать другие имена файлов, передав другую строку.

Если вновь запустить программу *makePublicPrivateKeys.py* и передать функции `makeKeyFiles()` ту же строку, то программа выдаст следующее.

```
Making key files...
WARNING: The file al_sweigart_pubkey.txt or al_sweigart_privkey.txt already
exists! Use a different name or delete these files and rerun this program.
```

Это предупреждение предназначено для того, чтобы мы не могли случайно перезаписать наши файлы ключей, в результате чего восстановление файлов, зашифрованных с их помощью, станет невозможным. Обязательно позаботьтесь о сохранности файлов ключей!

Создание функции `main()`

При запуске программы *makePublicPrivateKeys.py* вызывается функция `main()`, создающая файлы открытого и закрытого ключей с помощью функции `makeKeyFiles()`, которую мы вскоре определим.

```
 7. def main():
 8.     # Создаем пару "открытый/закрытый ключ", используя 1024-битовые ключи
 9.     print('Making key files...')
10.     makeKeyFiles('al_sweigart', 1024)
11.     print('Key files made.')
```

Поскольку для генерирования ключей компьютеру потребуется определенное время, в строке 9 выводится сообщение перед вызовом функции `makeKeyFiles()`, чтобы пользователь был в курсе происходящего.

Функция `makeKeyFiles()`, которой в строке 10 передается строка `'al_sweigart'` и целое число `1024`, генерирует ключи длиной 1024 бита и сохраняет их в файлах *al_sweigart_pubkey.txt* и *al_sweigart_privkey.txt*. Чем длиннее ключ, тем больше существует возможных ключей и тем выше криптографическая стойкость шифра. Однако увеличение длины ключа означает, что шифрование и дешифрование сообщений будут выполняться дольше. В качестве компромиссного решения, обеспечивающего разумный баланс между скоростью и стойкостью шифра, в книге используются ключи длиной 1024 бита. На практике, чтобы обеспечить надежность шифрования с

помощью открытого ключа, необходимо использовать ключи длиной 2048 или даже 3072 бита.

Генерирование ключей с помощью функции generateKey()

Для создания ключей прежде всего необходимо получить два случайных простых числа p и q. Эти числа должны быть достаточно большими и отличаться друг от друга.

```
13. def generateKey(keySize):
14.     # Создает открытый и закрытый ключи длиной keySize бит
15.     p = 0
16.     q = 0
17.     # Шаг 1: создаем два простых числа p and q и вычисляем n = p * q
18.     print('Generating p & q primes...')
19.     while p == q:
20.         p = primeNum.generateLargePrime(keySize)
21.         q = primeNum.generateLargePrime(keySize)
22.     n = p * q
```

Функция `generateLargePrime()` из программы *primeNum.py* (см. главу 22) возвращает два простых числа (строки 20 и 21), которые сохраняются в переменных `p` и `q`. Цикл продолжается до тех пор, пока переменные `p` и `q` не будут различаться. Переменная `keySize` определяет размеры чисел `p` и `q` в битах. В строке 22 мы перемножаем `p` и `q` и сохраняем их произведение в переменной `n`.

На следующем этапе вычисляется другая часть открытого ключа: e.

Вычисление значения e

Значение e вычисляется путем нахождения числа, являющегося взаимно простым с произведением $(p - 1)(q - 1)$. Мы не будем вдаваться в подробности того, почему e вычисляется именно так, а лишь отметим, что это необходимо для получения уникального шифротекста.

Число, взаимно простое с произведением чисел $p - 1$ и $q - 1$, вычисляется с помощью бесконечного цикла, который начинается в строке 26.

```
24.     # Шаг 2: создаем число e, взаимно простое с (p-1)*(q-1)
25.     print('Generating e that is relatively prime to (p-1)*(q-1)...')
26.     while True:
27.         # Перебираем случайные числа для e, пока не найдем допустимое
28.         e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
29.         if cryptomath.gcd(e, (p - 1) * (q - 1)) == 1:
30.             break
```

Функция `random.randrange()`, вызываемая в строке 28, возвращает случайное целое число и сохраняет его в переменной `e`. Проверка взаимной простоты `e` и `(p - 1) * (q - 1)` выполняется с помощью функции `gcd()` из модуля `cryptomath` (строка 29). В случае положительного результата проверки инструкция `break` в строке 30 прерывает бесконечный цикл. Иначе программа возвращается к строке 26 и продолжает генерировать различные случайные числа до тех пор, пока не будет найдено число, взаимно простое с `(p - 1) * (q - 1)`.

Далее мы переходим к вычислению закрытого ключа.

Вычисление значения d

Третий этап заключается в нахождении другой части закрытого ключа, т.е. числа d, которое представляет собой модульное обращение числа e. Для этого у нас уже есть соответствующая функция `findModInverse()` из модуля `cryptomath` (см. главу 13).

Функция `findModInverse()` вызывается в строке 34 и сохраняет результат в переменной `d`.

```
32.     # Шаг 3: вычисляем d - модульное обращение e
33.     print('Calculating d that is mod inverse of e...')
34.     d = cryptomath.findModInverse(e, (p - 1) * (q - 1))
```

Теперь в нашем распоряжении имеются все числа, необходимые для получения открытого и закрытого ключей.

Возврат ключей

Вспомните, что в криптосистемах с открытым ключом открытый и закрытый ключи состоят из двух чисел каждый. Целые числа n и e образуют открытый ключ, а целые числа n и d — закрытый. В строках 36 и 37 эти пары чисел сохраняются в виде кортежей в переменных `publicKey` и `privateKey`.

```
36.     publicKey = (n, e)
37.     privateKey = (n, d)
```

Далее значения ключей выводятся на экран.

```
39.     print('Public key:', publicKey)
40.     print('Private key:', privateKey)
41.
42.     return (publicKey, privateKey)
```

В строке 42 функция `generateKey()` возвращает кортеж, содержащий значения `publicKey` и `privateKey`.

Сгенерировав открытый и закрытый ключи, мы должны сохранить их в файлах, чтобы программа шифрования с открытым ключом могла впоследствии использовать их для шифрования и дешифрования сообщений. Польза от сохранения ключей в файлах состоит в том, что оба целочисленных значения, образующих каждый из ключей, насчитывают сотни цифр, поэтому их невозможно запомнить и слишком неудобно записывать на бумаге.

Создание файлов ключей с помощью функции `makeKeyFiles()`

Функция `makeKeyFiles()` сохраняет ключи в файлах, получая префикс имени файла и длину ключа в качестве параметров.

```
45. def makeKeyFiles(name, keySize):
46.     # Создает два файла, 'x_pubkey.txt' и 'x_privkey.txt' (где
47.     # x - значение параметра name), записывая в них целые числа
48.     # keySize, n и e, разделенные запятыми.
```

Данная функция создает два файла с именами *<name>_pubkey.txt* и *<name>_privkey.txt*, в которых сохраняются открытый и закрытый ключи соответственно. Передаваемый функции параметр `name` определяет часть *<name>* имен файлов.

Чтобы избежать случайного удаления файлов ключей вследствие повторного запуска программы, в строке 51 проверяется существование файлов открытого или закрытого ключей с заданными именами. Если такие файлы существуют, то программа завершает работу с выводом предупреждающего сообщения.

```
50.     # Проверка, предотвращающая перезапись старых файлов ключей
51.     if os.path.exists('%s_pubkey.txt' % (name)) or
           os.path.exists('%s_privkey.txt' % (name)):
52.         sys.exit('WARNING: The file %s_pubkey.txt or %s_privkey.txt
              already exists! Use a different name or delete these files
              and rerun this program.' % (name, name))
```

В случае успешного прохождения проверки в строке 54 вызывается функция `generateKey()`, которой передается параметр `keySize`, задающий длину открытого и закрытого ключей.

```
54.     publicKey, privateKey = generateKey(keySize)
```

Функция `generateKey()` возвращает кортеж, состоящий из двух кортежей, который присваивается переменным `publicKey` и `privateKey` с помощью операции группового присваивания. Первый кортеж содержит два целых числа, образующих открытый ключ, а второй — два целых числа для закрытого ключа.

Теперь можно приступать к созданию файлов ключей, в которых будут храниться числа, образующие каждый из ключей.

В строках 56–58 на экран выводится информация об открытом ключе.

```
56.     print()
57.     print('The public key is a %s and a %s digit number.' %
              (len(str(publicKey[0])), len(str(publicKey[1]))))
58.     print('Writing public key to file %s_pubkey.txt...' % (name))
```

Количество цифр в целых числах, хранящихся в элементах `publicKey[0]` и `publicKey[1]`, вычисляется в строке 57 путем преобразования этих чисел в строки с помощью функции `str()` и последующего определения длины строки с помощью функции `len()`. В строке 58 выводится сообщение о том, что открытый ключ записывается в файл.

В файл ключа записывается длина ключа, целое число `n` и целое число `e` (или `d`, в зависимости от типа ключа). Все значения разделяются запятыми. Ниже приведен пример файла *al_sweigart_pubkey.txt*.

```
1024,141189561571082936553468080511334338940916460395383120069233997353624
936052632037024975858937767170032863262291343040782042107289959628094482
332820877264418337183564774740424053363328720752073346965353041022569818O
493180588850258751531087325796653837774040742213790777243761337634294037 4
815839154897315760145075243071401233858428232725214391295151698044147558 4
541848071057874195191193439532768366941466140613308723567669334421693582O
895371023187272948699479259510582006935116306633036219116343447342195108 2
966346860965671789280887020440983279967498480147232734401682910892741619 4
333747039996892015365564628028293530 73,100718103971294791099836725874012 5
463706809260121858057654010522762625823857151597753664461626599485597536 4
767266381161481376979016411453129317520302962042724371959946895855174563 6
665558941526164523429965489703529940030465646848449715020479155556561228 6
772112515985605028550234129043360222306347259730569900 69
```

В строках 59–61 файл *<name>_pubkey.txt* открывается для записи, в него записываются компоненты открытого ключа, после чего файл закрывается.

```
59.     fo = open('%s_pubkey.txt' % (name), 'w')
60.     fo.write('%s,%s,%s' % (keySize, publicKey[0], publicKey[1]))
61.     fo.close()
```

В строках 63–68 выполняется все то же самое, что и в строках 56–61, за исключением того, что в файл *<name>_privkey.txt* записывается не открытый, а закрытый ключ.

```
63.     print()
64.     print('The private key is a %s and a %s digit number.' %
              (len(str(privateKey[0])), len(str(privateKey[1]))))
65.     print('Writing private key to file %s_privkey.txt...' % (name))
66.     fo = open('%s_privkey.txt' % (name), 'w')
67.     fo.write('%s,%s,%s' % (keySize, privateKey[0], privateKey[1]))
68.     fo.close()
```

Убедитесь в том, что никто не сможет получить доступ к вашему компьютеру и скопировать файлы ключей. Завладев файлом закрытого ключа, любой человек сможет дешифровать все ваши сообщения!

Вызов функции `main()`

Если файл *makePublicPrivateKeys.py* запускается как программа, а не импортируется в виде модуля другой программой, то в строках 73 и 74 вызывается функция `main()`.

```
71. # Если файл makePublicPrivateKeys.py выполняется как программа
72. # (а не импортируется как модуль), вызвать функцию main()
73. if __name__ == '__main__':
74.     main()
```

Теперь в вашем распоряжении имеется программа, генерирующая открытый и закрытый ключи. Шифрование с открытым ключом — удобная технология передачи сообщений между двумя людьми, у которых нет возможности обменяться секретными ключами. Но она обладает целым рядом недостатков, из-за которых шифры такого типа чаще всего применяются в рамках гибридных криптосистем.

Гибридные криптосистемы

Криптосистемы с открытым ключом и, в частности, алгоритм RSA требуют больших вычислительных ресурсов. Проблема особенно остро стоит в случае серверов, которые должны устанавливать тысячи зашифрованных соединений в секунду. Асимметричную криптографию имеет смысл применять для шифрования и распространения ключа гораздо более быстрого *симметричного шифра*, т.е. шифра, в котором для шифрования и дешифрования используется один и тот же секретный ключ. После того

как ключ симметричного шифра будет безопасно передан получателю посредством сообщения, зашифрованного открытым ключом, отправитель сможет шифровать все последующие сообщения симметричным шифром. Системы, в которых одновременно применяется как симметричное, так и асимметричное шифрование, называются *гибридными*. Более подробную информацию о гибридных криптосистемах можно найти в Википедии:

`https://ru.wikipedia.org/wiki/Гибридная_криптосистема`

Резюме

В этой главе вы узнали о том, как работает шифрование с открытым ключом и как написать программу, генерирующую открытый и закрытый ключи. В главе 24 мы используем эти ключи для шифрования и дешифрования сообщений в рамках криптосистемы с открытым ключом.

Контрольные вопросы

Ответы на контрольные вопросы приведены в приложении Б.

1. В чем разница между симметричными и асимметричными шифрами?
2. Алиса сгенерировала открытый и закрытый ключи. К сожалению, впоследствии она потеряла закрытый ключ.
   А. Смогут ли другие люди отправлять ей зашифрованные сообщения?
   Б. Сможет ли она дешифровать сообщения, отправленные ей ранее?
   В. Сможет ли она снабжать документы цифровой подписью?
   Г. Смогут ли другие люди верифицировать подписанные ею ранее документы?
3. Что такое аутентификация и конфиденциальность? В чем разница между ними?
4. Что такое невозможность отказа?

24

ПРОГРАММА ШИФРОВАНИЯ С ОТКРЫТЫМ КЛЮЧОМ

"Коллега однажды сказал мне, что в мире полно дырявых систем безопасности, спроектированных теми, кто читал мою книгу".

Брюс Шнайер, автор книги "Прикладная криптография"

В главе 23 вы познакомились с асимметричным шифрованием и узнали, как написать программу, генерирующую файлы открытого и закрытого ключей. Теперь у вас есть возможность переслать свой открытый ключ другим людям (или опубликовать его у себя на сайте), чтобы они могли шифровать им отправляемые вам сообщения. В этой главе мы напишем программу, способную шифровать и дешифровать сообщения с помощью сгенерированных вами ключей.

Предупреждение

Реализованная в книге криптосистема с открытым ключом основана на алгоритме RSA. По целому ряду причин, которые здесь не обсуждаются, такой шифр непригоден с практической точки зрения. Несмотря на то что он не поддается взлому с помощью методов криптоанализа, рассмотренных в книге, он оказывается уязвимым для более совершенных методов взлома, применяемых профессиональными криптографами в наши дни.

В этой главе...

- Функция `pow()`
- Функции `min()` и `max()`
- Списковый метод `insert()`

Как работают криптосистемы с открытым ключом

Как и все рассмотренные нами ранее шифры, асимметричный шифр с открытым ключом преобразует символы в числа, над которыми затем выполняются различные математические операции. Однако принципиальная особенность данного шифра, отличающая его от других шифров, заключается в том, что исходное сообщение разбивается на группы символов, каждая из которых преобразуется в одно число, называемое *блоком*, а далее уже эти блоки шифруются по одному за раз.

У блочного алгоритма есть вполне разумное обоснование. Если применять шифрование с открытым ключом к отдельным символам, то одни и те же символы открытого текста всегда будут преобразовываться в одни и те же символы шифротекста, в результате чего такой шифр ничем не будет отличаться от простого подстановочного шифра.

Создание блоков

В криптографии *блок* — это большое целое число, соответствующее фиксированному количеству символов сообщения. Программа *publicKeyCipher.py*, которую мы напишем в данной главе, преобразует строку сообщения в блоки, где каждый блок представляет собой целое число, замещающее 169 символов. Максимальный размер блока зависит от размера символьного набора и длины ключа. В нашей программе набором символов, из которых может состоять сообщение, служит 66-символьная строка `'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'`.

При этом должно выполняться следующее соотношение: $2^{длина\_ключа} > размер\_набора\_символов^{размер\_блока}$. Например, если используется 1024-битовый ключ и набор, включающий 66 символов, то максимальным размером блока будет целое число, не превышающее 169, поскольку 2^{1024} больше, чем 66^{169}, но меньше, чем 66^{170}. Если попытаться использовать слишком большие блоки, то арифметика шифрования с открытым ключом не сработает, и вы не сможете дешифровать шифротекст, созданный программой.

Рассмотрим, как преобразовать строку сообщения в большой целочисленный блок.

Преобразование строки в блок

Как и в предыдущих программах шифрования, для преобразования символов в целые числа и наоборот мы можем использовать индекс строки символьного набора. Мы будем хранить символьный набор в константе `SYMBOLS`, в которой символом с индексом 0 будет `'A'`, символом с индексом 1 — `'B'` и т.д. Чтобы увидеть, как работает подобное преобразование, введите в интерактивной оболочке следующие инструкции.

```
>>> SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
1234567890 !?.'
>>> len(SYMBOLS)
66
>>> SYMBOLS[0]
'A'
>>> SYMBOLS[30]
'e'
```

Символы в наборе можно представить их целочисленными индексами. Но нам нужно каким-то образом объединить эти небольшие числа в одно большое число, представляющее блок.

Для создания блока мы умножаем индекс символа в наборе на размер символьного набора, возводимый в постоянно увеличивающуюся степень. Сумма всех таких чисел и будет блоком. В качестве примера рассмотрим объединение небольших чисел в один большой блок для строки `'Howdy'`.

Целое число для блока начинается с `0`, а размер набора составляет 66 символов. Введите в интерактивной оболочке следующие инструкции.

```
>>> blockInteger = 0
>>> len(SYMBOLS)
66
```

Первый символ в строке `'Howdy'` — `'H'`. Найдем его индекс.

```
>>> SYMBOLS.index('H')
7
```

Поскольку это начальный символ сообщения, мы умножаем его индекс в символьном наборе на 66^0 (в Python оператор возведения в степень — `**`), в результате чего получаем значение `7`, которое прибавляется к блоку.

```
>>> 7 * (66 ** 0)
7
>>> blockInteger = blockInteger + 7
```

Далее находим аналогичный индекс для `'o'` — второго символа в строке `'Howdy'`. Так как это второй символ в сообщении, умножаем его индекс в символьном наборе на 66^1, а не на 66^0, и прибавляем результат к блоку.

```
>>> SYMBOLS.index('o')
40
>>> blockInteger += 40 * (66 ** 1)
>>> blockInteger
2647
```

Теперь целочисленное значение блока равно `2647`. Мы можем сократить процесс нахождения индекса каждого символа, используя одну строку кода.

```
>>> blockInteger += SYMBOLS.index('w') * (len(SYMBOLS) ** 2)
>>> blockInteger += SYMBOLS.index('d') * (len(SYMBOLS) ** 3)
>>> blockInteger += SYMBOLS.index('y') * (len(SYMBOLS) ** 4)
>>> blockInteger
957285919
```

В результате кодирования строки `'Howdy'` в один большой целочисленный блок мы получаем целое число 957 285 919, уникальным образом представляющее эту строку. Продолжая возводить число 66 во все бо́льшую степень, мы можем сформировать большое целое число для замещения строки любой длины вплоть до размера блока. Например, блок 277 981 представляет строку `'42!'`, а блок 10 627 106 169 278 065 987 481 042 235 655 809 080 528 — строку `'I named my cat Zophie.'`.

В связи с тем что размер нашего блока равен 169, в одном блоке можно зашифровать не более 169 символов. Если длина сообщения, которое мы хотим закодировать, превышает 169 символов, придется создавать дополнительные блоки. В программе *publicKeyCipher.py* блоки будут разделяться запятыми, чтобы пользователь видел, где заканчивается один блок и начинается следующий.

В табл. 24.1 приведен пример сообщения, разбитого на блоки, и показаны целые числа каждого блока. В блоке может храниться не более 169 символов сообщения.

Табл. 24.1. Разбивка сообщения на блоки

	Сообщение	Целое число блока
1-й блок (169 символов)	Alan Mathison Turing was a British cryptanalyst and computer scientist. He was highly influential in the development of computer science and provided a formalisation of	30138103381200276581206111663322701590471547608326152595431391575797140707837485089852659286061395648657712401264848061468979996871106525448961558640277994456848107158423162065952633246425985956987627719631460939256595688769305982915401292341459466451137309352608735432166613773623460986403811099485392482698
2-й блок (169 символов)	the concepts of algorithm and computation with the Turing machine. Turing is widely considered to be the father of computer science and artificial intelligence. During W	11068907809221474552159350801956343731326801027081927136514840854754026777527919580758722720267087026340702811097095557610085841376819190225258032442691476944762174257333902148064107269871669093655004577014280290424452471175143504911739898604483879159731507893719486011257479801658756445279245156715863348631
3-й блок (82 символа)	orld War II he worked for the Government Code and Cypher School at Bletchley Park.	15836797549616019144289524472175836978758376359748641280475094390565590227320959180772905419448598090532869157642283268874950952770993574179907697903 4

В данном примере 420-символьное сообщение состоит из двух блоков по 169 символов каждый и одного блока, содержащего оставшиеся 82 символа.

Арифметика шифрования и дешифрования с открытым ключом

Теперь, когда вы знаете, как преобразовывать символы в целочисленные блоки, рассмотрим математические операции, лежащие в основе шифрования блоков с помощью открытого ключа.

Вот так выглядят уравнения, описывающие процесс шифрования с открытым ключом:

$$C = M^e \bmod n,$$

$$M = C^d \bmod n.$$

Первое уравнение используется для шифрования каждого целочисленного блока, второе — для дешифрования. M — это целочисленный блок сообщения, C — целочисленный блок шифротекста. Числа e и n образуют открытый ключ, а числа d и n — закрытый ключ. Вспомните о том, что любой человек, в том числе и криптоаналитик, имеет свободный доступ к открытому ключу (e, n).

Мы создаем зашифрованное сообщение путем возведения каждого целочисленного блока в степень e, как это делалось в предыдущем разделе, и деления полученного результата по модулю n. Это дает нам целое число, представляющее зашифрованный блок C. Объединение всех блоков позволяет получить зашифрованное сообщение.

Предположим, например, что необходимо зашифровать пятисимвольную строку `'Howdy'` и отправить ее Алисе. После преобразования в целочисленный блок сообщение принимает следующий вид: `[957285919]` (данное сообщение целиком умещается в одном блоке, поэтому список содержит всего одно значение). Открытый ключ Алисы имеет длину 64 бита. Этого недостаточно для того, чтобы считать его надежным, но в нашем примере так проще и нагляднее. Числа n и e равны 116 284 564 958 604 315 258 674 918 142 848 831 759 и 13 805 220 545 651 593 223 соответственно. (В случае 1024-битовых ключей эти числа были бы намного больше.)

Шифруя сообщение, мы вычисляем результат операции

$$(957\,285\,919^{13\,805\,220\,545\,651\,593\,223}) \%$$
$$116\,284\,564\,958\,604\,315\,258\,674\,918\,142\,848\,831\,759$$

путем передачи этих чисел функции `pow()`.

```
>>> pow(957285919, 13805220545651593223,
      116284564958604315258674918142848831759)
43924807641574602969334176505118775186
```

Для быстрого вычисления столь большой степени функция `pow()` выполняет *возведение в степень по модулю*. Вычисление выражения `(957285919 ** 13805220545651593223) % 116284564958604315258674918142848831759` обычным способом дало бы тот же самый результат, но длилось бы несколько часов. Функция `pow()` возвращает целочисленный блок, представляющий зашифрованное сообщение.

Чтобы дешифровать зашифрованное сообщение, его получатель, владеющий закрытым ключом (d, n), должен возвести целочисленное значение каждого блока в степень d и выполнить деление по модулю n. Декодирование всех дешифрованных блоков в символы с их последующим объединением позволяет получить исходное сообщение.

Пусть, например, Алиса пытается расшифровать целочисленный блок 43 924 807 641 574 602 969 334 176 505 118 775 186. Компоненты n ее открытого и закрытого ключей совпадают, а компонента d закрытого ключа равна 72 424 475 949 690 145 396 970 707 764 378 340 583. Для дешифрования необходимо выполнить следующую операцию.

```
>>> pow(43924807641574602969334176505118775186,
      72424475949690145396970707764378340583,
      116284564958604315258674918142848831759)
957285919
```

Преобразовав целочисленный блок `957285919` в строку, мы получим `'Howdy'`, т.е. исходное сообщение. Сейчас вы узнаете, как преобразовать целочисленный блок в строку.

Преобразование блока в строку

Чтобы дешифровать блок в исходный целочисленный блок, прежде всего необходимо преобразовать его в небольшие целые числа, каждое из которых соответствует исходному символу. Процесс начинается с последнего символа, добавленного в блок. Для вычисления чисел, соответствующих отдельным символам, используется деление с округлением вниз и деление с остатком.

Вспомните, что в предыдущем примере целочисленное значение блока для строки `'Howdy'` было равно `957285919`. Исходное сообщение содержит пять символов, а значит, индекс последнего символа в сообщении равен 4. Размер символьного набора сообщения равен 66. Чтобы определить индекс последнего символа строки сообщения в символьном наборе, мы делим число 957 285 919 на 66^4 с округлением вниз, получая 50. Для округления вниз можно использовать оператор целочисленного деления (`//`). В символьном наборе элементом с индексом 50 (`SYMBOLS[50]`) будет символ `'y'`, который действительно является последним символом в строке `'Howdy'`.

В интерактивной оболочке эти вычисления можно выполнить с помощью следующих инструкций.

```
>>> blockInteger = 957285919
>>> SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
1234567890 !?.'
>>> blockInteger // (66 ** 4)
50
>>> SYMBOLS[50]
'y'
```

Следующий шаг — получение остатка от деления целочисленного значения блока на 66^4, что дает нам очередной целочисленный блок. Результат операции 957 285 919 % 66^4 равен 8 549 119, что соответствует целочисленному блоку для строки `'Howd'`. Последний символ этого блока можно определить путем деления на 66^3 с округлением вниз. Для этого введите в интерактивной оболочке следующие инструкции.

```
>>> blockInteger = 8549119
>>> SYMBOLS[blockInteger // (len(SYMBOLS) ** 3)]
'd'
```

Итак, в этом блоке последним символом оказался `'d'`, и преобразованная строка приобретает вид `'dy'`. Удалить этот символ из целочисленного значения блока можно тем же способом, что и раньше.

```
>>> blockInteger = blockInteger % (len(SYMBOLS) ** 3)
>>> blockInteger
211735
```

Число `211735` — это блок для строки `'How'`. Продолжая в том же духе, мы сможем восстановить всю строку.

```
>>> SYMBOLS[blockInteger // (len(SYMBOLS) ** 2)]
'w'
>>> blockInteger = blockInteger % (len(SYMBOLS) ** 2)
>>> SYMBOLS[blockInteger // (len(SYMBOLS) ** 1)]
'o'
>>> blockInteger = blockInteger % (len(SYMBOLS) ** 1)
>>> SYMBOLS[blockInteger // (len(SYMBOLS) ** 0)]
'H'
```

Теперь вы знаете, как извлечь символы строки `'Howdy'` из целочисленного блока `957285919`.

Почему нельзя взломать сообщение, зашифрованное с помощью открытого ключа

Различные типы криптографических атак, которые мы применяли ранее, оказываются бессильными против шифрования с открытым ключом, если оно корректно реализовано. Вот лишь несколько причин.

1. Атака методом грубой силы не работает ввиду слишком большого количества возможных ключей, подлежащих перебору.
2. Перебор по словарю не работает, поскольку ключи основаны на числах, а не на словах.

3. Атака с использованием шаблонов слов не работает, поскольку одно и то же слово исходного текста может быть зашифровано поразному, в зависимости от того, где именно оно встречается в блоке.
4. Частотный анализ не работает, поскольку каждый зашифрованный блок соответствует целой группе символов, что делает невозможным использованием счетчиков вхождений отдельных символов.

Поскольку открытый ключ (e, n) общедоступен, в случае перехвата шифротекста крипотаналитику будут известны числа e, n и C. Но, не зная число d, математически невозможно вычислить M, т.е. исходное сообщение.

Вспомните из главы 23, что e — это число, взаимно простое с числом $(p - 1)(q - 1)$, а d представляет собой модульное обращение чисел e и $(p-1)(q-1)$. В главе 13 вы узнали о том, что модульное обращение двух чисел вычисляется путем нахождения числа i, удовлетворяющего уравнению $(a \cdot i) \% m = 1$, где a и m входят в выражение $a \bmod m$. Таким образом, криптоаналитику известно, что d— это модульное обращение $e \bmod (p-1)(q-1)$, и он мог бы получить полный ключ дешифрования, решив уравнение $(e \cdot d) \bmod (p-1)(q-1) = 1$, но дело в том, что не существует способа определить, что собой представляет произведение $(p-1)(q-1)$.

Размеры ключей известны из файла открытого ключа, поэтому криптоаналитик изначально знает, что p и q меньше числа 2^{1024}, а e— число, взаимно простое с произведением $(p-1)(q-1)$. Однако e обладает этим свойством в отношении *множества* чисел, поэтому нахождение всех возможных произведений $(p-1)(q-1)$ чисел из диапазона от 0 до 2^{1024} — слишком масштабная задача для того, чтобы ее можно было решить методом грубой силы.

Несмотря на недостаточность данных для взлома шифра, криптоаналитик может получить определенную подсказку благодаря открытому ключу, который состоит из двух чисел (e, n). Нам известно, что $n = p \cdot q$, поскольку именно так мы вычисляли n, когда создавали открытый и закрытый ключи в главе 23. А поскольку p и q — простые числа, то для заданного n существуют ровно два числа, которые могут быть компонентами p и q.

Вспомните, что простое число не имеет никаких других делителей, кроме единицы и самого себя. Следовательно, если перемножить два простых числа, то делителями полученного произведения будут единица, само это произведение и два искомых простых числа.

Таким образом, все, что требуется сделать для взлома шифра с открытым ключом, — это найти множители числа n. Количество возможных вариантов, предположительно, не так уж и велико, ведь нам известно, что существуют всего два простых числа, произведение которых равно n. Определив числа p и q, мы сможем вычислить произведение $(p-1)(q-1)$ и использовать его для нахождения числа d. Процедура кажется не слишком

сложной. Для выполнения необходимых расчетов мы воспользуемся функцией `isPrime()` из программы *primeNum.py*, которую написали в главе 22.

Функцию `isPrime()` можно видоизменить таким образом, чтобы она возвращала первые из найденных множителей, поскольку нам известно, что для n могут существовать лишь два делителя, отличных от единицы и самого числа n.

```
def isPrime(num):
    # Возвращает (p,q), где p и q - делители num.

    # Проверяем, делится ли num на какое-либо из
    # чисел, меньших квадратного корня из num
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return (i, num / i)
    return None  # делители не найдены, значит, num - простое число
```

Если бы мы захотели написать программу для взлома шифра с открытым ключом, то могли бы просто вызывать эту функцию, передавать ей число n (которое можно взять из файла открытого ключа) и ждать, пока она не найдет множители p и q. Далее можно было бы вычислить произведение $(p - 1)(q - 1)$, а затем — модульное обращение $e \bmod (p - 1)(q - 1)$, получив тем самым ключ дешифрования d. После этого не составило бы труда вычислить M, т.е. исходный текст.

Есть лишь одна проблема. Вспомните, что n представляет собой число, для записи которого требуется около 600 цифр. Функция `math.sqrt()` не в состоянии обрабатывать столь большие числа, поэтому она выдаст сообщение об ошибке. Но даже если бы она поддерживала такие числа, цикл `for` выполнялся бы *очень* долго. Компьютеру не хватило бы и 5 миллиардов лет непрерывной работы, настолько велики эти числа!

Именно этим и объясняется стойкость шифров с открытым ключом: *с математической точки зрения никакого обходного пути для нахождения делителей больших чисел не существует.* Нет ничего проще, чем взять два простых числа p и q и перемножить их для получения числа n. В то же время крайне трудоемко решить обратную задачу: найти для достаточно большого числа n простые делители p и q. Если взять небольшое число, скажем, 15, то легко определить, что его можно разложить на множители 5 и 3. Но ситуация кардинальным образом изменится, если попытаться разложить на множители такое, например, число, как 178 565 887 643 607 245 654 502 737. Вот почему шифры с открытым ключом практически невозможно взломать.

Исходный код программы Public Key Cipher

Откройте в редакторе файлов новое окно, выбрав пункты меню File⇨ New File. Введите в этом окне приведенный ниже код и сохраните его в файле *publicKeyCipher.py*.

publicKeyCipher.py

```
 1. # Шифрование с открытым ключом
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import sys, math
 5.
 6. # Открытый и закрытый ключи для этой программы создаются
 7. # программой makePublicPrivateKeys.py. Программа должна
 8. # выполняться в той же папке, в которой находятся файлы ключей.
 9.
10. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
      1234567890 !?.'
11.
12. def main():
13.     # Проверяем, что необходимо сделать: записать зашифрованное
14.     # сообщение в файл или дешифровать сообщение, прочитанное из файла
15.     filename = 'encrypted_file.txt'  # файл для чтения/записи
16.     mode = 'encrypt'  # задать 'encrypt' или 'decrypt'
17.
18.     if mode == 'encrypt':
19.         message = 'Journalists belong in the gutter because that is
              where the ruling classes throw their guilty secrets. Gerald
              Priestland. The Founding Fathers gave the free press the
              protection it must have to bare the secrets of government
              and inform the people. Hugo Black.'
20.         pubKeyFilename = 'al_sweigart_pubkey.txt'
21.         print('Encrypting and writing to %s...' % (filename))
22.         encryptedText = encryptAndWriteToFile(filename,
              pubKeyFilename, message)
23.
24.         print('Encrypted text:')
25.         print(encryptedText)
26.
27.     elif mode == 'decrypt':
28.         privKeyFilename = 'al_sweigart_privkey.txt'
29.         print('Reading from %s and decrypting...' % (filename))
30.         decryptedText = readFromFileAndDecrypt(filename,
              privKeyFilename)
31.
32.         print('Decrypted text:')
33.         print(decryptedText)
34.
```

```
35.
36. def getBlocksFromText(message, blockSize):
37.     # Преобразует строку сообщения в список целочисленных блоков
38.     for character in message:
39.         if character not in SYMBOLS:
40.             print('ERROR: The symbol set does not have the
                      character %s' % (character))
41.             sys.exit()
42.     blockInts = []
43.     for blockStart in range(0, len(message), blockSize):
44.         # Вычисляем целочисленный блок для текущей группы символов
45.         blockInt = 0
46.         for i in range(blockStart, min(blockStart + blockSize,
              len(message))):
47.             blockInt += (SYMBOLS.index(message[i])) * (len(SYMBOLS) **
                  (i % blockSize))
48.         blockInts.append(blockInt)
49.     return blockInts
50.
51.
52. def getTextFromBlocks(blockInts, messageLength, blockSize):
53.     # Преобразует список целочисленных блоков в строку исходного
54.     # сообщения. Необходимо задать длину исходного сообщения, чтобы
55.     # корректно преобразовать последний блок.
56.     message = []
57.     for blockInt in blockInts:
58.         blockMessage = []
59.         for i in range(blockSize - 1, -1, -1):
60.             if len(message) + i < messageLength:
61.                 # Декодирование строки сообщения для нужного числа
62.                 # символов (задается переменной blockSize) из блока
63.                 charIndex = blockInt // (len(SYMBOLS) ** i)
64.                 blockInt = blockInt % (len(SYMBOLS) ** i)
65.                 blockMessage.insert(0, SYMBOLS[charIndex])
66.         message.extend(blockMessage)
67.     return ''.join(message)
68.
69.
70. def encryptMessage(message, key, blockSize):
71.     # Преобразует строку сообщения в список целочисленных блоков
72.     # и шифрует каждый блок. Для шифрования нужен ОТКРЫТЫЙ ключ.
73.     encryptedBlocks = []
74.     n, e = key
75.
76.     for block in getBlocksFromText(message, blockSize):
77.         # шифротекст = сообщение ^ e mod n
78.         encryptedBlocks.append(pow(block, e, n))
79.     return encryptedBlocks
80.
```

```
81.
82. def decryptMessage(encryptedBlocks, messageLength, key, blockSize):
83.     # Дешифрует список зашифрованных блоков в строку сообщения,
84.     # длину которого необходимо задать, чтобы корректно дешифровать
85.     # последний блок. Для дешифрования нужен ЗАКРЫТЫЙ ключ.
86.     decryptedBlocks = []
87.     n, d = key
88.     for block in encryptedBlocks:
89.         # сообщение = шифротекст ^ d mod n
90.         decryptedBlocks.append(pow(block, d, n))
91.     return getTextFromBlocks(decryptedBlocks, messageLength,
          blockSize)
92.
93.
94. def readKeyFile(keyFilename):
95.     # Считывает из указанного файла открытый или
96.     # закрытый ключ в виде кортежа (n,e) или (n,d)
97.     fo = open(keyFilename)
98.     content = fo.read()
99.     fo.close()
100.    keySize, n, EorD = content.split(',')
101.    return (int(keySize), int(n), int(EorD))
102.
103.
104. def encryptAndWriteToFile(messageFilename, keyFilename, message,
       blockSize=None):
105.    # Считывает ключ из файла, шифрует сообщение и сохраняет его
106.    # в выходном файле, возвращая зашифрованную строку
107.    keySize, n, e = readKeyFile(keyFilename)
108.    if blockSize == None:
109.        # Если параметр blockSize не задан, сделать его максимально
            допустимым для текущей длины ключа и размера набора символов
110.        blockSize = int(math.log(2 ** keySize, len(SYMBOLS)))
111.    # Проверяем, достаточно ли длины ключа для данного размера блока
112.    if not (math.log(2 ** keySize, len(SYMBOLS)) >= blockSize):
113.        sys.exit('ERROR: Block size is too large for the key and
              symbol set size. Did you specify the correct key file
              and encrypted file?')
114.    # Шифруем сообщение
115.    encryptedBlocks = encryptMessage(message, (n, e), blockSize)
116.
117.    # Преобразование больших целочисленных значений в единую строку
118.    for i in range(len(encryptedBlocks)):
119.        encryptedBlocks[i] = str(encryptedBlocks[i])
120.    encryptedContent = ','.join(encryptedBlocks)
121.
122.    # Запись зашифрованной строки в выходной файл
123.    encryptedContent = '%s_%s_%s' % (len(message), blockSize,
         encryptedContent)
```

```
124.     fo = open(messageFilename, 'w')
125.     fo.write(encryptedContent)
126.     fo.close()
127.     # Возвращаем зашифрованную строку
128.     return encryptedContent
129.
130.
131. def readFromFileAndDecrypt(messageFilename, keyFilename):
132.     # Считывает ключ из файла, считывает зашифрованное сообщение из
133.     # входного файла и дешифрует его, возвращая дешифрованную строку
134.     keySize, n, d = readKeyFile(keyFilename)
135.
136.
137.     # Считываем размер сообщения и зашифрованное сообщение из файла
138.     fo = open(messageFilename)
139.     content = fo.read()
140.     messageLength, blockSize, encryptedMessage = content.split('_')
141.     messageLength = int(messageLength)
142.     blockSize = int(blockSize)
143.
144.     # Проверяем, достаточно ли длины ключа для данного размера блока
145.     if not (math.log(2 ** keySize, len(SYMBOLS)) >= blockSize):
146.         sys.exit('ERROR: Block size is too large for the key and
                symbol set size. Did you specify the correct key file
                and encrypted file?')
147.
148.     # Преобразуем зашифрованное сообщение в целочисленные блоки
149.     encryptedBlocks = []
150.     for block in encryptedMessage.split(','):
151.         encryptedBlocks.append(int(block))
152.
153.     # Дешифруем большие целочисленные блоки
154.     return decryptMessage(encryptedBlocks, messageLength, (n, d),
          blockSize)
155.
156.
157. # Если файл publicKeyCipher.py выполняется как программа
158. # (а не импортируется как модуль), вызвать функцию main()
159. if __name__ == '__main__':
160.     main()
```

Пример выполнения программы `Public Key Cipher`

Давайте попробуем выполнить программу *publicKeyCipher.py* и зашифровать секретное сообщение. Чтобы отправить нужному человеку секретное сообщение, зашифрованное с помощью этой программы, получите файл открытого ключа данного человека и поместите его в одну папку с файлом программы.

Прежде чем приступить к шифрованию сообщения, убедитесь в том, что в строке 16 для переменной `mode` установлено значение `'encrypt'`. Обновите переменную `message` в строке 19, задав в ней текст сообщения. Укажите для переменной `pubKeyFilename` в строке 20 имя файла открытого ключа. В переменной `filename` (строка 15) хранится имя выходного файла, в который должен быть записан шифротекст. В строке 22 переменные `filename`, `pubKeyFilename` и `message` передаются функции `encryptAndWriteToFile()`, которая шифрует сообщение и сохраняет его в файле.

Запустив программу, вы должны получить примерно следующие результаты.

```
Encrypting and writing to encrypted_file.txt...
Encrypted text:
258_169_45108451524907138236859816039483721219475907590237903918239237768
64369948566603013231572537249780228617020983244277382842255301862133801888
88057732963483392298908904649695569377970724343149165228396922770345794633
59471384355989841893072346500886898507443612627071299717824076104502080479
92712968784162173477696570182779184902972157857592572908558122210889070169
90498302554217447160649477967360153100891558762342778833813452473536806245
58562967293970955701610727546938828451241925684094837372334973040879696240
04351615822168945414809602073875465635714074772465708958607695479122809499
85856627850647512542354899687383467956491253384333697511553976133225040269
69986883515062301758243811684004923608357374181764593337194564531336584762
27117603524859702197231645452654506945283876638759983934054240668777211351
51131345425258973397196221901606661497839037861117596445677366986094295451
60590171433908254272501514053098568511723259877817654563814140365701043538
59244660091910391099621028192177415196156469972977305212676293746827002938
83231466824069323003214109731255640062996151863579947865219607231642491864
48787555631633942494897580466092361668276724294829630167831204182893447378
86824809308122356133539825048880814063389057192492939651199537310635280371
```

Программа записывает шифротекст в файл *encrypted_file.txt*. Это результат шифрования строки, хранящейся в переменной `message` (строка 19). Поскольку используемый вами открытый ключ наверняка будет отличаться от моего, ваши результаты будут другими, но формат выходной строки останется тем же. Как нетрудно увидеть, шифротекст разбит на два блока, т.е. два больших целых числа, разделенных запятой. Число `258` в начале шифротекста соответствует длине исходного сообщения. Далее идет символ подчеркивания и число `169`, определяющее размер блока.

Чтобы дешифровать это сообщение, задайте для переменной `mode` значение `'decrypt'` и вновь запустите программу. Убедитесь в том, что в строке 28 в качестве значения переменной `privKeyFilename` указано имя соответствующего файла закрытого ключа и что этот файл, равно как и файл

encrypted_file.txt, находится в том же каталоге, что и файл *publicKeyCipher.py*. На этот раз программа дешифрует зашифрованное сообщение, хранящееся в файле *encrypted_file.txt*, и результат будет примерно таким.

```
Reading from encrypted_file.txt and decrypting...
Decrypted text:
Journalists belong in the gutter because that is where the ruling classes
throw their guilty secrets. Gerald Priestland. The Founding Fathers gave
the free press the protection it must have to bare the secrets of
government and inform the people. Hugo Black.
```

Учтите, что программа *publicKeyCipher.py* способна шифровать и дешифровать только простые текстовые файлы.

Теперь перейдем к более тщательному рассмотрению исходного кода программы.

Начало программы

В шифрах с открытым ключом используются числа, поэтому мы преобразуем строку сообщения в целое число. Оно вычисляется на основании индексов символов в наборе, который хранится в константе `SYMBOLS` (строка 10).

```
 1. # Шифрование с открытым ключом
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import sys, math
 5.
 6. # Открытый и закрытый ключи для этой программы создаются
 7. # программой makePublicPrivateKeys.py. Программа должна
 8. # выполняться в той же папке, в которой находятся файлы ключей.
 9.
10. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
      1234567890 !?.'
```

Как программа определяет, что ей делать: шифровать или дешифровать

Чтобы программа *publicKeyCipher.py* могла определить, какую именно операцию — шифрование или дешифрование — следует выполнить и какой файл ключа следует использовать, мы сохраняем определенные значения в соответствующих переменных, которые задаются в функции `main()`.

```
12. def main():
13.     # Проверяем, что необходимо сделать: записать зашифрованное
14.     # сообщение в файл или дешифровать сообщение, прочитанное из файла
15.     filename = 'encrypted_file.txt'  # файл для чтения/записи
16.     mode = 'encrypt'  # задать 'encrypt' или 'decrypt'
```

Если в строке 16 для переменной `mode` задано значение `'encrypt'`, то программа зашифрует сообщение и запишет его в файл, имя которого содержится в переменной `filename`. Если же переменная `mode` равна `'decrypt'`, то программа прочитает содержимое зашифрованного файла (его имя берется из переменной `filename`) и дешифрует его.

Строки 18–25 определяют порядок действий в случае шифрования.

```
18.     if mode == 'encrypt':
19.         message = 'Journalists belong in the gutter because that is
              where the ruling classes throw their guilty secrets. Gerald
              Priestland. The Founding Fathers gave the free press the
              protection it must have to bare the secrets of government
              and inform the people. Hugo Black.'
20.         pubKeyFilename = 'al_sweigart_pubkey.txt'
21.         print('Encrypting and writing to %s...' % (filename))
22.         encryptedText = encryptAndWriteToFile(filename,
              pubKeyFilename, message)
23.
24.         print('Encrypted text:')
25.         print(encryptedText)
```

Переменная `message` (строка 19) содержит текст, подлежащий шифрованию, а переменная `pubKeyFilename` (строка 20) задает имя файла открытого ключа (в данном примере это файл *al_sweigart_pubkey.txt*). Не забывайте о том, что переменная `message` может содержать лишь символы, которые входят в состав используемого символьного набора, хранящегося в переменной `SYMBOLS`. В строке 22 вызывается функция `encryptAndWriteToFile()`, которая шифрует сообщение с помощью открытого ключа и записывает его в файл, задаваемый переменной `filename`.

Строки 27–33 определяют порядок действий, когда переменная `mode` содержит строку `'decrypt'`. В этом случае вместо того, чтобы шифровать сообщение, программа считывает содержимое файла закрытого ключа, имя которого задано в переменной `privKeyFilename` (строка 28).

```
27.     elif mode == 'decrypt':
28.         privKeyFilename = 'al_sweigart_privkey.txt'
29.         print('Reading from %s and decrypting...' % (filename))
```

```
30.         decryptedText = readFromFileAndDecrypt(filename,
              privKeyFilename)
31.
32.         print('Decrypted text:')
33.         print(decryptedText)
```

Далее мы передаем переменные `filename` и `privKeyFilename` функции `readFromFileAndDecrypt()`, которая возвращает дешифрованное сообщение, сохраняемое в переменной `decryptedText` (строка 30). В строке 33 дешифрованное сообщение выводится на экран, и на этом работа программы завершается.

Теперь рассмотрим, как реализуются основные этапы шифрования с открытым ключом и, в частности, как сообщения преобразуются в блоки.

Преобразование строк в блоки с помощью функции `getBlocksFromText()`

Функция `getBlocksFromText()` преобразует сообщение в 128-байтовые блоки. Она получает строку сообщения и размер блока в качестве параметров и возвращает список блоков, т.е. список больших целых чисел, замещающих сообщение.

```
36. def getBlocksFromText(message, blockSize):
37.     # Преобразует строку сообщения в список целочисленных блоков
38.     for character in message:
39.         if character not in SYMBOLS:
40.             print('ERROR: The symbol set does not have the
                       character %s' % (character))
41.             sys.exit()
```

Код в строках 38–41 позволяет убедиться в том, что строка `message` содержит лишь символы из набора, хранящегося в переменной `SYMBOLS`. Параметр `blockSize` необязательный, и в нем может быть указан любой размер блока. Приступая к созданию блоков, мы прежде всего преобразуем строку в байты.

Блок создается путем объединения всех индексов символьного набора, соответствующих символам сообщения, в одно большое число, как это делалось в разделе “Преобразование строки в блок”. Для хранения блоков создается список `blockInts` (строка 42).

```
42.     blockInts = []
```

Мы хотим, чтобы размер каждого блока составлял `blockSize` байт, но если сообщение не делится равномерно на блоки такого размера, то количество символов в последнем блоке будет меньше, чем `blockSize`. Для обработки подобных ситуаций мы используем функцию `min()`.

Функции `min()` и `max()`

Функция `min()` возвращает наименьший из своих аргументов. Чтобы увидеть, как она работает, введите в интерактивной оболочке следующий код.

```
>>> min(13, 32, 13, 15, 17, 39)
13
```

Аргументом функции `min()` также может быть список или кортеж. Чтобы убедиться в этом, введите в интерактивной оболочке следующие инструкции.

```
>>> min([31, 26, 20, 13, 12, 36])
12
>>> spam = (10, 37, 37, 43, 3)
>>> min(spam)
3
```

В данном случае функция `min()` возвращает наименьший из элементов списка или кортежа. Противоположная ей функция `max()` возвращает наибольший из своих аргументов.

```
>>> max(18, 15, 22, 30, 31, 34)
34
```

Теперь вернемся к программе и рассмотрим, как с помощью функции `min()` последний блок усекается до нужного размера.

Сохранение блоков в переменной `blockInt`

В цикле `for`, который начинается в строке 43, создаются целые числа для каждого блока. В качестве значения переменной `blockStart` устанавливается индекс первого символа создаваемого блока.

```
43.     for blockStart in range(0, len(message), blockSize):
44.         # Вычисляем целочисленный блок для текущей группы символов
45.         blockInt = 0
46.         for i in range(blockStart, min(blockStart + blockSize,
              len(message))):
```

Для хранения создаваемого блока предназначена переменная `blockInt`, первоначально равная `0` (строка 45). Во вложенном цикле `for`, который начинается в строке 46, переменная `i` последовательно становится равной индексам всех символов сообщения, включаемых в блок. Индексы стартуют от значения `blockStart` и достигают значения `blockStart + blockSize` *либо* `len(message)`, в зависимости от того, что меньше. Функция `min()` в строке 46 возвращает наименьшее из двух значений.

Второй аргумент функции `range()` в строке 46 должен быть наименьшим из значений `blockStart + blockSize` и `len(message)`, поскольку каждый блок всегда состоит из 128 символов (или другого фиксированного значения, задаваемого переменной `blockSize`), *за исключением* последнего блока. Последний блок может содержать ровно 128 символов, но гораздо вероятнее, что их будет меньше. В таком случае мы хотим ограничить переменную цикла `i` значением `len(message)`, поскольку оно соответствует индексу последнего символа в сообщении.

Определив диапазон символов, составляющих блок, мы выполняем арифметическую операцию для преобразования символов в одно большое число. Вспомните, как в разделе "Преобразование строки в блок" мы создавали большое целое число путем умножения индекса каждого символа в символьном наборе на $66^{индекс\_символа}$ (66 — это длина строки `SYMBOLS`). В программе мы умножаем значение `SYMBOLS.index(message[i])` (целочисленный индекс символа в наборе) на выражение `(len(SYMBOLS) ** (i % blockSize))` для каждого символа, и прибавляем результат к значению `blockInt`.

```
47.             blockInt += (SYMBOLS.index(message[i])) * (len(SYMBOLS) **
                  (i % blockSize))
```

Мы хотим, чтобы показателем степени служил индекс относительно *начала текущего блока*, который всегда будет изменяться от нуля до `blockSize`. Мы не можем напрямую использовать переменную `i` в качестве показателя степени *индекс_символа*, поскольку она индексирует всю строку сообщения, проходя значения от `0` до `len(message)`. В таком случае мы получили бы индекс, намного больший, чем `66`. Выполнив деление `i` по модулю `blockSize`, мы получаем индекс относительно начала блока; именно поэтому в строке 47 выполняется операция `len(SYMBOLS) ** (i % blockSize)`, а не просто `len(SYMBOLS) ** i`.

По завершении вложенного цикла мы получаем целочисленное значение блока. В строке 48 это значение добавляется в список `blockInts`. На следующей итерации основного цикла, начатого в строке 43, вычисляется следующий блок сообщения.

```
48.         blockInts.append(blockInt)
49.     return blockInts
```

В итоге мы получаем список целочисленных значений блоков, `blockInts`, который возвращается функцией в строке 49.

На данном этапе мы преобразовали всю строку сообщения в целочисленные блоки, но необходимо понять, как выполнить обратное преобразование целочисленных блоков в исходное сообщение в процессе дешифрования.

Дешифрование блоков с помощью функции `getTextFromBlocks()`

Функция `getTextFromBlocks()` решает задачу, противоположную той, которую решает функция `getBlocksFromText()`. Она имеет параметр `blockInts`, содержащий список целочисленных блоков, параметр `messageLength`, задающий длину сообщения, и параметр `blockSize`, определяющий размер блока, и возвращает строку, соответствующую данному списку блоков. Параметр `messageLength` нужен для того, чтобы функция могла обработать последний блок в тех случаях, когда он содержит менее `blockSize` символов. Этот процесс был описан в разделе "Преобразование блока в строку".

```
52. def getTextFromBlocks(blockInts, messageLength, blockSize):
53.     # Преобразует список целочисленных блоков в строку исходного
54.     # сообщения. Необходимо задать длину исходного сообщения, чтобы
55.     # корректно преобразовать последний блок.
56.     message = []
```

Список `message`, создаваемый в строке 56, будет хранить все символы, получаемые в результате обработки списка целочисленных блоков `blockInts`.

В цикле `for`, который начинается в строке 57, мы проходим по всем целочисленным блокам, хранящимся в списке `blockInts`. В строках 58–65 вычисляются буквы, закодированные в текущем блоке.

```
57.     for blockInt in blockInts:
58.         blockMessage = []
59.         for i in range(blockSize - 1, -1, -1):
```

Функция `getTextFromBlocks()` разбивает блок на `blockSize` целых чисел, каждое из которых соответствует индексу символа в символьном

наборе. Мы должны извлекать эти индексы из переменной `blockInt` в обратном порядке, поскольку в процессе шифрования сообщения мы начинали с наименьших показателей степени (66^0, 66^1, 66^2 и т.д.), тогда как в процессе дешифрования операции деления и деления с остатком должны выполняться сначала для наибольших показателей. Вот почему переменная вложенного цикла `for`, начинающегося в строке 59, имеет начальное значение `blockSize - 1`, а затем уменьшается на 1 на каждой итерации, пока не станет равна 0.

Прежде чем преобразовать индекс в символ, необходимо убедиться в том, что в процессе декодирования блоков мы не вышли за пределы сообщения. Это делается путем проверки того, что количество преобразованных на данный момент символов, т.е. `len(message) + i`, все еще не превышает значение `messageLength` (строка 60).

```
60.             if len(message) + i < messageLength:
61.                 # Декодирование строки сообщения для нужного числа
62.                 # символов (задается переменной blockSize) из блока
63.                 charIndex = blockInt // (len(SYMBOLS) ** i)
64.                 blockInt = blockInt % (len(SYMBOLS) ** i)
```

Для извлечения символов из блока мы следуем процедуре, описанной в разделе "Преобразование блока в строку". Каждый символ добавляется в список `blockMessage`. В процессе шифрования блоков порядок следования символов фактически меняется на противоположный, поэтому декодированный символ нельзя присоединить в конец списка `blockMessage`. Вместо этого символ вставляется в начало списка с помощью метода `insert()`.

Использование спискового метода `insert()`

Списковый метод `append()` добавляет значение в конец списка, тогда как метод `insert()` позволяет вставить значение в *любую* позицию списка. В качестве аргументов метод `insert()` получает целочисленный индекс позиции вставки и добавляемое значение. Чтобы увидеть, как это работает, введите в интерактивной оболочке следующие инструкции.

```
>>> spam = [2, 4, 6, 8]
>>> spam.insert(0, 'hello')
>>> spam
['hello', 2, 4, 6, 8]
>>> spam.insert(2, 'world')
>>> spam
['hello', 2, 'world', 4, 6, 8]
```

В этом примере мы создаем список spam, а затем вставляем в него строку 'hello', делая ее первым элементом списка (индекс 0). Как видите, элемент можно вставить в позицию с любым допустимым индексом, в том числе 2.

Объединение элементов списка message в одну строку

Мы используем вычисленное значение charIndex в качестве индекса в строке SYMBOLS, чтобы извлечь из нее нужный символ, который добавляется в начало списка blockMessage.

```
65.                 blockMessage.insert(0, SYMBOLS[charIndex])
66.         message.extend(blockMessage)
67.     return ''.join(message)
```

В конце каждой итерации основного цикла все символы текущего блока добавляются в список message (строка 66). По завершении цикла функция getTextFromBlocks() возвращает список, преобразованный в строку.

Функция encryptMessage()

Функция encryptMessage() шифрует каждый блок, используя строку исходного сообщения, а также кортеж открытого ключа из двух целых чисел, создаваемый функцией readKeyFile(), которую мы напишем далее. Функция возвращает список зашифрованных блоков.

```
70. def encryptMessage(message, key, blockSize):
71.     # Преобразует строку сообщения в список целочисленных блоков
72.     # и шифрует каждый блок. Для шифрования нужен ОТКРЫТЫЙ ключ.
73.     encryptedBlocks = []
74.     n, e = key
```

В строке 73 создается список encryptedBlocks, в котором будут храниться зашифрованные блоки. В строке 74 два целых числа, хранящихся в кортеже key, присваиваются переменным n и e. Далее реализуется арифметика шифрования с открытым ключом.

Над каждым блоком исходного сообщения выполняются определенные арифметические операции, в результате которых мы получаем целое число, представляющее собой зашифрованный блок. В частности, мы возводим целочисленное значение блока в степень e и выполняем деление по модулю n, применяя функцию pow(block, e, n) (строка 78).

```
76.     for block in getBlocksFromText(message, blockSize):
77.         # шифротекст = сообщение ^ e mod n
78.         encryptedBlocks.append(pow(block, e, n))
79.     return encryptedBlocks
```

Полученный зашифрованный блок добавляется в список `encrypted-Blocks`.

Функция `decryptMessage()`

Функция `decryptMessage()` дешифрует список блоков и возвращает дешифрованную строку сообщения. В качестве аргументов ей передается список зашифрованных блоков, длина сообщения, кортеж закрытого ключа и размер блока.

В строке 86 создается список `decryptedBlocks`, предназначенный для хранения дешифрованных блоков, а в строке 87 мы применяем трюк с групповым присваиванием для записи двух целочисленных составляющих кортежа ключа в переменные `n` и `d`.

```
82. def decryptMessage(encryptedBlocks, messageLength, key, blockSize):
83.     # Дешифрует список зашифрованных блоков в строку сообщения,
84.     # длину которого необходимо задать, чтобы корректно дешифровать
85.     # последний блок. Для дешифрования нужен ЗАКРЫТЫЙ ключ.
86.     decryptedBlocks = []
87.     n, d = key
```

Арифметика дешифрования блоков та же, что и в случае шифрования, за исключением того, что целочисленное значение блока возводится не в степень `e`, а в степень `d` (строка 90).

```
88.     for block in encryptedBlocks:
89.         # сообщение = шифротекст ^ d mod n
90.         decryptedBlocks.append(pow(block, d, n))
```

Список дешифрованных блоков вместе с параметрами `messageLength` и `blockSize` передается функции `getTextFromBlocks()`, и результат возвращается в виде дешифрованного текста.

```
91.     return getTextFromBlocks(decryptedBlocks, messageLength,
          blockSize)
```

Теперь следует рассмотреть, каким образом функция `readKeyFile()` считывает открытый и закрытый ключи из файлов и создает кортежи, которые передаются функциям `encryptMessage()` и `decryptMessage()`.

Чтение открытого и закрытого ключей из соответствующих файлов

Функция `readKeyFile()` считывает числовые значения ключей из файлов, создаваемых программой *makePublicPrivateKeys.py* (см. главу 23). Имя файла задается параметром `keyFilename`. Файл должен находиться в той же папке, что и программа *publicKeyCipher.py*.

В строках 97–99 функция открывает файл, загружает его содержимое в строковом виде в переменную `content`, после чего закрывает файл.

```
 94. def readKeyFile(keyFilename):
 95.     # Считывает из указанного файла открытый или
 96.     # закрытый ключ в виде кортежа (n,e) или (n,d)
 97.     fo = open(keyFilename)
 98.     content = fo.read()
 99.     fo.close()
100.     keySize, n, EorD = content.split(',')
101.     return (int(keySize), int(n), int(EorD))
```

В файле ключа хранится размер ключа в байтах, число `n`, а также число `e` либо `d`, в зависимости от типа ключа (открытый или закрытый). Как было показано в предыдущей главе, эти значения записываются в строковом виде и разделяются запятыми, что позволяет извлечь их с помощью строкового метода `split()`, задав запятую в качестве разделителя. Возвращаемый методом `split()` список содержит три элемента, которые помещаются в переменные `keySize`, `n` и `EorD` с помощью инструкции группового присваивания (строка 100).

Вспомните, что переменная `content` представляет собой строку, поэтому возвращаемые методом `split()` элементы списка также будут строками. Чтобы преобразовать их в числа, мы применяем к ним функцию `int()`. В результате функция `readKeyFile()` возвращает три целых числа: `int(keySize)`, `int(n)` и `int(EorD)`.

Запись зашифрованного сообщения в файл

Функция `encryptAndWriteToFile()` вызывает функцию `encryptMessage()` для шифрования строки с помощью прочитанного ключа и создает файл, содержащий зашифрованное содержимое.

```
104. def encryptAndWriteToFile(messageFilename, keyFilename, message,
       blockSize=None):
105.     # Считывает ключ из файла, шифрует сообщение и сохраняет его
106.     # в выходном файле, возвращая зашифрованную строку
107.     keySize, n, e = readKeyFile(keyFilename)
```

Функция имеет три строковых параметра: имя файла, в который необходимо записать зашифрованное сообщение (`messageFilename`), имя файла открытого ключа (`keyFilename`) и сообщение, подлежащее шифрованию (`message`). Необязательный четвертый параметр `blockSize` задает размер блока.

Первый шаг процесса шифрования — чтение значений `keySize`, `n` и `e` из файла ключа с помощью функции `readKeyFile()` (строка 107).

Для параметра `blockSize` предусмотрено значение по умолчанию `None`.

```
108.     if blockSize == None:
109.         # Если параметр blockSize не задан, сделать его максимально
               допустимым для текущей длины ключа и размера набора символов
110.         blockSize = int(math.log(2 ** keySize, len(SYMBOLS)))
```

Если параметр `blockSize` не задан, то размер блока выбирается максимально допустимым для заданной длины ключа и используемого набора символов. Не забывайте о том, что должно выполняться условие $2^{длина\_ключа} > размер\_набора\_символов^{размер\_блока}$. Поэтому для определения предельного размера блока вызывается функция `math.log()`, которая вычисляет логарифм значения $2^{длина\_ключа}$ по основанию `len(SYMBOLS)` (строка 110).

Арифметика шифрования с открытым ключом корректно работает лишь в том случае, если длина ключа равна или превышает размер блока, поэтому выполняемая в строке 112 проверка играет важную роль.

```
111.     # Проверяем, достаточно ли длины ключа для данного размера блока
112.     if not (math.log(2 ** keySize, len(SYMBOLS)) >= blockSize):
113.         sys.exit('ERROR: Block size is too large for the key and
                 symbol set size. Did you specify the correct key file
                 and encrypted file?')
```

Если длина ключа слишком мала, выводится сообщение об ошибке и программа завершает работу. В этом случае пользователь должен либо уменьшить значение параметра `blockSize`, либо использовать ключ большей длины.

Теперь, когда у нас имеются компоненты ключа `n` и `e`, можно вызвать функцию `encryptMessage()`, которая возвращает список целочисленных блоков.

```
114.     # Шифруем сообщение
115.     encryptedBlocks = encryptMessage(message, (n, e), blockSize)
```

Функция `encryptMessage()` ожидает, что ключ будет передан ей в виде кортежа из двух целых чисел. Именно поэтому переменные `n` и `e` помещаются в кортеж, который передается в качестве второго аргумента.

Далее мы преобразуем зашифрованные блоки в строку, которую можно записать в файл. Все блоки объединяются в одну строку и разделяются запятыми. Использовать для этого инструкцию `','.join(encryptedBlocks)` напрямую нельзя, поскольку метод `join()` применим только к спискам со строковыми значениями. На данный момент список `encryptedBlock` содержит целые числа, следовательно, их необходимо преобразовать в строки.

```
117.     # Преобразование больших целочисленных значений в единую строку
118.     for i in range(len(encryptedBlocks)):
119.         encryptedBlocks[i] = str(encryptedBlocks[i])
120.     encryptedContent = ','.join(encryptedBlocks)
```

В цикле `for`, который начинается в строке 118, мы проходим по всем индексам списка `encryptedBlocks`, преобразуя целое число `encryptedBlocks[i]` в строку. По завершении цикла список `encryptedBlocks` будет содержать строки, а не целые числа.

Только после этого список строк, содержащихся в переменной `encryptedBlocks`, можно передать методу `join()`, который объединяет их в строку, используя запятые в качестве разделителей. Объединенная строка сохраняется в переменной `encryptedContent` (строка 120).

Кроме зашифрованных целочисленных блоков, в файл записываются также длина сообщения и размер блока.

```
122.     # Запись зашифрованной строки в выходной файл
123.     encryptedContent = '%s_%s_%s' % (len(message), blockSize,
           encryptedContent)
```

В строке 123 содержимое переменной `encryptedContent` преобразуется таким образом, чтобы она включала размер сообщения в виде целого числа (`len(message)`), за которым следуют символ подчеркивания, размер блока (`blockSize`), еще один символ подчеркивания и строка зашифрованных блоков (`encryptedContent`).

Последним этапом процесса шифрования становится запись содержимого в файл. В строке 124 вызывается функция `open()`, которой передается имя файла, хранящееся в параметре `messageFilename`. Учтите, что если файл с таким именем уже существует, то он будет перезаписан.

```
124.     fo = open(messageFilename, 'w')
125.     fo.write(encryptedContent)
126.     fo.close()
127.     # Возвращаем зашифрованную строку
128.     return encryptedContent
```

Строка, содержащаяся в переменной encryptedContent, записывается в файл посредством вызова метода write() в строке 125. Далее файловый объект fo закрывается (строка 126).

Наконец, содержимое переменной encryptedContent возвращается функцией encryptAndWriteToFile() в строке 128. Это делается для того, чтобы программа, вызвавшая данную функцию, могла, к примеру, вывести строку на экран.

Теперь, когда вы узнали, как функция encryptAndWriteToFile() шифрует строку сообщения и записывает результаты в файл, мы можем перейти к рассмотрению того, как функция readFromFileAndDecrypt() дешифрует зашифрованное сообщение.

Дешифрование содержимого файла

Как и функция encryptAndWriteToFile(), функция readFromFileAndDecrypt() имеет параметры, задающие имя файла зашифрованного сообщения и имя файла ключа. Убедитесь в том, что функции передается имя файла закрытого, а не открытого ключа.

```
131. def readFromFileAndDecrypt(messageFilename, keyFilename):
132.     # Считывает ключ из файла, считывает зашифрованное сообщение из
133.     # входного файла и дешифрует его, возвращая дешифрованную строку
134.     keySize, n, d = readKeyFile(keyFilename)
```

Первым этапом процедуры дешифрования становится вызов функции readKeyFile() для считывания значений переменных keySize, n и d из файла ключа (аналогично тому, как это происходило в процессе шифрования сообщения).

Второй этап — чтение содержимого файла. В строке 138 файл, заданный в параметре messageFilename, открывается для чтения.

```
137.     # Считываем размер сообщения и зашифрованное сообщение из файла
138.     fo = open(messageFilename)
139.     content = fo.read()
140.     messageLength, blockSize, encryptedMessage = content.split('_')
141.     messageLength = int(messageLength)
142.     blockSize = int(blockSize)
```

Метод read() (строка 139) возвращает содержимое файла в виде одной строки, которую вы увидели бы, если бы открыли файл в текстовом редакторе, таком как Блокнот или TextEdit, скопировали все его содержимое и вставили в программу.

Вспомните, что формат зашифрованного файла — это три целых числа, разделенных символами подчеркивания. Они задают длину сообщения, размер блока и собственно зашифрованные блоки. В строке 140 вызывается метод `split()`, возвращающий указанные три значения в виде списка. С помощью операции группового присваивания эти значения сохраняются в переменных `messageLength`, `blockSize` и `encryptedMessage` соответственно.

Поскольку метод `split()` возвращает строки, содержимое переменных `messageLength` и `blockSize` преобразуется в целые числа с помощью функции `int()` (строки 141 и 142).

В функции также проверяется, чтобы размер блока не превышал длину ключа (строка 145).

```
144.     # Проверяем, достаточно ли длины ключа для данного размера блока
145.     if not (math.log(2 ** keySize, len(SYMBOLS)) >= blockSize):
146.         sys.exit('ERROR: Block size is too large for the key and
               symbol set size. Did you specify the correct key file
               and encrypted file? ')
```

Чаще всего это формальная проверка, ведь если бы размер блока был слишком большим, то зашифрованный файл вообще не был бы создан. Наиболее вероятной причиной ошибки может стать неверно заданный файл закрытого ключа в параметре `keyFilename`, а это означает, что сообщение в любом случае было бы дешифровано некорректно.

Строка `encryptedMessage` содержит несколько блоков, разделенных запятыми, которые мы преобразуем обратно в целые числа и сохраняем в списке `encryptedBlocks`.

```
148.     # Преобразуем зашифрованное сообщение в целочисленные блоки
149.     encryptedBlocks = []
150.     for block in encryptedMessage.split(','):
151.         encryptedBlocks.append(int(block))
```

В цикле `for`, который начинается в строке 150, мы проходим по списку блоков, созданному путем вызова метода `split()` для строки `encryptedMessage`. Список содержит строковые значения блоков, которые преобразуются в целочисленный вид и добавляются в список `encryptedBlock` (строка 151). По завершении цикла список `encryptedBlocks` будет содержать целочисленные блоки зашифрованного сообщения.

В строке 154 список `encryptedBlocks` передается функции `decryptMessage()` вместе с аргументом `messageLength` (длина сообщения), закры-

тым ключом (кортеж из двух целых чисел n и d) и аргументом blockSize (размер блока).

```
153.     # Дешифруем большие целочисленные блоки
154.     return decryptMessage(encryptedBlocks, messageLength, (n, d),
           blockSize)
```

Возвращаемая функцией decryptMessage() строка дешифрованного сообщения в свою очередь возвращается функцией readFileAndDecrypt().

Вызов функции main()

Наконец, если файл *publicKeyCipher.py* запускается как программа, а не импортируется в виде модуля другой программой, то в строках 159 и 160 вызывается функция main().

```
157. # Если файл publicKeyCipher.py выполняется как программа
158. # (а не импортируется как модуль), вызвать функцию main()
159. if __name__ == '__main__':
160.     main()
```

Резюме

Примите поздравления — вы дочитали книгу до конца! Главы под названием "Взлом шифров с открытым ключом" не будет ввиду отсутствия простых методик взлома, способных справиться с этой задачей в сроки, которые не исчислялись бы триллионами лет.

В данной главе рассматривался упрощенный вариант RSA, и тем не менее это реальный шифр, применяемый в профессиональных криптографических приложениях. Например, когда вы регистрируетесь на сайте или совершаете покупки в Интернете, подобные шифры позволяют защищать пароли и номера кредитных карт от злоумышленников, которые могут перехватывать ваш сетевой трафик.

Несмотря на то что арифметика шифрования с открытым ключом везде одна и та же, вам не следует использовать рассмотренную здесь программу для защиты своих секретных файлов. Способы взлома подобных программ достаточно сложные, и тем не менее они существуют. Например, из-за того что случайные числа, генерируемые функцией random.randint(), не являются истинно случайными и могут быть предсказаны, злоумышленник способен выяснить, какие числа были выбраны в качестве простых чисел для вашего закрытого ключа.

Все шифры, описанные в предыдущих главах, уязвимы для взлома и не предназначены для практического применения. Вообще говоря, не стоит пытаться писать собственный криптографический код для защиты своих конфиденциальных данных, поскольку весьма вероятно, что в процессе реализации программы вы допустите ряд трудно обнаруживаемых ошибок. В результате злоумышленники и спецслужбы смогут воспользоваться этими уязвимостями для взлома ваших зашифрованных сообщений.

Шифр надежен лишь в том случае, если сообщение удается сохранить в тайне, когда о шифре известно все, кроме вашего личного ключа. Нельзя рассчитывать на то, что криптоаналитик не имеет доступа к тому же криптографическому приложению, которым пользуетесь вы, или что ему неизвестно, какой шифр вы применили. Всегда исходите из того, что враг знает систему!

Профессиональные программы шифрования пишутся специалистами в области криптографии, которые годами изучали математические принципы и слабые места различных шифров. Но даже в этом случае написанные ими программы анализируются другими криптографами, которые пытаются найти в них ошибки или потенциальные уязвимости. Вам тоже вполне по силам изучить все, что необходимо знать о криптосистемах, включая соответствующий математический аппарат. Для этого не обязательно быть самым крутым хакером, нужно лишь потратить достаточно времени на овладение необходимым багажом знаний.

Надеюсь, книга послужит тем фундаментом, опираясь на который вы станете отличным программистом. О программировании и криптографии можно было бы рассказать гораздо больше, чем охвачено здесь, поэтому не останавливайтесь на полпути! Настоятельно рекомендую прочитать книгу Саймона Сингха "Книга шифров. Тайная история шифров и их расшифровки" (`https://pda.coollib.com/b/233545/read`), в которой увлекательно рассказывается об истории криптографии.

Желаю удачи!

A

ОТЛАДКА КОДА PYTHON

IDLE содержит встроенный отладчик, позволяющий выполнять программу пошагово, что делает его важным инструментом поиска ошибок. Запустив программу в режиме отладки, можно исследовать значения переменных в любой ее точке, получив тем самым ценную информацию о ее работе.

В этом приложении вы узнаете о том, как работает отладчик, а затем попрактикуетесь в отладке одной из программ и научитесь задавать точки останова.

Как работает отладчик

Чтобы запустить отладчик IDLE, выберите в окне интерактивной оболочки пункты меню Debug⇨Debugger (Отладка⇨Отладчик). На экране появится окно Debug Control (Управление отладкой), приведенное на рис. A.1.

Установите в этом окне флажки Stack (Стек), Locals (Локальные переменные), Source (Исходный код) и Globals (Глобальные переменные), чтобы отображать весь объем отладочной информации. Если окно Debug Control открыто, то всякий раз, когда вы будете запускать программу из окна редактора файлов, отладчик будет приостанавливать ее выполнение перед первой инструкцией и выводить следующую информацию:

- инструкция, которая должна быть выполнена;
- список всех локальных переменных и их значений;
- список всех глобальных переменных и их значений.

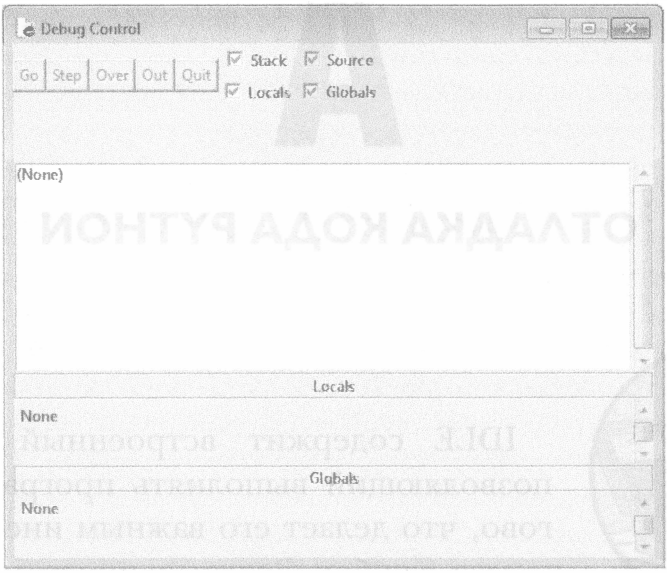


Рис. А.1. *Окно Debug Control*

Запустив программу в режиме отладки, вы заметите, что в списке глобальных переменных встречаются такие переменные, как `__builtins__`, `__doc__`, `__file__` и др., которые не были определены в программе. Их предназначение в книге не рассматривается, поэтому игнорируйте их.

Программа будет находиться в режиме ожидания до тех пор, пока вы не щелкнете на одной из следующих пяти кнопок в окне Debug Control: Go, Step, Over, Out или Quit. Рассмотрим назначение каждой из них.

Кнопка Go

Щелчок на кнопке Go (Выполнить) приводит к запуску программы в обычном режиме: она будет выполняться вплоть до своего завершения или до ближайшей *точки останова*. Последняя представляет собой строку, по достижении которой отладчик приостанавливает выполнение программы. Подробнее о точках останова мы поговорим далее. Если хотите, чтобы программа продолжила работу в обычном режиме после точки останова, вновь щелкните на кнопке Go.

Кнопка Step

Щелчок на кнопке Step (Шаг с заходом) приводит к выполнению следующей инструкции, после чего программа вновь приостанавливается. Если к этому моменту значения глобальных или локальных переменных успели измениться, то их списки, отображаемые в окне Debug Control, обновятся. Если очередной инструкцией является вызов функции, то отладчик перейдет в нее, т.е. вызовет функцию и остановится на первой строке ее кода. Это произойдет даже в том случае, когда функция содержится в отдельном модуле или является одной из встроенных функций Python.

Кнопка Over

Щелчок на кнопке Over (Шаг с обходом) приводит к выполнению следующей инструкции, как и при щелчке на кнопке Step. Но если следующей строкой кода является вызов функции, то отладчик не станет заходить в нее, а выполнит код функции целиком, как одну инструкцию, после чего приостановит работу программы. Например, если очередной инструкцией является вызов функции `print()`, то навряд ли вам интересно увидеть, как она реализована. Вам лишь нужно, чтобы переданная ей строка была выведена на экран. Поэтому обычно кнопкой Over пользуются чаще, чем кнопкой Step.

Кнопка Out

Щелчок на кнопке Out (Шаг с выходом) приводит к выполнению программы в обычном режиме до тех пор, пока не завершится текущая функция. Если перед этим вы выполнили шаг с заходом в функцию с помощью кнопки Step, а теперь просто хотите продолжить работу программы до конца функции, щелкните на кнопке Out, и отладчик дойдет до инструкции, следующей за вызовом функции.

Кнопка Quit

Чтобы прекратить отладку и завершить работу программы, щелкните на кнопке Quit (Выход). Если хотите возобновить работу программы в обычном режиме, отключите отладчик, повторно выбрав пункты меню Debug⇨Debugger.

Отладка программы Reverse Cipher

Давайте попробуем выполнить отладку одной из программ, рассмотренных в книге. Откройте программу *reverseCipher.py* (см. главу 4). Если вы не читали главу 4, то введите в окне редактора файлов приведенный ниже код и сохраните его в файле *reverseCipher.py*. Можете также загрузить этот файл на сайте издательства (см. введение).

reverseCipher.py

```
 1. # Программа обратного шифрования
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. message = 'Three can keep a secret, if two of them are dead.'
 5. translated = ''
 6.
 7. i = len(message) - 1
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Выберите в окне интерактивной оболочки пункты меню Debug⇨Debugger (Отладка⇨Отладчик). Когда вы нажмете клавишу <F5> или выберете пункт Run⇨Run Module (Выполнить⇨Выполнить модуль), отладчик запустит программу и остановится на строке 4. Строки 1–3 в данной программе либо пустые, либо содержат комментарии, поэтому отладчик автоматически пропускает их. Отладчик всегда останавливается на той строке кода, которая будет выполняться следующей. Окно Debug Control примет вид, показанный на рис. А.2.

Щелкните один раз на кнопке Over, чтобы выполнить строку 4, в результате чего переменной `message` будет присвоено значение `'Three can keep a secret, if two of them are dead.'`. Теперь в разделе Globals окна Debug Control должна появиться переменная `message` вместе со своим значением.

Далее отладчик переходит к строке 5, которая будет подсвечена в окне редактора файлов (рис. А.3).

Подсветка указывает на то, в каком именно месте программы находится в данный момент отладчик. Можете продолжить пошагово выполнять программу, щелкая на кнопке Over. Если же хотите возобновить ее работу в обычном режиме, щелкните на кнопке Go.

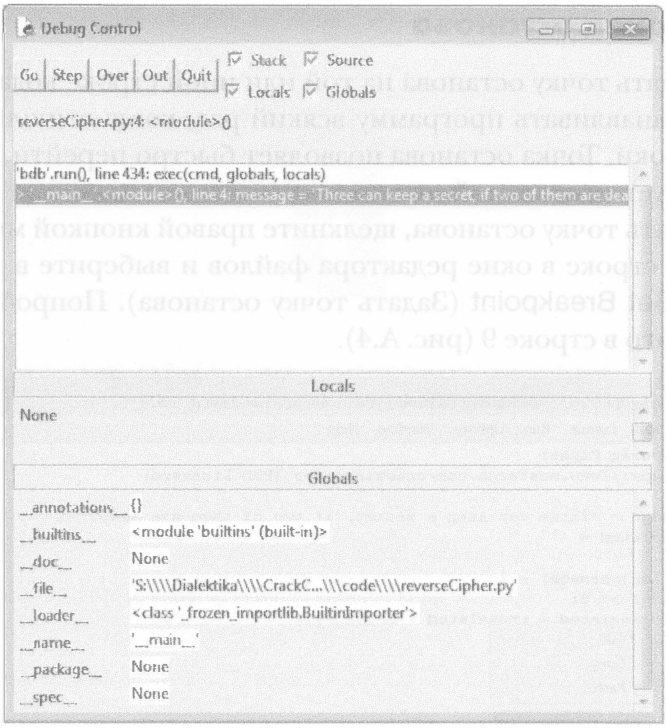


Рис. A.2. *Вид окна Debug Control при первом запуске программы под управлением отладчика*

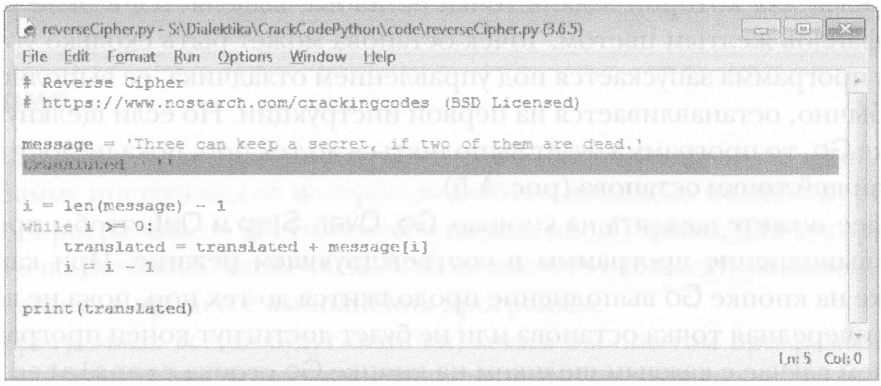


Рис. A.3. *Отладчик остановился на строке 5*

Всякий раз, когда отладчик делает паузу, это дает возможность просмотреть значения программных переменных и понять, как они меняются от инструкции к инструкции. Если в программе имеются ошибки или она делает не то, что от нее ожидают, анализ хода ее выполнения помогает выяснить, что же происходит на самом деле.

Задание точек останова

Можно задать точку останова на той или иной строке кода, и отладчик будет приостанавливать программу всякий раз, когда выполнение достигает этой строки. Точка останова позволяет быстро перейти к тому месту, которое требует тщательной отладки.

Чтобы задать точку останова, щелкните правой кнопкой мыши на соответствующей строке в окне редактора файлов и выберите в контекстном меню пункт Set Breakpoint (Задать точку останова). Попробуйте, например, сделать это в строке 9 (рис. А.4).

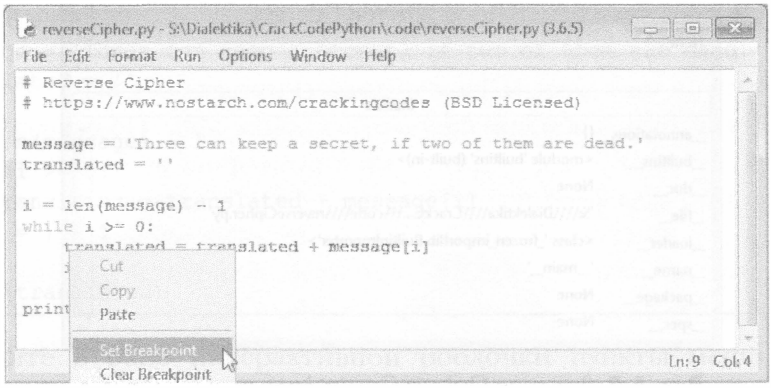


Рис. А.4. *Задание точки останова в строке 9*

Строка, для которой задана точка останова, выделяется в окне редактора файлов желтым цветом. Точек останова может быть сколько угодно. Когда программа запускается под управлением отладчика, ее выполнение, как обычно, останавливается на первой инструкции. Но если щелкнуть на кнопке Go, то программа будет выполняться до тех пор, пока не достигнет ближайшей точки останова (рис. А.5).

Далее можете щелкать на кнопках Go, Over, Step и Out, чтобы продолжить выполнение программы в соответствующем режиме. При каждом щелчке на кнопке Go выполнение продолжится до тех пор, пока не встретится очередная точка останова или не будет достигнут конец программы. В нашем случае с каждым щелчком на кнопке Go строка `translated`, отображаемая в разделе Globals, будет увеличиваться по мере шифрования сообщения.

Чтобы удалить точку останова, щелкните правой кнопкой мыши на соответствующей строке в окне редактора файлов и выберите в контекстном меню пункт Clear Breakpoint (Отменить точку останова). Желтая подсветка строки исчезнет, и в будущем отладчик не будет останавливаться на этой строке.

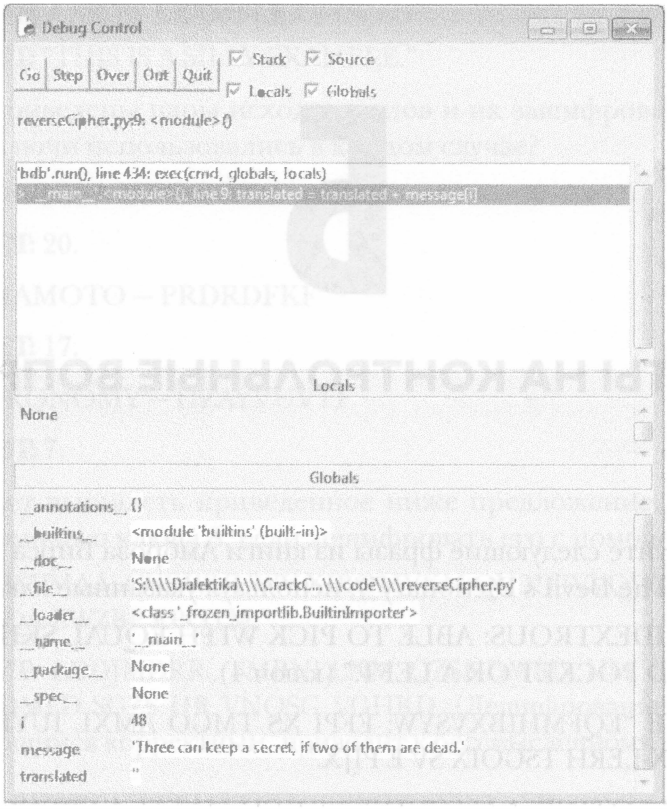


Рис. А.5. *Отладчик достиг точки останова в строке 9*

Резюме

Отладчик — полезный инструмент, позволяющий пошагово выполнять программу инструкция за инструкцией. С его помощью можно прерывать работу программы по достижении определенной строки, для которой задана точка останова. Также он дает возможность наблюдать за изменением переменных в процессе выполнения программы.

В программах всегда будут ошибки, каким бы опытом программирования вы ни обладали. Использование отладчика поможет разобраться в том, что именно происходит в программе, и облегчит выявление ошибок.

Б

ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

Глава 1

1. Зашифруйте следующие фразы из книги Амброза Бирса "Словарь Сатаны" ("The Devil's Dictionary"), используя указанные ключи.

   А. "AMBIDEXTROUS: ABLE TO PICK WITH EQUAL SKILL A RIGHT-HAND POCKET OR A LEFT." (ключ 4).

   ОТВЕТ: "EQFMHIBXVSYW: EFPI XS TMGO AMXL IUYEP WOMPP E VMKLX-LERH TSGOIX SV E PIJX."

   Б. "GUILLOTINE: A MACHINE WHICH MAKES A FRENCHMAN SHRUG HIS SHOULDERS WITH GOOD REASON." (ключ 17).

   ОТВЕТ: "XLZCCFKZEV: R DRTYZEV NYZTY DRBVJ R WIVETYDRE JYILX YZJ JYFLCUVIJ NZKY XFFU IVRJFE."

   В. "IMPIETY: YOUR IRREVERENCE TOWARD MY DEITY." (ключ 21).

   ОТВЕТ: "DHKDZOT: TJPM DMMZQZMZIXZ OJRVMY HT YZDOT."

2. Дешифруйте следующие зашифрованные фрагменты текста, используя указанные ключи.

   А. "ZXAI: P RDHIJBT HDBTIXBTH LDGC QN HRDIRWBTC XC PBTGXRP PCS PBTGXRPCH XC HRDIAPCS." (ключ 15).

   ОТВЕТ: "KILT: A COSTUME SOMETIMES WORN BY SCOTCHMEN IN AMERICA AND AMERICANS IN SCOTLAND."

   Б. "MQTSWXSV: E VMZEP EWTMVERX XS TYFPMG LSRSVW." (ключ 4).

   ОТВЕТ: "IMPOSTOR: A RIVAL ASPIRANT TO PUBLIC HONORS."

3. Зашифруйте следующее предложение, используя ключ 0:

"THIS IS A SILLY EXAMPLE."

ОТВЕТ: "THIS IS A SILLY EXAMPLE."

4. Ниже приведены пары исходных слов и их зашифрованных версий. Какие ключи использовались в каждом случае?

   А. ROSEBUD – LIMYVOX

   ОТВЕТ: 20.

   Б. YAMAMOTO – PRDRDFKF

   ОТВЕТ: 17.

   В. ASTRONOMY – HZAYVUVTF

   ОТВЕТ: 7.

5. Как будет выглядеть приведенное ниже предложение, зашифрованное с помощью ключа 8, если дешифровать его с помощью ключа 9?

   "UMMSVMAA: CVKWUUWV XIBQMVKM QV XTIVVQVO I ZMDMVOM BPIB QA EWZBP EPQTM."

   ОТВЕТ: LDDJMDRR: TMBNLLNM OZSHDMBD HM OKZMMHMF Z QDUDMFD SGZS HR VNQSG VGHKD. (Дешифрование с ключом, не являющимся корректным, приводит к бессмысленному тексту.)

Глава 2

1. Как выглядит оператор деления: / или \?

   ОТВЕТ: /. (Символ обратной косой черты \ используется для экранирования символов в строках, описанного в главе 3.)

2. Какое из приведенных ниже значений является целым, а какое – вещественным?

```
42
3.141592
```

   ОТВЕТ: `42` – целочисленное значение, `3.141592` – вещественное (с плавающей точкой).

3. Какие из приведенных ниже строк не являются выражениями?

```
4 x 10 + 2
3 * 7 + 1
2 +
42
2 + 2
spam = 42
```

ОТВЕТ: `4 x 10 + 2` (оператором умножения является не `x`, а `*`), `2 +` (это неполное выражение) и `spam = 42` (это инструкция, а не выражение).

4. **Если вы введете следующие строки в интерактивной оболочке, то какие результаты отобразятся на экране после выполнения строк ❶ и ❷?**

```
  spam = 20
❶ spam + 20
  SPAM = 30
❷ spam
```

ОТВЕТ: строка ❶ выведет `40`, строка ❷ — `20` (`spam` и `SPAM` — две разные переменные).

Глава 3

1. **Если выполнить инструкцию присваивания `spam = 'Cats'`, то что выведут следующие строки кода?**

```
spam + spam + spam
```

ОТВЕТ: `'CatsCatsCats'`

```
spam * 3
```

ОТВЕТ: `'CatsCatsCats'`

2. **Что выведут следующие строки кода?**

```
print("Dear Alice,\nHow are you?\nSincerely,\nBob")
```

ОТВЕТ:

```
Dear Alice,
How are you?
Sincerely,
Bob
```

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

ОТВЕТ: `HelloHello`

3. **Если выполнить инструкцию присваивания `spam = 'Four score and seven years is eighty seven years.'`, то что выведет каждая из следующих строк кода?**

```
print(spam[5])
```

ОТВЕТ: `s`

```
print(spam[-3])
```

ОТВЕТ: `r`

```
print(spam[0:4] + spam[5])
```

ОТВЕТ: `Fours`

```
print(spam[-3:-1])
```

ОТВЕТ: `rs`

```
print(spam[:10])
```

ОТВЕТ: `Four score`

```
print(spam[-5:])
```

ОТВЕТ: `ears.`

```
print(spam[:])
```

ОТВЕТ: `Four score and seven years is eighty seven years.`

4. **В каком окне отображается подсказка >>>: в окне интерактивной оболочки или в окне редактора файлов?**

   ОТВЕТ: в окне интерактивной оболочки.

5. **Что выведет следующая строка кода?**

```
#print('Hello, world!')
```

   ОТВЕТ: она не выведет ничего.

Глава 4

1. **Что выведет на экран следующая инструкция?**

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

   ОТВЕТ: `10`. (Оба вызова `len()` возвращают целочисленные значения, и сложение двух целых чисел `5` дает число `10`, которое и выводится на экран.)

2. **Что выведет на экран следующий код?**

```
i = 0
while i < 3:
    print('Hello')
    i = i + 1
```

ОТВЕТ:

```
Hello
Hello
Hello
```

3. **А что выведет такой код?**

```
i = 0
spam = 'Hello'
while i < 5:
    spam = spam + spam[i]
    i = i + 1
print(spam)
```

ОТВЕТ: `HelloHello`

4. **А такой?**

```
i = 0
while i < 4:
    while i < 6:
        i = i + 2
        print(i)
```

ОТВЕТ:

```
2
4
6
```

Глава 5

1. **Используйте программу *caesarCipher.py* для шифрования следующих сообщений с помощью указанных ключей.**

   А. `'"You can show black is white by argument," said Filby, "but you will never convince me."'` **(ключ 8).**

   ОТВЕТ: `"gw3EkivE1pw5EjtiksEq1E5pq2mEj7Eizo3umv2,"E1iqlE Nqtj7,E"j32E7w3E5qttEvm4mzEkwv4qvkmEumH"`

   Б. `'1234567890'` **(ключ 21).**

   ОТВЕТ: `HIJKLMNOPQ`

2. **Используйте программу *caesarCipher.py* для дешифрования следующих зашифрованных сообщений с помощью указанных ключей.**

   А. `'Kv?uqwpfu?rncwukdng?gpqwijB'` **(ключ 2).**

   ОТВЕТ: `It sounds plausible enough.`

Б. `'XCBSw88S18A1S 2SB41SE .8zSEwAS50D5A5x81V'` (**ключ 22**).

ОТВЕТ: `But all else of the world was invisible.`

3. **Какая инструкция Python будет импортировать модуль *watermelon.py*?**

   ОТВЕТ: `import watermelon`

4. **Что будут выводить на экран следующие фрагменты кода?**

   А.

```
spam = 'foo'
for i in spam:
    spam = spam + i
print(spam)
```

   ОТВЕТ: `foofoo`

   Б.

```
if 10 < 5:
    print('Hello')
elif False:
    print('Alice')
elif 5 != 5:
    print('Bob')
else:
    print('Goodbye')
```

   ОТВЕТ: `Goodbye`

   В.

```
print('f' not in 'foo')
```

   ОТВЕТ: `False`

   Г.

```
print('foo' in 'f')
```

   ОТВЕТ: `False`

   Д.

```
print('hello'.find('oo'))
```

   ОТВЕТ: `-1`. (Если строка не найдена, метод `find()` возвращает значение `-1`.)

Глава 6

1. Взломайте следующий зашифрованный текст, дешифруя по одной строке за раз, поскольку строки зашифрованы с использованием разных ключей. Не забудьте о необходимости экранирования кавычек любого вида.

```
qeFIP?eGSeECNNS,
5coOMXXcoPSZIWoQI,
avnl1olyD4l'ylDohww6DhzDjhuDil,

z.GM?.cEQc. 70c.7KcKMKHA9AGFK,
?MFYp2pPJJUpZSIJWpRdpMFY,
ZqH8sl5HtqHTH4s3lyvH5zH5spH4t pHzqHlH3l5K

Zfbi,!tif!xpvme!qspcbcmz!fbu!nfA
```

ОТВЕТ:

```
I love my kitty,
My kitty loves me,
Together we're happy as can be,


Though my head has suspicions,
That I keep under my hat,
Of what if I shrank to the size of a rat.


Yeah, she would probably eat me.
```

Глава 7

1. Зашифруйте перестановочным шифром с ключом 9 приведенные ниже сообщения, используя карандаш и бумагу. Количество символов в каждом из них предоставлено исключительно из соображений удобства.

   - "Underneath a huge oak tree there was of swine a huge company," (61 символ).

     ОТВЕТ: Uhot on ahoamdakef pe r harhtesunnur wgyegewie,aeean t sec

   - "That grunted as they crunched the mast: For that was ripe and fell full fast." (77 символов).

     ОТВЕТ: Tteeshiefheydtaplaad :telst ct t arhFwaf.gsueoanur n rsdlutcm lnhhatrf

- "Then they trotted away for the wind grew high: One acorn they left, and no more might you spy." (94 символа).

  ОТВЕТ: T atg:renishtwhr nfogperaeeO t hynoy wnt,mt. t w eh o ttfih earyheoniayneoedrdgc d uy hol m

2. В приведенной ниже программе переменные `spam` являются глобальными или локальными?

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

    print(spam)
```

ОТВЕТ: обе переменные глобальные (инструкция `global` делает переменную `spam` в функции `foo()` глобальной).

3. Каким будет результат вычисления каждого из приведенных ниже выражений?

```
[0, 1, 2, 3, 4][2]
```

ОТВЕТ: `2`

```
[[1, 2], [3, 4]][0]
```

ОТВЕТ: `[1, 2]`

```
[[1, 2], [3, 4]][0][1]
```

ОТВЕТ: `2`

```
['hello'][0][1]
```

ОТВЕТ: `'e'`

```
[2, 4, 6, 8, 10][1:3]
```

ОТВЕТ: `[4, 6]`

```
list('Hello world!')
```

ОТВЕТ: `['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']`

```
list(range(10))[2]
```

ОТВЕТ: `2`

4. Каким будет результат вычисления каждого из приведенных ниже выражений?

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

ОТВЕТ: 2

```
len([])
```

ОТВЕТ: 0

```
len([,', ,', ,'])
```

ОТВЕТ: 3

```
[4, 5, 6] + [1, 2, 3]
```

ОТВЕТ: `[4, 5, 6, 1, 2, 3]`

```
3 * [1, 2, 3] + [9]
```

ОТВЕТ: `[1, 2, 3, 1, 2, 3, 1, 2, 3, 9]`

```
42 in [41, 42, 42, 42]
```

ОТВЕТ: `True`

5. Назовите четыре составных оператора присваивания.

ОТВЕТ: `+=, -=, *=, /=`

Глава 8

1. Воспользовавшись карандашом и бумагой, расшифруйте приведенные ниже сообщения с помощью ключа 9. Символ ▪ обозначает одиночный пробел. Для удобства в конце каждого сообщения указано количество содержащихся в нем символов.

   - H▪cb▪▪irhdeuousBdi▪▪▪prrtyevdgp▪nir▪▪eerit▪eatoreechadihf▪pake n▪ge▪b▪te▪dih▪aoa.da▪tts▪tn (89 символов).

     ОТВЕТ: He picked up the acorn and buried it straight By the side of a river both deep and great.

   - A▪b▪▪drottthawa▪nwar▪eci▪t▪nlel▪ktShw▪leec,hheat▪.na▪▪e▪soogma h▪a▪▪ateniAcgakh▪dmnor▪▪ (86 символов).

     ОТВЕТ: At length he came back, and with him a She And the acorn was grown to a tall oak tree.

   - Bmmsrl▪dpnaua!toeboo'ktn▪uknrwos.▪yaregonr▪w▪nd,tu▪▪oiady▪hgt Rwt▪▪▪A▪hhanhhasthtev▪▪e▪t▪e▪▪eo (93 символа).

ОТВЕТ: But with many a hem! and a sturdy stroke, At length he brought down the poor Raven's own oak.

2. Если ввести в интерактивной оболочке приведенные ниже выражения, то что будет выведено на экран каждой строкой?

```
>>> math.ceil(3.0)
```

ОТВЕТ: 3

```
>>> math.floor(3.1)
```

ОТВЕТ: 3

```
>>> round(3.1)
```

ОТВЕТ: 3

```
>>> round(3.5)
```

ОТВЕТ: 4

```
>>> False and False
```

ОТВЕТ: `False`

```
>>> False or False
```

ОТВЕТ: `False`

```
>>> not not True
```

ОТВЕТ: `True`

3. Составьте таблицы истинности для операторов `and`, `or` и `not`.

Оператор `and`				
A	**and**	**B**	**==**	**Результат**
True	and	True	==	True
True	and	True	==	False
False	and	False	==	False
False	and	False	==	False

Оператор `or`				
A	**or**	**B**	**==**	**Результат**
True	or	True	==	True
True	or	True	==	True
False	or	False	==	True
False	or	False	==	False

Оператор not			
not	**A**	**==**	**Результат**
not	True	==	False
not	False	==	True

4. Какая из приведенных ниже инструкций корректна?

```
if __name__ == '__main__':
if __main__ == '__name__':
if _name_ == '_main_':
if _main_ == '_name_':
```

ОТВЕТ: `if __name__ == '__main__':`

Глава 9

1. Если вы запустили приведенную ниже программу и она вывела на экран число 8, то что будет выведено на экран, когда вы запустите программу в следующий раз?

```
import random
random.seed(9)
print(random.randint(1, 10))
```

ОТВЕТ: 8. (В результате задания одного и того же затравочного числа будут генерироваться одни и те же псевдослучайные числа.)

2. Что выведет на экран следующая программа?

```
spam = [1, 2, 3]
eggs = spam
ham = eggs
ham[0] = 99
print(ham == spam)
```

ОТВЕТ: `True`

3. В каком модуле содержится функция `deepcopy()`?

ОТВЕТ: в модуле `copy`.

4. Что выведет на экран следующая программа?

```
import copy
spam = [1, 2, 3]
eggs = copy.deepcopy(spam)
ham = copy.deepcopy(eggs)
ham[0] = 99
print(ham == spam)
```

ОТВЕТ: `False`

Глава 10

1. Как правильно: `os.exists()` или `os.path.exists()`?

   ОТВЕТ: `os.path.exists()`

2. Когда началась эпоха Unix?

   ОТВЕТ: в полночь 1 января 1970 года (среднее время по Гринвичу).

3. Каким будет результат вычисления следующих выражений?

```
'Foobar'.startswith('Foo')
```

   ОТВЕТ: `True`

```
'Foo'.startswith('Foobar')
```

   ОТВЕТ: `False`

```
'Foobar'.startswith('foo')
```

   ОТВЕТ: `False`

```
'bar'.endswith('Foobar')
```

   ОТВЕТ: `False`

```
'Foobar'.endswith('bar')
```

   ОТВЕТ: `True`

```
'The quick brown fox jumped over the yellow lazy dog.'.title()
```

   ОТВЕТ: `'The Quick Brown Fox Jumped Over The Yellow Lazy Dog.'`

Глава 11

1. Что выведет следующий код?

```
spam = {'name': 'Al'}
print(spam['name'])
```

   ОТВЕТ: `'Al'`

2. Что выведет следующий код?

```
spam = {'eggs': 'bacon'}
print('bacon' in spam)
```

   ОТВЕТ: `False`

3. **Напишите цикл `for` для вывода значений, содержащихся в следующем словаре.**

```
spam = {'name': 'Zophie', 'species': 'cat', 'age': 8}
```

ОТВЕТ:

```
for key in spam:
    print(spam[key])
```

4. **Что выведет следующий код?**

```
print('Hello, world!'.split())
```

ОТВЕТ: `['Hello,', 'world!']`

5. **Что выведет следующий код?**

```
def spam(eggs=42):
    print(eggs)
spam()
spam('Hello')
```

ОТВЕТ:

```
42
Hello
```

6. **Каков процент слов английского языка в следующем предложении?**

```
"Whether it's flobulllar in the mind to quarfalog the slings and
arrows of outrageous guuuuuuuuur."
```

ОТВЕТ: 80% (12 из 15 слов: 12 / 15 = 0,8, т.е. 80%).

Глава 12

1. **Какой результат вернет приведенное ниже выражение?**

```
'   Hello world'.strip()
```

ОТВЕТ: `'Hello world'`

2. **Какие символы являются пробельными?**

ОТВЕТ: символы пробела, табуляции и перехода на новую строку.

3. **Почему вызов `'Hello world'.strip('o')` возвращает строку, которая по-прежнему содержит букву `'o'`?**

ОТВЕТ: потому что метод `strip()` удаляет лишь символы, находящиеся в начале и в конце строки.

4. Почему вызов `'xxxHelloxxx'.strip('X')` возвращает строку, которая по-прежнему содержит букву `'x'`?

   ОТВЕТ: потому что метод `strip('X')` будет удалять только символы `'X'`, но не `'x'`.

Глава 13

1. Каковы результаты вычисления приведенных ниже выражений?

   ```
   17 % 1000
   ```

   ОТВЕТ: `17`

   ```
   5 % 5
   ```

   ОТВЕТ: `0`

2. Чему равен НОД чисел 10 и 15?

   ОТВЕТ: 5 (поскольку 5 — наибольшее из чисел, на которые нацело делятся как 10, так и 15).

3. Что будет содержать переменная `spam` после выполнения инструкции `spam, eggs = 'hello', 'world'`?

   ОТВЕТ: `'hello'`

4. НОД чисел 17 и 31 равен 1. Означает ли это, что числа 17 и 31 являются взаимно простыми?

   ОТВЕТ: да (числа называются взаимно простыми, если их НОД равен 1).

5. Почему числа 6 и 8 не являются взаимно простыми?

   ОТВЕТ: потому что их НОД равен 2, а не 1.

6. Напишите формулу для модульного обращения выражения A mod C.

   ОТВЕТ: модульным обращением i является число, удовлетворяющее соотношению $(A \cdot i) \% C = 1$.

Глава 14

1. Комбинацией каких двух шифров является аффинный шифр?

   ОТВЕТ: мультипликативного и шифра сдвига (или шифра Цезаря).

2. Что такое кортеж и чем он отличается от списка?

   ОТВЕТ: кортеж — это тип данных, который, как и список, может содержать несколько значений. Однако, в отличие от списка, значения кортежа не могут изменяться.

3. Почему аффинный шифр является слабым, если ключ A равен 1?

   ОТВЕТ: поскольку результатом умножения любого числа на 1 является само число, это означает, что аффинный шифр будет кодировать любую букву той же буквой.

4. Почему аффинный шифр является слабым, если ключ B равен 0?

   ОТВЕТ: поскольку результатом сложения любого числа с 0 является само число, это означает, что аффинный шифр будет кодировать любую букву той же буквой.

Глава 15

1. Чему равно выражение `2 ** 5`?

   ОТВЕТ: `32`

2. Чему равно выражение `6 ** 2`?

   ОТВЕТ: `36`

3. Что выводит следующий код?

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

   ОТВЕТ:

```
0
1
3
4
```

4. Выполняется ли функция `main()`, определенная в файле *affineHacker.py*, если другая программа импортирует модуль `affineHacker`?

   ОТВЕТ: нет. (Если файл импортируется другой программой, то переменной `__name__` автоматически присваивается значение `'affineHacker'`, поэтому функция `main()` не вызывается.)

Глава 16

1. Почему взлом простого подстановочного шифра методом грубой силы невозможен даже с помощью сверхмощного компьютера?

   ОТВЕТ: даже для самого мощного компьютера количество возможных ключей оказывается слишком большим.

2. Что будет содержать переменная `spam` после выполнения следующего кода?

```
spam = [4, 6, 2, 8]
spam.sort()
```

ОТВЕТ: `[2, 4, 6, 8]`

3. Что такое функция-обертка?

   ОТВЕТ: функция-обертка вызывает другую функцию, передавая ей свои аргументы, и возвращает значение, которое возвращает эта функция.

4. Что возвращает метод `'hello'.islower()`?

   ОТВЕТ: `True`

5. Что возвращает метод `'HELLO 123'.isupper()`?

   ОТВЕТ: `True`

6. Что возвращает метод `'123'.islower()`?

   ОТВЕТ: `False`. (Чтобы метод `islower()` вернул значение `True`, строка должна содержать хотя бы одну букву в нижнем регистре и не содержать букв в верхнем регистре.)

Глава 17

1. Каким будет шаблон для слова "hello"?

   ОТВЕТ: 0.1.2.2.3

2. Имеют ли слова "mammoth" и "goggles" общий шаблон?

   ОТВЕТ: да (0.1.0.0.2.3.4).

3. Какое из слов — "Alleged", "efficiently" или "poodle" — является возможным вариантом дешифрования шифрослова "PYYACAO"?

   ОТВЕТ: "Alleged" (строки "PYYACAO" и "Alleged" имеют общий шаблон: 0.1.1.2.3.2.4).

Глава 18

1. Какому шифру аналогичен шифр Виженера, если не считать того, что в нем используется несколько ключей вместо одного?

   ОТВЕТ: шифр Виженера аналогичен шифру Цезаря. (Если выбрать для шифра Виженера ключ длиной один символ, то он станет идентичным шифру Цезаря.)

2. Сколько возможных вариантов существует для ключа шифра Виженера длиной 10 символов?

А. Сотни.

Б. Тысячи.

В. Миллионы.

Г. Более триллиона.

ОТВЕТ: вариант *г)*, т.е. более триллиона (точнее, количество возможных ключей равно 141 167 095 653 376).

3. К какому типу шифров относится шифр Виженера?

   ОТВЕТ: шифр Виженера относится к полиалфавитным сдвиговым шифрам, поскольку он представляет собой шифр сдвига (подобный шифру Цезаря), в котором используется несколько наборов подстановочных символов.

Глава 19

1. Что такое частотный анализ?

   ОТВЕТ: это раздел криптоанализа, связанный с подсчетом частотности букв в шифротексте.

2. Какие шесть букв чаще всего встречаются в текстах на английском языке?

   ОТВЕТ: ETAOIN.

3. Что будет содержать переменная `spam` после выполнения следующего кода?

```
spam = [4, 6, 2, 8]
spam.sort(reverse=True)
```

   ОТВЕТ: `[8, 6, 4, 2]`

4. Если переменная `spam` содержит словарь, то как получить список его ключей?

   ОТВЕТ: `list(spam.keys())`

Глава 20

1. Что такое перебор по словарю?

   ОТВЕТ: это атака методом грубой силы, суть которой заключается в полном переборе слов из словаря в качестве возможных ключей.

2. Какую информацию о шифротексте предоставляет метод Касиски?

   ОТВЕТ: метод Касиски может помочь в определении длины ключа Виженера, примененного в шифротексте.

3. Какие два изменения происходят при преобразовании списка в множество с помощью функции `set()`?

   ОТВЕТ: удаляются дубликаты значений, и теряется упорядоченность значений (в отличие от списков элементы множеств не упорядочиваются).

4. Переменная `spam` представляет собой список `['cat', 'dog', 'mouse', 'dog']`, содержащий четыре элемента. Сколько элементов будет содержать список, возвращаемый функцией `list(set(spam))`?

   ОТВЕТ: 3 (удаляется дубликат строки `'dog'`).

5. Что выведет следующий код?

```
print('Hello', end='')
print('World')
```

   ОТВЕТ: `HelloWorld` (в одной строке).

Глава 21

1. Почему в этой главе мы не создавали программу для одноразового шифроблокнота?

   ОТВЕТ: потому что использование шифра Виженера, в котором длина случайного ключа совпадает с длиной сообщения, равносильно применению одноразового шифроблокнота.

2. Эквивалентом какого шифра является двухразовый шифроблокнот?

   ОТВЕТ: повторное применение одноразового шифроблокнота эквивалентно шифрованию текста с помощью шифра Виженера.

3. Если использовать ключ, длина которого в два раза превышает размер исходного сообщения, то станет ли шифрование с помощью одноразового шифроблокнота в два раза более безопасным?

   ОТВЕТ: нет. Стойкость шифра останется той же, поскольку дополнительные буквы ключа не используются.

Глава 22

1. Сколько всего существует простых чисел?

   ОТВЕТ: количество простых чисел бесконечно велико. Такого понятия, как "наибольшее" простое число, не существует.

2. Как называются целые числа, не являющиеся простыми?

   ОТВЕТ: такие числа называются *составными*.

3. Назовите два алгоритма нахождения простых чисел?

   ОТВЕТ: можно назвать даже три: перебор делителей, тест Миллера — Рабина и решето Эратосфена.

Глава 23

1. В чем разница между симметричными и асимметричными шифрами?

   ОТВЕТ: в симметричном шифре один и тот же ключ используется как для шифрования, так и для дешифрования сообщений. В асимметричном шифре для этих целей используются два разных ключа.

2. Алиса сгенерировала открытый и закрытый ключи. К сожалению, впоследствии она потеряла закрытый ключ.
3. Смогут ли другие люди отправлять ей зашифрованные сообщения?

   ОТВЕТ: да. Для отправки Алисе зашифрованных сообщений другие люди смогут использовать открытый ключ Алисы.

4. Сможет ли она дешифровать сообщения, отправленные ей ранее?

   ОТВЕТ: нет. Для дешифрования этих сообщений требуется знать ее закрытый ключ.

5. Сможет ли она снабжать документы цифровой подписью?

   ОТВЕТ: нет. Для создания цифровой подписи Алисе потребуется ее закрытый ключ.

6. Смогут ли другие люди верифицировать подписанные ею ранее документы?

   ОТВЕТ: да. Для верификации документов, подписанных Алисой до потери ею своего закрытого ключа, они смогут использовать открытый ключ Алисы.

7. Что такое аутентификация и конфиденциальность? В чем разница между ними?

   ОТВЕТ: аутентификация — это процедура верификации личности, позволяющая удостовериться в том, что человек действительно тот, за кого себя выдает. Конфиденциальность — это обеспечение недоступности информации для посторонних лиц и предоставление доступа к ней лишь тем, для кого она предназначена.

8. Что такое невозможность отказа?

   ОТВЕТ: это свойство цифровых подписей, гарантирующее, что владелец подписи не сможет впоследствии отрицать факт подписания документа.

Предметный указатель

И

К

Л

М

Н

О

П

Р

С

Т

Ф

Х

Ц

Ч

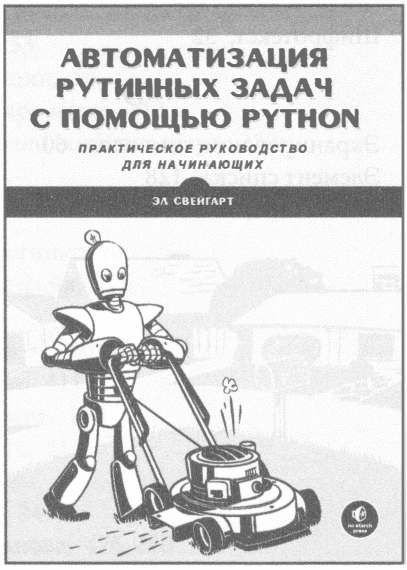
Ш

Э

АВТОМАТИЗАЦИЯ РУТИННЫХ ЗАДАЧ С ПОМОЩЬЮ PYTHON

практическое руководство для начинающих

Эл Свейгарт



www.williamspublishing.com

Книга научит вас использовать Python для написания программ, способных в считанные секунды сделать то, на что раньше у вас уходили часы ручного труда, причем никакого опыта программирования от вас не требуется. Как только вы овладеете основами программирования, вы сможете создавать программы на языке Python, которые будут без труда выполнять в автоматическом режиме различные полезные задачи, такие как:

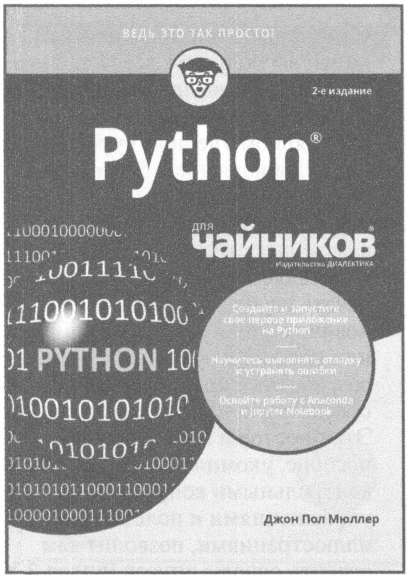
- поиск определенного текста в файле или в множестве файлов;
- создание, обновление, перемещение и переименование файлов и папок;
- поиск в Интернете и загрузка онлайн-контента;
- обновление и форматирование данных в электронных таблицах Excel любого размера;
- разбиение, слияние, разметка водяными знаками и шифрование PDF-документов;
- рассылка напоминаний в виде сообщений электронной почты или текстовых уведомлений;
- заполнение онлайновых форм.

ISBN 978-5-6040724-2-4 **в продаже**

PYTHON ДЛЯ ЧАЙНИКОВ

2-Е ИЗДАНИЕ

Джон Пол Мюллер



www.dialektika.com

Python — это мощный язык программирования, на котором можно создавать самые разные приложения, не зависящие от платформы. Он идеально подходит для новичков, особенно если нужно быстро научиться программировать и начать создавать реальные проекты. Благодаря пошаговым инструкциям, приведенным в книге, вы сможете в краткие сроки освоить основы языка. Работая в среде Jupyter Notebook, вы будете применять принципы грамотного программирования для создания смешанного представления кода, заметок, математических уравнений и графиков.

Основные темы книги:

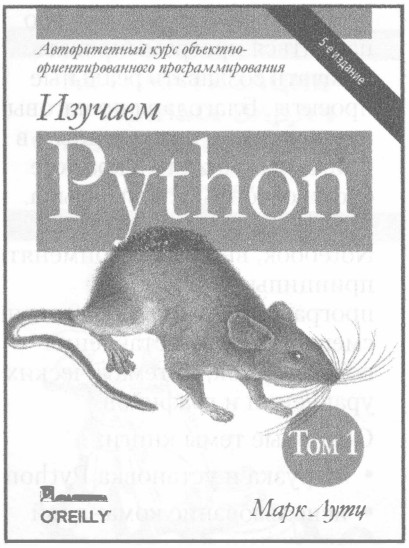
- загрузка и установка Python;
- использование командной строки;
- знакомство со средой Jupyter Notebook;
- основы программирования на Python;
- создание коллекций и списков;
- взаимодействие с пакетами;
- поиск и устранение ошибок.

ISBN: 978-5-907144-26-2

в продаже

ИЗУЧАЕМ PYTHON
ТОМ 1
5-Е ИЗДАНИЕ

Марк Лутц



www.williamspublishing.com

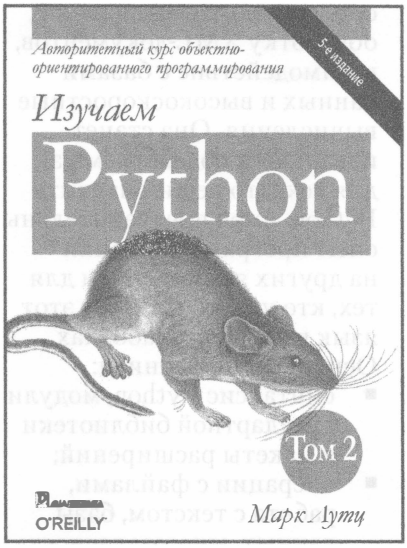
С помощью этой практической книги вы получите всестороннее и глубокое введение в основы языка Python. Будучи основанным на популярном учебном курсе Марка Лутца, обновленное 5-е издание книги поможет вам быстро научиться писать эффективный высококачественный код на Python. Она является идеальным способом начать изучение Python, будь вы новичок в программировании или профессиональный разработчик программного обеспечения на других языках. Это простое и понятное учебное пособие, укомплектованное контрольными вопросами, упражнениями и полезными иллюстрациями, позволит вам освоить основы линеек Python 3.X и 2.X. Вы также ознакомитесь с расширенными возможностями языка, получившими широкое распространение в коде Python.

ISBN 978-5-907144-52-1

в продаже

ИЗУЧАЕМ PYTHON
ТОМ 2
5-Е ИЗДАНИЕ

Марк Лутц



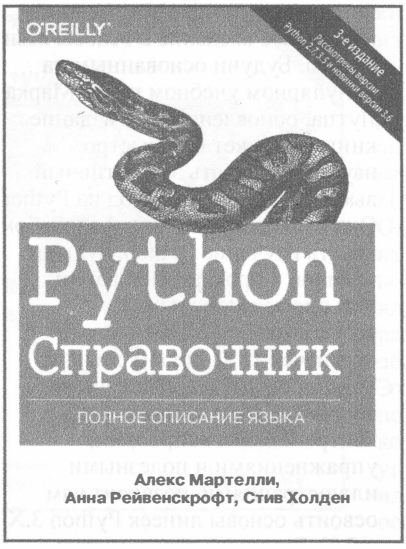
www.williamspublishing.com

С помощью этой практической книги вы получите всестороннее и глубокое введение в основы языка Python. Будучи основанным на популярном учебном курсе Марка Лутца, обновленное 5-е издание книги поможет вам быстро научиться писать эффективный высококачественный код на Python. Она является идеальным способом начать изучение Python, будь вы новичок в программировании или профессиональный разработчик программного обеспечения на других языках. Это простое и понятное учебное пособие, укомплектованное контрольными вопросами, упражнениями и полезными иллюстрациями, позволит вам освоить основы линеек Python 3.X и 2.X. Вы также ознакомитесь с расширенными возможностями языка, получившими широкое распространение в коде Python.

ISBN 978-5-907144-53-8 **в продаже**

PYTHON СПРАВОЧНИК
ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА 3-Е ИЗДАНИЕ

Алекс Мартелли
Анна Рейвенскрофт
Стив Холден



Книга охватывает чрезвычайно широкий спектр областей применения Python, включая веб-приложения, сетевое программирование, обработку XML-документов, взаимодействие с базами данных и высокоскоростные вычисления. Она станет идеальным подспорьем как для тех, кто решил изучить Python, имея предварительный опыт программирования на других языках, так и для тех, кто уже использует этот язык в своих разработках. Основные темы книги:

- синтаксис Python, модули стандартной библиотеки и пакеты расширений;
- операции с файлами, работа с текстом, базы данных, многозадачность и обработка числовых данных;
- основы работы с сетями, и клиентские модули сетевых протоколов;
- модули расширения Python, средства пакетирования и распространения расширений, модулей и приложений.

www.williamspublishing.com

ISBN 978-5-6040723-8-7

в продаже

ИЗУЧИТЕ PYTHON, ВЗЛАМЫВАЯ СЕКРЕТНЫЕ ШИФРЫ

РАССМОТРЕН PYTHON 3

Научитесь программировать на Python, создавая и взламывая шифры, с помощью которых пересылаются секретные сообщения!

После знакомства с основами программирования на Python вы узнаете, как создавать, тестировать и взламывать классические шифры, включая перестановочный шифр и шифр Виженера. Постепенно мы перейдем от простых алгоритмов, таких как обратный шифр и шифр Цезаря, к обсуждению криптосистем с открытым ключом, применяемых в наши дни для защиты онлайн-транзакций.

В каждой главе приводится полноценная программа с пошаговым описанием алгоритма ее работы. Прочитав книгу, вы научитесь программировать на Python и сможете создавать собственные криптографические системы!

Основные темы книги

- Создание криптографических приложений на Python
- Применение словарей для быстрой проверки того, содержит ли дешифрованное сообщение осмысленный текст на английском языке или случайный набор букв
- Создание тестов, позволяющих убедиться в том, что код шифрования и дешифрования работает корректно
- Программирование (и взлом!) аффинного шифра, в котором для шифрования сообщения применяется модульная арифметика
- Взлом шифров методом грубой силы и с помощью частотного анализа

Об авторе

Эл Свейгарт — профессиональный разработчик, автор множества книг по программированию, включая бестселлер *Автоматизация рутинных задач с помощью Python*. Многие его книги доступны на сайте `inventwithpython.com`.

Категория: компьютерные технологии/криптография/Python

ДИАЛЕКТИКА
www.dialektika.com

no starch press
www.nostarch.com

ISBN 978-5-907203-02-0
9 785907 203020 19148