Думать на языке Python

Думать как компьютерный специалист

Allen Downey

Green Tea Press

Needham, Massachusetts

Версия 1.1.24+Kart [Python 3.2]

Версия русского перевода: 1.06

Оригинальное название книги (на английском):
Think Python: How to Think Like a Computer Scientist
Автор: Allen Downey
Издательство: *Green Tea Press*, Needham, Massachusetts

История публикаций:

Апрель 2002: Первое издание How to Think Like a Computer Scientist.

Август 2007: Переработанное издание, название изменено на *How to Think Like a (Python) Programmer*.

Июнь 2008: Переработанное издание, название изменено на *Think Python: How to Think Like a Computer Scientist*.

Информация об издательстве:

Green Tea Press

9 Washburn Ave

Needham MA 02492

Оригинал этой книги создан в формате LaTeX. Это означает, что его можно преобразовывать в другие форматы, которые будут отображаться одинаково вне зависимости от используемой платформы.

Исходный LaTeX код этой книги (на английском) доступен по адресу: `http://www.thinkpython.com`

Русский перевод:

Об ошибках в русской версии вы можете сообщить по адресу:

`thinkpython [ at ] yandex.com`

Оглавление

Предисловие
Интересная история написания этой книги

В январе 1999 года я готовился преподавать вступительные уроки по языку Java. Я уже проводил подобный курс три раза и совсем уже начал отчаиваться. Уровень неуспеваемости у моих студентов был настолько высок, что даже среди успевающих студентов общий уровень был слишком низок.

Я заметил, что одной из причин тому были книги. Они были слишком большими, с огромным количеством необязательной информации о деталях языка Java и недостаточно внятным руководством о том, как нужно, собственно, программировать. Все студенты попадались в одну и ту же ловушку: вначале они усваивали материал легко, постепенно прогрессировали, а затем, где-то в районе 5 главы, дверца захлопывалась — было слишком много нового материала, и его преподносили слишком быстро. Мне приходилось остаток семестра тратить на то, чтобы все эти разрозненные сведения собрать воедино.

За две недели до начала занятий я решил написать собственную книгу. Я преследовал следующие цели:

- Краткость. Лучше прочитать 10 страниц, чем не прочитать 50.
- Внимательность по отношению к используемым терминам. Я старался свести использование жаргонных слов к минимуму и давать определения новым терминам по мере их появления.
- Постепенность в усложнении материала. Чтобы студенты не попадались в ловушку, я взял самые трудные темы и разбил их на несколько маленьких шагов.
- Концентрация на самом процессе программирования, а не на свойствах данного языка.

Мне нужно было придумать название для книги, и я выбрал следующее: *Как думать как компьютерный специалист* (How to Think Like a Computer Scientist).

Моя первая версия книги была довольно грубой, но она работала. Студенты действительно ее читали и понимали в достаточной мере, так что в классе я мог сосредотачиваться на трудных и интересных темах и, что самое важное, у студентов было время попрактиковаться в классе.

Я выпустил эту книгу под свободной лицензией (GNU Free Documentation License), что позволяет пользователям копировать, модифицировать и распространять эту книгу.

Дальше случилось нечто интересное. Джефф Элкнер (Jeff Elkner), школьный учитель из штата Вирджиния, адаптировал мою книгу к языку Python. Он послал мне копию своего перевода, и я приобрел не совсем обычный опыт изучения языка Python через чтение своей собственной книги.

Мы с Джеффом пересмотрели книгу, вставили туда разделы с углубленным изучением (case study), написанные Крисом Мейерсом (Chris Meyers) и выпустили ее в 2001 году с названием *Как думать как компьютерный специалист: изучение Python* (How to Think Like a Computer Scientist: Learning Python) под свободной лицензией (GNU Free Documentation License). В качестве издательства Green Tea Press я выпустил эту книгу и начал продавать бумажные версии через интернет-магазин Amazon.com и книжные магазины при колледжах. В издательстве Green Tea Press есть и другие книги. Вы можете найти их по адресу: `greenteapress.com`.

В 2003 году я начал работать в Olin College. Там я начал преподавать Python впервые. Контраст с языком Java был разительным. Студенты прилагали меньше усилий, узнавали больше, работали над более интересными проектами и, в общем итоге, получали больше удовольствия.

В течение последних пяти лет я продолжал развивать эту книгу, исправляя ошибки, улучшая некоторые из примеров и добавляя новый материал, особенно упражнения. В 2008 году я начал полностью пересматривать книгу. В то же самое время со мной связался редактор университетского издательства из Кембриджа (Cambridge University Press), который был заинтересован в выпуске следующего издания.

Результатом является данная книга с менее грандиозным названием *Думайте на языке Python* (Think Python). Вот некоторые отличия:

- В конце каждой главы я добавил раздел по отладке. Эти разделы описывают общую методику отыскания и предотвращения ошибок, а также описывают и те, которые являются специфическими для языка Python.
- Я удалил материал из последних нескольких глав о построении списков и древовидных данных.

Мне нравятся эти темы, но я посчитал, что они не совсем вписываются в остальную часть книги.

- Я добавил больше упражнений: от простых тестов на понимание пройденного материала до довольно существенных проектов.
- Я добавил несколько глав с углубленным изучением – более длинные примеры с упражнениями, решениями и обсуждением проблем. Некоторые из них базируются на Swampy – наборе программ Python, которые я написал для использования у себя в классе. Сам набор Swampy, примеры кода и некоторые решения доступны по адресу: `thinkpython.com`.
- Я расширил обсуждение разработки программ по плану и основные шаблонные методы программирования (design patterns).

Я надеюсь, что вам понравится работать с этой книгой, а также, что она поможет вам научиться программировать и думать, хотя бы чуть-чуть так, как думает компьютерный специалист.

Allen B. Downey

Needham MA

Allen Downey является адъюнкт-профессором компьютерных наук (Associate Professor of Computer Science) в колледже Franklin W. Olin College of Engineering.

Благодарности

Прежде всего и больше всех я благодарю Джеффа Элкнера (Jeff Elkner), который адаптировал мою книгу с Java на Python, начал весь этот проект и познакомил меня с тем, что стало моим любимым языком программирования.

Я также благодарен Крису Мейерсу (Chris Meyers), написавшему несколько разделов для книги *Как думать как компьютерный специалист*.

Я благодарен фонду свободного программного обеспечения (Free Software Foundation) за их разработку свободной лицензии GNU Free Documentation License, что помогло моей совместной работе с Джеффом и Крисом.

Также я благодарен редакторам в Lulu, которые работали над книгой *Как думать как компьютерный специалист*.

Я благодарен всем студентам, работавшим с ранними версиями этой книги, и всем тем, кто внесли свой вклад (перечислены ниже), посылая исправления и предложения.

Я благодарен моей жене Лизе (Lisa) за ее помощь в работе над этой книгой, и издательству Green Tea Press, а также всем остальным.

Список внесших свой вклад

Более сотни внимательных и вдумчивых читателей присылали мне свои предложения и исправления в течение последних нескольких лет. Их вклад и энтузиазм оказал неоценимую помощь этому проекту.

Если у вас есть предложения или замеченные ошибки, пожалуйста, шлите их по адресу

`feedback@thinkpython.com`. Если я приму ваши изменения, то в следующем издании я внесу вас в список внесших вклад (если только вы не попросите, чтобы я не вносил вас туда).

Если вы приведете как минимум часть предложения, в котором встречается ошибка, это упростит мне поиск. Номера страниц и разделов тоже хороши, но не настолько, как часть предложения. Спасибо!

- Lloyd Hugh Allen прислал исправление в разделе 8.4
- Yvon Boulianne прислал исправление семантической ошибки из главы 5.
- Fred Bremmer прислал исправление в разделе 2.1
- Jonah Cohen написал Perl скрипт для конвертирования LaTeX-версии этой книги в HTML.
- Michael Conlon прислал исправление грамматической ошибки из главы 2, а также улучшения, касающиеся стиля главы 1. Также он был тем, кто начал дискуссию по техническим аспектам интерпретаторов.

- Benoit Girard прислал исправление комичной ошибки из раздела 5.6
- Courtney Gleason и Katherine Smith написали файл `horsebet.py`, который использовался в ранней версии этой книги. Их программу можно найти на указанном выше сайте.
- Lee Harr прислал исправлений больше, чем мы можем здесь разместить, и, по-справедливости, его бы следовало назвать одним из главных редакторов текста.
- James Kaylin – студент, который использует эту книгу. Он прислал множество исправлений.
- David Kershaw исправил ошибку в функции `catTwice` из раздела 3.10.
- Eddie Lam прислал многочисленные исправления к главам 1, 2 и 3. Также он исправил Makefile таким образом, что тот создает индекс при первом запуске. Также он помог нам создать схему нумерации версий книги.
- Man-Yong Lee прислал исправление к коду из примера в разделе 2.4.
- David Mayo указал на то, что слово "unconsciously" (*бессознательно*) из главы 1 необходимо изменить на "subconsciously" (*подсознательно*).
- Chris McAloon прислал несколько исправлений для разделов 3.9 и 3.10.
- Matthew J. Moelter в течение долгого времени присылал исправления и предложения к книге.
- Simon Dicon Montford сообщил об отсутствующем определении функции и нескольких опечатках в главе 3. Он также нашел ошибки в функции `encrement` из главы 13.
- John Ouzts исправил определение "return value" (возвращаемое значение) в главе 3.
- Kevin Parks прислал ценные комментарии и предложения по поводу того, как улучшить распространение этой книги.
- David Pool указал на опечатку в словаре терминов из главы 1, а также сказал несколько слов ободрения.
- Michael Schmitt прислал исправления к главе о файлах и исключениях.
- Robin Shaw указал на ошибку в разделе 13.1, где функция `printTime` использовалась в примере, не будучи определена.
- Paul Sleigh нашел ошибку в главе 7, а также ошибку в Perl скрипте, написанном Jonah Cohen для создания HTML из LaTeX.
- Craig T. Snydal пробует применять эту книгу в университете Drew University. Он прислал несколько ценных предложений и исправлений.
- Ian Thomas и его студенты пользуются этой книгой на курсе по программированию. Они были первыми, кто испытывал главы из последней половины книги, и сделали многочисленные исправления и внесли множество предложений.
- Keith Verheyden прислал исправление к главе 3.
- Peter Winstanley указал нам на долго продержавшуюся ошибку в главе 3, вызванную нашим слабым знанием латинского языка.
- Chris Wrobel сделал исправления в коде из главы, объясняющей ввод/вывод файлов и исключения.
- Moshe Zadka внес неоценимый вклад в этот проект. Кроме того, что он написал черновой вариант главы, посвященной словарям, он также давал ценные руководящие советы на ранних стадиях написания книги.
- Christoph Zwerschke прислал несколько исправлений и педагогических советов. Также он объяснил разницу между *gleich* и *selbe*.
- James Mayer прислал полный набор орфографических ошибок и опечаток, две из которых были в списке внесших свой вклад.
- Hayden McAfee указал на непоследовательность между двумя примерами, потенциально способную внести путаницу.
- Angel Arnal является частью международной команды, работающий с испанской версией этой

книги. Он также обнаружил несколько ошибок в английской версии.

- Tauhidul Hoque и Lex Berezhn создали иллюстрации к главе 1 и улучшили много других иллюстраций.
- Dr. Michele Alzetta обнаружил ошибку в главе 8, а также прислал несколько педагогических советов и предложений по поводу чисел Фибоначчи и Old Maid (старая дева).
- Andy Mitchell нашел опечатку в главе 1 и неработающий пример в главе 2.
- Kalin Harvey сделал несколько предложений, делающих главу 7 более ясной, а также обнаружил несколько опечаток.
- Christopher P. Smith обнаружил несколько опечаток и помог нам обновить книгу до версии языка Python 2.2.
- David Hutchins обнаружил несколько опечаток в предисловии.
- Gregor Lingl преподает Python в одной из школ Вены, Австрия. Он работает над немецким переводом этой книги и обнаружил пару досадных ошибок в главе 5.
- Julie Peters обнаружила несколько ошибок в предисловии.
- Florin Oprina сделал улучшение в функции `makeTime`, исправил ошибку в функции `printTime`, а также обнаружил несколько опечаток.
- D. J. Webre внес предложения по улучшению ясности в главе 3.
- Ken нашел несколько ошибок в главах 8, 9 и 11.
- Ivo Wever нашел ошибку в главе 5, а также внес предложение по улучшению ясности главы 3.
- Curtis Yanko предложил, как улучшить ясность в главе 2.
- Ben Logan указал на множество опечаток, а также помог решить проблемы с конвертированием книги в HTML.
- Jason Armstrong указал на отсутствующее слово в главе 2.
- Louis Cordier указал на место в главе 16, в котором код не соответствовал тексту.
- Brian Cain внес несколько предложений по улучшению ясности глав 2 и 3.
- Rob Black прислал множество исправлений, включая те, что относились к версии Python 2.2.
- Jean-Philippe Rey из Ecole Centrale Paris прислал множество патчей, включая некоторые обновления для Python 2.2, а также предложил другие улучшения.
- Jason Mader из университета George Washington University внес множество полезных предложений и исправлений.
- Jan Gundtofte-Bruun напомнил нам, что "a error" должно писаться как "an error" (ошибка).
- Abel David и Alexis Dinno напомнили нам, что множественным числом слова "matrix" (матрица) является слово "matrices", а не "matrixes". Эта ошибка продержалась в книге несколько лет, но два читателя с одинаковыми инициалами сообщили о ней в один и тот же день. Невероятно.
- Charles Thayer подсказал нам, чтобы мы избавились от точек с запятой, которые мы поставили в конце некоторых выражений, а также то, чтобы мы прояснили использование терминов "аргумент" и "параметр".
- Roger Sperberg указал нам на некоторую проблему с логикой в главе 3.
- Sam Bull указал на запутанный параграф из главы 2.
- Andrew Cheung указал на два случая "использования до определения".
- C. Corey Capel обнаружил пропущенное слово в Third Theorem of Debugging (третья теорема отладки), а также на опечатку в главе 4.
- Alessandra помогла прояснить некоторые неясности с Turtle.
- Wim Champagne обнаружил ошибку в примере со словарем.
- Douglas Wright указал на проблему с делением с отбрасыванием дробной части в функции `arc`.

- Jared Spindor обнаружил некоторые лишние знаки в конце предложения.
- Lin Peiheng прислал множество ценных предложений.
- Ray Hagtvedt указал на две ошибки и еще на одну не совсем ошибку.
- Torsten Hübsch указал на непоследовательность в Swampy.
- Inga Petuhhov исправила пример в главе 14.
- Arne Babenhauserheide прислал несколько ценных исправлений.
- Mark E. Casida указал на повторяющиеся слова.
- Scott Tyler вставил нечто пропущенное. А затем прислал набор исправлений.
- Gordon Shephard прислал несколько исправлений, все были в разных email-ах.
- AndrewTurner обнаружил ошибку в главе 8.
- Adam Hobart исправил ошибку деления с отбрасыванием дробной части в функции `arc`.
- Daryl Hammond и Sarah Zimmerman указали на то, что я предоставил `math.pi` слишком рано. И еще Zim указала на опечатку.
- George Sass нашел баг в разделе Отладка.
- Brian Bingham предложил упражнение 11.8.
- Leah Engelbert-Fenton указал на то, что я использовал `tuple` в качестве имени переменной вопреки своему же собственному совету. Также он нашел ряд опечаток и случай "использования до определения".
- Joe Funke обнаружил опечатку.
- Chao-chao Chen обнаружил непоследовательность в примере с числами Фибоначчи.
- Jeff Paine знает, чем отличается "space" (пробел) от "spam" (спам).
- Lubos Pintes прислал опечатку.
- Gregg Lind и Abigail Heithoff предложили упражнение 14.6.
- Max Hailperin прислал множество исправлений и предложений. Max является одним из авторов экстраординарной книги *Concrete Abstractions*, которую вы, возможно, захотите прочитать, когда закончите эту книгу.
- Chotipat Pornavalai нашел ошибку в сообщении об ошибке.
- Stanislaw Antol прислал список очень ценных замечаний.
- Eric Pashman прислал множество исправлений к главам 4-11.
- Miguel Azevedo обнаружил несколько опечаток.
- Jianhua Liu прислал большой список исправлений.
- Nick King обнаружил пропущенное слово.
- Martin Zuther прислал длинный список предложений.
- Adam Zimmerman обнаружил непоследовательность с использованием слова "instanse" и несколько других опечаток.
- Ratnakar Tiwari предложил дать примечание по поводу вырожденных треугольников.
- Anurag Goel предложил другое решение для функции `is_abecederian` и прислал некоторые дополнительные исправления. Также он знает, как правильно писать имя Jane Austen (Джейн Остин).
- Kelli Kratzer нашел одну из опечаток.
- Mark Griffiths указал на запутанный пример из главы 3.
- Roydan Ongie нашел ошибку в моей реализации метода Ньютона.
- Patryk Wolowiec помог мне с одной проблемой в HTML версии.

- Mark Chonofsky поведал мне о новом ключевом слове Python 3.0.
- Russell Coleman помог мне с геометрией.
- Wei Huang обнаружил несколько типографских ошибок.
- Karen Barber указал на старую опечатку в книге.
- Nam Nguyen обнаружил опечатку, а также заметил, что я использовал один из шаблонных методов программирования, но не упомянул его названия – Decorator.
- Stéphane Morin прислала несколько исправлений и предложений.
- Paul Stoop исправил ошибку в функции `uses_only`.
- Eric Bronner указал на путаницу при обсуждении порядка выполнения вычислений.
- Alexandros Gezerlis установил новый стандарт в количестве и качестве предложений, которые он предоставил. Мы глубоко признательны!
- Gray Thomas знает разницу между right (правый) и left (левый).
- Giovanni Escobar Sosa прислал длинный список исправлений и предложений.
- Alix Etienne исправил один из URL.

От переводчика

Книга Аллена Дауни, на мой взгляд, идеально подходит для тех, у кого нет опыта в программировании вообще, а не только на языке Python. Автор начинает с самых азов, но не ограничивается ими. Он доводит читателя до довольно сложных концепций объектно-ориентированного программирования. Данная книга не является пересказом документации по языку Python. Автор ставит своей целью научить читателя образу специалиста по компьютерам, а не просто рассказать о конкретных конструкциях языка Python.

Отличительной особенностью книги является то, что автор строит последующий материал на основании предыдущего. Это является своеобразной гарантией того, что если читатель добросовесно читает книгу с самого начала и выполняет упражнения, то он будет способен дойти до самого конца и усвоить весь предлагаемый материал. Нередко в последующих главах автор возвращается к предыдущим и показывает, как можно более эффективно решить предыдущую задачу в свете новых знаний.

Необходимо помнить о том, что цель книги - научить программировать. Это не сборник готовых рецептов на все случаи жизни. Поэтому не все алготитмы, содержащиеся в книге, будут самыми оптимальными. Часто это сделано намеренно, т.к. автор ставит своей целью научить читателя тому, как придти к оптимальному решению, а не просто дает ему готовую программу.

Следует сказать несколько слов об используемой терминологии. В оригинале книги автор использует много терминов, для которых нет хороших устоявшихся эквивалентов в русском языке. В некоторых случаях разные термины используются как синонимы. Например, между *параметром* и *аргументом*, говоря строго, имееся различие, но во многих случаях оно не существенно. То же самое касается и, например, слов *expression* (выражение) и *statement* (предложение). Но если постоянно использовать слово *предложение*, то возникает устойчивая ассоциация с языкознанием.

Разумеется, замечания о найденных ошибках и неточностях перевода приветствуются. Адрес дан в информации о копирайте.

Глава 1
Как работают программы

Целью данной книги является научить вас думать как компьютерный специалист. Подобный образ мышления сочетает лучшие качества математика, инженера и ученого. Подобно математикам, компьютерные специалисты используют формальные языки для выражения идей (особенно вычислений). Подобно инженерам, они проектируют различные вещи, собирая различные компоненты в единую систему, при этом выбирая наилучшие варианты. Подобно ученым, они наблюдают за поведением сложных систем, высказывают гипотезы и испытывают предположения.

Единственно важным умением компьютерного специалиста является умение **решения задач.** Это означает, прежде всего, способность формулировать задачу, творческий подход к ее разрешению, и выражение решения ясно и точно. Таким образом, процесс изучения программирования выливается в прекрасную возможность практиковаться в умении разрешать задачи. Поэтому эта глава и называется "Как работают программы".

С одной стороны, вы будете учиться программировать, что само по себе является полезным навыком. С другой стороны, вы будете использовать программирование как конечное средство. Чем дальше мы будем продвигаться, тем более ясной будет становиться цель.

1.1 Язык программирования Python

Язык программирования, который вы будете изучать, называется Python. Python является примером **языка высокого уровня** (high-level language). Другие языки высокого уровня, о которых вы возможно слышали это C, C++, Perl и Java.

Есть и так называемые **языки низкого уровня** (low-level language), которые еще иногда называют "машинные языки" или "ассемблерные языки". Говоря упрощенно, компьютеры способны лишь на выполнение программ, написанных на языках низкого уровня. Поэтому программы, написанные на языках высокого уровня, должны пройти предварительную обработку, прежде чем они смогут выполняться компьютером. Такая дополнительная обработка занимает некоторое время, что является небольшим недостатком высокоуровневых языков.

Но их преимущества значительно превосходят все недостатки. Во-первых, гораздо легче программировать на языках высокого уровня. Написание программ на них отнимает меньше времени; они проще, и их легче читать; больше вероятность, что они написаны без ошибок, и их легче модифицировать в дальнейшем. Во-вторых, языки высокого уровня являются **переносимыми** (portable), что означает, что программы, написанные на них, могут работать на различных видах компьютеров с небольшими изменениями или вовсе без них. Низкоуровневые программы пишутся только под определенный вид компьютера, и их нужно переписывать, чтобы запустить на другом.

Ввиду этих преимуществ, практически все программы пишутся на языках высокого уровня. Низкоуровневые языки используются лишь в специализированных приложениях.

Есть два вида программ, которые переводят языки высокого уровня в языки низкого уровня: **интерпретаторы** (interpreter) и **компиляторы** (compiler)**.** Интерпретатор читает программу на языке высокого уровня и выполняет ее, что означает, что он делает то, что говорит программа. Он выполняет программу шаг за шагом, по очереди читая строки кода и выполняя соответствующие вычисления.

Компилятор читает программу и полностью переводит (компилирует) ее в низкоуровневый язык до того, как программа начнет выполняться. В этом смысле программа, написанная на языке высокого уровня называется **исходным кодом** (или исходником – source code), переведенная программа называется **объектным кодом** (object code) или **исполнимым файлом** (executable). После того, как программа скомпилирована, вы можете выполнять ее любое количество раз без повторной компиляции.

Python относится к интерпретируемым языкам, т.к. программы, написанные на нем, выполняются интерпретатором либо в **интерактивном режиме** (interactive mode), либо в **скриптовом режиме** (script mode)**.** В интерактивном режиме вы просто вводите программу, а интерпретатор выводит результат:

```
>>> 1 + 1
2
```

Шеврон >>> называется **приглашением** (prompt). Интерпретатор использует его, чтобы показать, что он готов и ожидает ввод от пользователя. Если вы напечатаете `1 + 1` (и нажмете <Enter>), интерпретатор выведет `2`.

Вы также можете хранить код в файле и использовать интерпретатор, чтобы выполнять содержимое этого файла, который в данном случае будет называться **скриптом** (script). По соглашению, имена файлов скриптов в Python заканчиваются на .py (в терминологии Windows – имеют расширение **.py**).

Чтобы выполнить скрипт, вы должны предоставить интерпретатору имя файла. В UNIX-подобных операционных системах (в том числе Mac OS X и Linux) вы должны напечатать в окне терминала слово `python`, а затем имя программы (скрипта). Допустим, ваша программа называется `prog.py`. Тогда чтобы запустить ее вы должны набрать:

```
python prog.py
```

В Windows, как правило, достаточно дважды щелкнуть по значку сохраненного скрипта для его запуска.

В других операционных системах способ выполнения скриптов может отличаться. Вы можете либо найти инструкции для вашей операционной системы, либо идите прямиком на сайт `python.org`.

Работа в интерактивном режиме удобна для отладки маленьких кусков кода, т.к. вы набираете код и тут же исполняете его. Но если ваша программа занимает уже больше, чем несколько строк, вам лучше сохранить ее в файл, чтобы вы могли потом запускать ваш код и вносить в него изменения. В интерактивном режиме вам пришлось бы повторно вводить весь код всякий раз, как только вы сделаете какие-нибудь изменения.

1.2 Что такое программа?

Программа (program) это последовательность инструкций, которая определяет порядок выполнения вычислений. Вычисления могут быть математическими, как, например, решение системы уравнений или нахождение корней многочлена, но также это могут быть и символьные вычисления, такие, как поиск и замена текста в документе или (что может показаться несколько странным) в компьютерной программе.

В разных языках детали могут отличаться, но есть несколько основных операций, которые имеются почти в каждом языке:

ввод (input): получение данных с клавиатуры, файла или другого устройства.

вывод (output): отображение данных на экране или вывод в файл или другое устройство.

математические вычисления (math): выполнение основных математических операций, как, например, сложение и умножение.

условное выполнение (conditional execution): проверка некоторых условий и выполнение кода в соответствии с ними.

повторение (repetition): выполнение некоторого действия снова и снова, обычно, с некоторыми изменениями.

Хотите – верьте, хотите – нет, но всего вышесказанного достаточно для написания любой программы. Любая программа, которую вы когда-либо использовали, не важно, насколько она сложна, состоит из инструкций, подобных вышеупомянутым. Поэтому вы можете представлять себе программирование как процесс разбиения большой и сложной задачи на меньшие и еще меньшие подзадачи, пока эти подзадачи не будут достаточно просты, чтобы их можно было бы выразить в виде одной из этих основных инструкций.

Возможно, все это выглядит несколько запутанно, но мы еще вернемся к данной теме позднее, когда будем говорить об **алгоритмах** (algorithm).

1.3 Что такое отладка?

При программировании мы часто допускаем ошибки. Исторически сложилось так, что ошибки в программах называются **багами** (bug), а процесс их нахождения называется **отладкой** (debugging).

Есть три вида ошибок, которые могут быть в программе: синтаксические ошибки, ошибки при выполнении и семантические ошибки. Необходимо различать их, чтобы процесс отладки шел быстрее.

1.3.1 Синтаксические ошибки (syntax errors)

Python может выполнять программы лишь в том случае, если их синтаксис правильный. В противном случае интерпретатор выдает сообщение об ошибке. **Синтаксис** (syntax) относится к структуре программы и к правилам этой структуры. Например, скобки должны быть парными, поэтому выражение `(1 + 2)` является допустимым, а, например, `8)` будет **синтаксической ошибкой** (syntax error).

Когда вы читаете, например, на английском языке, вы можете мириться со многими синтаксическими ошибками. Именно поэтому мы можем читать, например, поэзию автора E.E. Cumming[1] без особых затруднений. Python же не прощает подобных вольностей. Если он найдет хотя бы одну синтаксическую ошибку в программе, он тут же выдаст сообщение о найденной ошибке и завершит свою работу. Вы не сможете выполнять программу, в которой есть синтаксические ошибки. В течение нескольких первых недель вашей карьеры программиста вы, вероятно, будете тратить много времени на выявление синтаксических ошибок. С приобретением опыта вы будете делать их меньше, а находить их быстрее.

1.3.2 Ошибки при выполнении (runtime errors)

Второй тип ошибок это ошибки при выполнении. Они так называются потому, что не проявляются до тех пор, пока программа не начнет выполнение. Такие ошибки еще называют **исключениями** (exception), т.к. они обычно указывают на то, что произошло нечто исключительное (и плохое).

Ошибки при выполнении довольно редко встречаются в маленьких программах, с которыми вы познакомитесь в нескольких первых главах. Поэтому может пройти значительное время, пока вы совершите одну из них.

1.3.3 Семантические ошибки (semantic errors)

Третий тип ошибок называется **семантическими (смысловыми) ошибками.** Если в вашей программе есть семантическая ошибка, то программа все равно успешно выполнится в том смысле, что компьютер не выдаст никаких сообщений об ошибках, но программа будет делать не то, что вы от нее ожидаете. Она будет делать нечто совсем другое. Конкретно, она будет делать именно то, что вы сказали ей делать.

---

[1] Американский поэт и романист. Его поэмы отличаются эксперементальным оформлением, самое заметное из которых – отстутствие заглавных букв - *прим. пер.*

Проблема заключается в том, что программа, которую вы написали, не является той программой, которую вы намеревались написать. Значение программы (ее семантика) неправильное. Выявление семантических ошибок может оказаться непростым делом, т.к. вам придется возвращаться назад, проверяя вывод программы и пытаясь понять, что она делает *на самом деле*

1.3.4 Экспериментальная отладка

Отладка это один из наиболее важных навыков, который вы приобретете. Хотя она и может вас совсем утомить, тем не менее, отладка является одной из наиболее интеллектуальных, вызывающих и интереснейших частей программирования.

В некотором роде, отладка подобна работе сыщика. У вас есть улики и свидетельства, и вам необходимо на их основе сделать выводы, объясняющие тот результат, который вы видите.

Еще отладка подобна экспериментальной науке. Как только вы поняли, в чем заключается ошибка, вы модифицируете вашу программу и испытываете ее снова. Если ваша гипотеза была верной, то вы можете предсказать результат, который произведет ваша модифицированная программа. Вы приблизитесь на один шаг ближе к работающей программе. Если же ваша гипотеза была неверной, то вам необходимо будет выдвинуть новую. Как говорил Шерлок Холмс: "Когда вы отбросите все невозможное, то то, что осталось, и будет истиной, какой бы невероятной она вам ни казалась" (Конан Дойль, *Знак четырех*).

Для некоторых людей программирование и отладка являются одним и тем же. Т.е. программирование это процесс постепенной отладки программы, пока она не станет тем, что вы от нее ожидаете. Идея заключается в том, что вы должны начать с программы, которая делает *что-нибудь*, и делать небольшие изменения, отлаживая их по ходу дела, так что вы всегда имеете дело с работающей программой.

Например, Linux это операционная система, которая содержит тысячи строк кода, но она началась с небольшой программы, которую использовал Линус Торвальдс для изучения компьютера с процессором Intel 80386. Согласно Ларри Гринфилду, "Одним из ранних проектов Линуса была программа, которая переключалась между печатанием AAAA и BBBB. Потом эта программа превратилась в Linux." (*The Linux Users' Guide* Beta Version 1).

В последующих главах мы дадим больше рекомендаций по отладке и другим методам работы в программировании.

1.4 Формальные и естественные языки

Естественные языки (natural languages) это те языки, на которых говорят люди, например, английский, испанский, французский. Они не были созданы людьми (хотя люди и пытаются навязывать им некоторый порядок); они развивались естественным образом.

Формальные языки (formal languages) это языки, используемые людьми в особых случаях. Хорошим примером может служить язык математических формул, который хорошо подходит для описания взаимоотношений чисел и символов. Химики тоже используют формальный язык для представления химической структуры молекул. И, наиболее важное:

> **Языки программирования являются формальными языками, которые были созданы для того, чтобы ими записывать вычисления.**

Формальные языки тяготеют к тому, чтобы ограничить синтаксические правила. Например, 3+3=6 является синтаксически правильным математическим выражением, а 3+=3$6 таковым не является. Также и H_2O является синтаксически правильной химической формулой, а $_2$Zz – нет.

Синтаксические правила разделяются на две категории, относящиеся к **лексемам** (token) и к структуре. Лексемы являются основными элементами языка, такими как слово, числа и химические элементы. Проблема с 3+=3$6 заключается в том, что $ не является разрешенной лексемой в математике (по крайней мере, насколько мне это известно). Подобным образом и $_2$Zz не является разрешенным выражением, т.к. нет элемента с обозначением Zz.

Второй тип синтаксических ошибок относится к структуре выражения, т.е. к способу, которым лексемы упорядочены. Выражение 3+=3$6 является некорректным, т.к. хотя + и = допустимые

лексемы, они не могут следовать непосредственно одна за другой. Также и в химических формулах число, обозначающее количество атомов, должно писаться после имени элемента, а не до.

Упражнение 1.1. Напишите хорошо структурированное предложение на русском языке, но с недопустимыми лексемами. Затем напишите другое предложение с правильными лексемами, но с неправильной структурой.

Когда вы читаете предложение на естественном языке или выражение на формальном, то вы должны выяснить структуру этого предложения (хотя в случае с естественными языками вы делаете это подсознательно). Этот процесс называется **парсингом** (parsing), т.е. разбором по частям.

Например, когда вы слышите предложение "the penny dropped" (копейка упала), вы понимаете, что "the penny" является подлежащим (или субъектом), а "dropped" является сказуемым (или предикатом). Как только вы разобрались с синтаксисом данного предложения, вы можете понять, что оно означает, т.е. вы можете выявить семантику данного предложения. Подразумевая, что вам известно, что такое "penny", и что означает "to drop", вы поймете смысл данного высказывания.

Хотя между формальными и естественными языками есть много общего – лексемы, структура, синтаксис и семантика, – имеются также и различия:

двусмысленность (ambiguity): естественные языки полны двусмысленностей, которые разрешаются исходя из контекста и другой информации. Формальные языки создаются таким образом, чтобы не иметь двусмысленностей вовсе или свести их к минимуму, что означает, что любое выражение имеет только одно значение, вне зависимости от контекста.

избыточность (redundancy): чтобы решить проблему двусмысленности и уменьшить непонимание, естественные языки используют много избыточной информации. В результате, они часто многословны. Формальные языки менее избыточны и более лаконичны.

буквальность (literalness): естественные языки полны идиом и метафор. Если я скажу "the penny dropped", то, возможно, что тут на самом деле речь не идет ни о копейке, ни о каком бы то ни было падении[2]. Формальные языки всегда означают то, что в них говорится.

Людям, которые вырастают, говоря на естественных языках – т.е. всем нам, – часто бывает очень трудно приспособиться к формальным языкам. В некотором отношении разница между формальным и естественным языком такая же, как и между поэзией и прозой. Даже более того:

Поэзия: слова используются исходя не только из их значения, но и из их звучания. В результате целое стихотворение создает определенный эффект или эмоциональный отклик. Двусмысленность не только встречается, но и создается сознательно.

Проза: буквальное значение слов более важно, структура служит к лучшему пониманию. Проза более доступна для анализа, но все еще часто двусмысленна.

Программы: значение компьютерных программ недвусмысленно и буквально. Его можно полностью понять, анализируя лексемы и структуру.

Вот несколько советов по чтению программ (и других формальных языков). Прежде всего помните, что формальные языки гораздо более сжаты, чем естественные, поэтому для того, чтобы человеку их прочитать и понять, нужно больше времени. Также и структура очень важна, так что чтение сверху вниз и слева направо не всегда помогает. Вместо этого учитесь разбивать программу на части в своей голове, определяя все лексемы и интерпретируя структуру. Наконец, детали имеют значение. Незначительные орфографические и пунктуационные ошибки, которые вы можете без особых последствий допускать в естественных языках, могут быть критическими в формальном языке.

1.5 Первая программа

Традиционно первая программа, которую вы пишите на новом языке, называется "Hello, World!" ("Привет, мир!"), т.к. она выводит на экран надпись "Hello, World!". На Python такая программа выглядит следующим образом:

[2] Идиома "the penny dropped" означает, что кто-то внезапно осознал ситуацию - *прим. пер*.

```
print('Hello, World!')
```

Это является примером **функции `print`**, которая на самом деле ничего не печатает на бумаге. Она отображает результат на экране. В нашем случае результатом будут слова

```
Hello, World!
```

Знаки кавычек в этой программе служат для обозначения начала и конца отображаемого текста; сами они на экран не выводятся.

Некоторые люди судят о качестве языка программирования по тому, насколько просто в нем создать программу "Hello, World!". По этому критерию Python занимает довольно высокую оценку.

1.6 Отладка

Рекомендуется, чтобы вы читали эту книгу перед включенным компьютером, чтобы вы могли сами испытывать те программы, о которых читаете. Большинство примеров вы можете выполнить в интерактивном режиме, но если вы создадите из вашего кода скрипт, то вам будет проще его изменять и испытывать.

Когда вы изучаете новое вам свойство языка, вы должны попробовать сделать ошибки. Например, что произойдет в программе "Hello, World!", если вы не поставите замыкающую кавычку? Что, если вы не поставите обе? Что будет, если вы введете имя функции `print` неправильно?

Такие эксперименты позволят вам лучше усвоить то, о чем вы читаете. Это также помогает и с отладкой, т.к. вы узнаете, в каких случаях появляются те или иные сообщения об ошибках. Лучше делать ошибки сейчас и намеренно, чем после и случайно.

Программирование и, особенно, отладка иногда способны вызывать сильные эмоции. Если вы пытаетесь отыскать трудноуловимую ошибку, то вы можете испытывать злость, подавленность и замешательство.

Есть свидетельства того, что люди обращаются с компьютерами так, как если бы они были бы людьми. Когда они работают хорошо, мы думаем о них как о наших товарищах, но если же они упрямы и недружелюбны, мы обращаемся с ними так же, как мы обращаемся и с упрямыми и недружелюбными людьми.

Если вы будете готовы к подобным проявлениям, то это только поможет вам в дальнейшем. Один из советов заключается в том, чтобы думать о них как о наемных служащих с некоторыми сильными сторонами, такими как скорость и точность, но в то же время и с некоторыми слабостями, в частности, с полным отсутствием сопереживания и неспособностью видеть картину целиком.

Ваша работа заключается в том, чтобы быть хорошим менеджером: найдите способы, чтобы воспользоваться их сильными сторонами и свести к минимуму влияние слабых сторон. Также ищите способы, чтобы использовать ваши эмоции во благо для решения проблемы, и не позволяйте вашим чувствам мешать вам работать эффективно.

Хоть обучение процессу отладки и может ввести вас в расстройство, но оно является ценным умением, которое может пригодиться вам и в других областях, помимо программирования. В конце каждой главы находится раздел, посвященный отладке, подобный этому, который вы сейчас читаете, в котором содержатся мои мысли по поводу отладки. Надеюсь, они вам помогут!

1.7 Словарь терминов

решение задач (problem solving): процесс формулирования задачи, нахождение решения и выражение этого решения.

язык высокого уровня (high-level laguage): язык программирования, подобный Python, который разработан для того, чтобы человек мог легко его понимать.

язык низкого уровня (low-level language): язык программирования, разработанный для того, чтобы компьютер мог его легко понимать и выполнять; также называется "машинным" или "ассемблерным" языком.

переносимость (portability): свойство программы, которая может работать на разных видах компьютеров.

интерпретировать (interpret): переводить программу, написанную на языке высокого уровня, в машинный код строка за строкой, одновременно выполняя переведенный строки.

компилировать (compile): полностью перевести всю программу, написанную на языке высокого уровня, на язык низкого уровня, перед тем, как начать ее выполнение.

исходный код (source code): программа на языке высокого уровня перед ее компилированием.

объектный код (object code): результат работы компилятора после перевода программы.

исполнимый файл (executable): другое название объектного кода, готового к выполнению.

приглашение (prompt): символы, отображаемые интерпретатором, чтобы показать, что он готов принимать ввод от пользователя.

скрипт (script): программа, хранящаяся в файле (обычно, речь идет об интерпретируемой программе).

интерактивный режим (interactive mode): способ использования интерпретатора Python для ввода команд и выражений, когда выводится приглашение интерпретатора.

скриптовый режим (script mode): способ использования интерпретатора Python для чтения и выполнения выражений, хранящихся в файлах – скриптах.

программа (program): набор инструкций, определяющих вычисления.

алгоритм (algorithm): общий процесс для решения категории проблем.

баг (bug): ошибка в программе.

отладка (debugging): процесс нахождения и удаления трех видов программных ошибок.

синтаксис (syntax): структура программы.

синтаксическая ошибка (syntax error): ошибка в программе, делающая невозможным парсинг (и, соответственно, ее интерпретацию или компиляцию).

исключение (exception): ошибка, проявляющаяся во время работы программы.

семантика (semantics): значение программы.

семантическая ошибка (semantic error): ошибка в программе, заставляющая ее делать нечто иное, нежели то, что намеревался программист.

естественный язык (natural language): любой из языков, на котором говорят люди, который развивался естественным образом.

формальный язык (formal language): любой из языков, изобретенных людьми для специального применения, такой, как, например, выражение математических идей или компьютерных программ; все языки программирования являются естественными языками.

лексема или **токен** (token): один из основных элементов синтаксической структуры программы, аналогичный слову в естественном языке.

парсинг (parsing): исследование программы и анализ ее синтаксической структуры.

1.8 Упражнения

Упражнение 1.2 Зайдите на сайт `python.org`. Этот сайт содержит информацию о языке Python и ссылки на другие страницы, дающие вам возможность осуществлять поиск по документации.

Например, если вы введете слово `print` в окно поиска, первая появившаяся ссылка будет указывать на документацию функции `print`. На данной стадии вам не все будет понятно, но, как минимум, вы будете знать, где искать информацию в дальнейшем.

Упражнение 1.3 Запустите интерпретатор Python и наберите `help()` для запуска интерактивной помощи. Также вы можете напечатать, например, `help(print)` для получения информации по функции `print`. Когда закончите чтение, введите `quit` в приглашение, чтобы вернуться в интерпретатор.

Если это у вас не работает, то вероятнее всего, что вам нужно будет установить дополнительную документацию по Python или установить переменные окружения. Детали зависят от вашей операционной системы и версии Python.

Упражнение 1.3 Запустите интерпретатор Python и используйте его как калькулятор. Его синтаксис для математических выражений почти такой же, как и в обычной математической записи. Например, символы +, – и / означают сложение, вычитание и деление. Символом умножения является *.

Если вы пробежите 10-км дистанцию за 43 минуты и 30 секунд, каково ваше среднее время в перерасчете на мили? Какова ваша средняя скорость в миль/час? (Подсказка: одна миля содержит 1.61 км).

Глава 2
Переменные, выражения и предложения

2.1 Значения и типы

Значение или **величина** (value) является одной из основных вещей, с которыми работает программа, также, как и, например, буква или число. Значения, с которыми мы до сих пор встречались, были 1, 2 и "Hello, World!".

Эти значения принадлежат к различным **типам** (type)**:** 2 является числовым значением целого типа (integer), а "Hello, World!" это **строка** (string). Вы (и интерпретатор) можете легко распознать строку, т.к. она заключена в двойные кавычки.

Функция `print` также пригодна и для целых чисел:

```
>>> print(4)
4
```

Если вы не уверены в том, какой тип имеет ваша величина, вам может помочь сам интерпретатор:

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Строки относятся к типу `str`, целые числа к типу `int`, а числа с десятичной точкой относятся к типу `float`, т.к. такие числа предоставлены в формате с **плавающей точкой** (floating poing).

```
>>> type(3.2)
<class 'float'>
```

А как насчет таких значений, как, например, `'17' и '3.2'`?

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Это строковый тип.

Когда вы вводите большое целое число, то у вас может возникнуть искушение использовать запятые для разделения всего числа на группы по три цифры[3], например, 1,000,000. В Python так недопустимо вводить целые числа, но, тем не менее, Python понимает такую конструкцию как нечто иное:

```
>>> print(1,000,000)
1 0 0
```

Это не совсем то, что мы ожидали увидеть! Python воспринимает 1,000,000 как последовательность целых чисел, разделенных запятой и выводит их соответственно, разделяя их пробелами.

Это первый раз, когда мы встречаемся с семантической ошибкой. Программа благополучно выполняется без каких бы то ни было предупреждений об ошибках, но она не делает то, что мы от нее ожидали.

2.2 Переменные

Одной из наиболее мощных возможностей, которую предоставляет программирование, является возможность манипулировать **переменными** (variable)**.** Переменная это имя, которое ссылается на некоторое значение.

Операция присваивания (assingment statement) создает новые переменные и дает им значения:

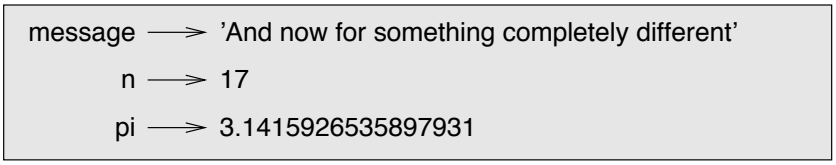
---

[3] Так принято писать в западной системе образования - *прим. пер.*

```
>>> message = 'And now for something completely different.'
>>> n = 17
>>> pi = 3.1415926535897931
```

Мы видим примеры трех присваиваний. В первом примере строковое значение присваивается переменной `message`; во втором примере переменной `n` присваивается значение `17`; третьей переменной `pi` присваивается приблизительное значение числа π.

Для изображения операции присваивания на бумаге обычно пишут имя переменной и затем стрелку, указывающую на присваиваемое значение. Такое обозначение называется **диаграммой состояния** (state diagram), т.к. она показывает, каково состояние каждой переменной. Следующая диаграмма показывает результат предыдущего примера:



Чтобы отобразить значение переменной, вы можете воспользоваться функцией `print`:

```
>>> print(n)
17
>>> print(pi)
3.14159265359
```

Переменная имеет тот же самый тип, что и содержащееся в ней значение:

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

Упражнение 2.1 Выполните следующее:

```
>>> m = 1,000,000
>>> type(m)
```

Теперь обратите внимание на то, что переменная `m` не является переменной целого типа.

2.3 Имена переменных и ключевые слова

Обычно программисты выбирают для своих переменных имена со смыслом – из этого ясно, для чего они предназначены.

Имена переменных могут быть сравнительно длинными. Они могут состоять из букв и цифр, но они должны начинаться с буквы. Можно использовать буквы латинского алфавита как `ВЕРХНЕГО`, так и `нижнего` регистра, но лучше, все же, начинать имена переменных с нижнего регистра (позже вы увидите почему). В именах переменных можно использовать знак подчеркивания `_`. Довольно часто используются имена, состоящие из нескольких слов, такие как `my_name` или `airspeed_of_unladen_swallow`.

Если вы дадите переменной некорректное имя, вы получите сообщение об ошибке:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` неправильно, т.к. не начинается с буквы. `more@` неправильно, т.к. содержит недопустимый в имени переменной символ `@`. Но почему имя `class` также выдает ошибку?

Дело в том, что `class` является одним из **ключевых** или **зарезервированных слов** в Python. Интерпретатор использует ключевые слова, чтобы определить структуру программы. Поэтому их нельзя использовать в качестве имен переменных. В Python имеется 30 ключевых слов:

```
and         elif        import      raise
as          else        in          return
assert      except      is          try
break       finally     lambda      while
class       for         nonlocal    with
continue    from        not         yield
def         global      or
del         if          pass
```

Возможно, вам следует всегда держать этот список у себя под рукой. Если интерпретатор выдаст ошибку по поводу имени одной из ваших переменных, и вы не будете знать почему, попробуйте посмотреть в этот список. Кстати, вы всегда можете посмотреть эти ключевые слова, отдав команду интерпретатору:

```
>>> help('keywords')
```

2.4 Предложения

Предложение (statement) это такой кусочек кода, который интерпретатор Python может выполнить. До сих пор мы видели два вида предложений: вызов функции `print` и операция присваивания.

Если вы вводите предложение в интерактивном режиме, интерпретатор выполняет его и выводит результат, если таковой имеется.

Скрипт же, как правило, состоит из нескольких предложений. Если есть больше одного предложения, то результаты выводятся один за одним по мере выполнения этих предложений. Помните, что скрипты хранятся в вашей файловой системе, и способ их запуска на выполнение зависит от используемой операционной системы. Инструкции касательно вашей системы вы можете найти на сайте `python.org` или спросить более опытного программиста.

Например, следующий скрипт

```
print(1)
x=2
print(x)
```

выведет на экран

```
1
2
```

Операция присваивания на экране не отобразится.

2.5 Операторы и операнды

Операторы (operator) это специальные символы, которые обозначают различные вычислительные действия, например, сложение или умножение. Те значения, к которым применяются операторы, называются **операнды** (operand).

Следующие операторы (без запятой): `+`, `-`, `*`, `/`, `**` производят сложение, вычитание, умножение, деление и возведение в степень соответственно, как в следующих примерах:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

В некоторых других языках для возведения в степень используется `^`, но в Python этот оператор используется для операции побитного сравнения, называемой XOR. В этой книге я не буду вдаваться в

подробности о побитных операциях, но вы можете прочитать о них в Википедии `wiki.python.org/moin/BitwiseOperators`

Есть также и специальный оператор деления `//`, который действует как и обычный оператор деления, только этот отбрасывает дробную часть (справа от запятой):

```
>>> minute = 133
>>> minute//60
2
```

Если хотя бы один из участвующих операндов имеет тип `float`, Python конвертирует все остальные числа в `float` и возвращает результат тоже `float`.

2.6 Выражения

Выражение[4] (expression) это комбинация значений, переменных и операторов. Значение само по себе тоже считается выражением. То же самое касается и переменных. Поэтому ниже приведены допустимые с точки зрения Python выражения (предполагая, что переменной `x` было присвоено заранее какое-то значение):

```
17
x
x+17
```

Если вы вводите выражение в интерактивном режиме, то интерпретатор **вычисляет** (evaluate) и отображает результат:

```
>>> 1+1
2
```

Но в скриптах само по себе выражение не делает ничего! Новички часто приводятся этим в замешательство.

Упражнение 2.2 Введите следующие выражения в интерпретатор Python и посмотрите, что они делают:

```
5
x=5
x+1
```

Теперь запишите те же самые выражения в скрипт и запустите его. Что вы видите на экране? Измените каждое выражение так, чтобы превратить его в вызов функции `print`, и запустите скрипт снова.

2.7 Порядок вычислений

Когда выражение состоит из более чем одного оператора, тогда порядок производимых вычислений определяется **приоритетом операций** (rules of precedence)**.** При совершении математических вычислений Python следует правилам, принятым в математике.

- **Скобки** имеют наивысший приоритет. Их используют для того, чтобы вычисления производились в том порядке, в котором нам нужно. Поэтому `2*(3-1)` равняется `4`, а `(1+1)**(5-2)` равняется `8`. Вы также можете пользоваться скобками и для того, чтобы ваше выражение было легче читать, как, например, `(minute*100)/60`, хотя они в данном случае и не влияют на порядок вычислений.
- Возведение в **степень** имеет следующий приоритет, поэтому `2**1+1` равняется `3`, не `4`, а `3*1**3` будет `3`, не `27`**.**
- Следующий приоритет за **умножением** и **делением**, имеющие одинаковый приоритет, который выше приоритета **сложения** и **вычитания**, которые, в свою очередь, имеют между собой

[4] Далее в последующих главах в русской версии слово "предложение" (statement) будет часто переводиться словом "выражение" (expression) - *прим. пер.*

одинаковый приоритет. Поэтому `2*3-1` будет `5`, не `4`, а `6+4/2` будет `8`, а не `5`.

- Операторы с одинаковым приоритетом вычисляются слева направо. Поэтому в выражении `degrees/2*pi` сначала происходит деление, а затем выражение умножается на `pi`. Если же `degrees` нужно разделить на 2π, то необходимо использовать скобки: `degrees / (2 * pi)`, либо использовать оператор деления повторно: `degrees/2/pi`.

2.8 Строковые операции

В общем случае вы не можете выполнять математические операции над строками, даже если они содержат числа. Поэтому следующие выражения недопустимы в Python:

```
'2'-'1' 'eggs'/'easy' 'third'*'a charm'
```

Оператор + работает со строками, но не так, как вы, возможно, ожидали. Он производит **сцепление** (concatenation), т.е. соединение их концами, например:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> print(first + second)
throatwarbler
```

Оператор * тоже работает со строками. Он производит повторение. Например:

```
>>> 'Spam' * 3
SpamSpamSpam
```

Поэтому если один из операндов строка, то другой должен быть целым числом.

Использование + и * аналогично использованию сложения и умножения. Также как и `4*3` эквивалентно `4+4+4`, мы вправе ожидать, что `'Spam'*3` будет эквивалентно `'Spam'+'Spam'+'Spam'`, и так оно и есть. С другой стороны, есть значительное отличие между сцеплением и повторением строк и сложением и умножением целых чисел. Можете ли вы указать на свойство, каким обладает сложение целых чисел, которого нет у сцепления строк?

2.9 Комментарии

Чем длиннее и сложнее становятся программы, тем труднее становится их читать. Формальные языки очень лаконичны, поэтому часто бывает нелегко с первого взгляда на код понять, что он делает и почему.

Поэтому рекомендуется добавлять в вашу программу пояснения на естественном языке, которые будут объяснять, что делает ваша программа в данном случае. Такие примечания называются **комментариями** (comment), и они начинаются с символа #:

```
# вычисление процентного отношения прошедшего времени
percentage = (minute * 100) / 60
```

В примере выше комментарии расположены на отдельной строке. Но вы можете также добавлять их в и конце строки:

```
percentage = (minute * 100) / 60 # проценты
```

Все, что находится на строке правее символа #, игнорируется интерпретатором и не оказывает никакого влияния на программу.

Комментарии полезны, когда они объясняют не совсем очевидные фрагменты кода. Разумно будет предположить, что читатель сам может определить, *что* делает код; гораздо более полезно объяснить, *почему* он это делает.

Избыточные комментарии бесполезны:

```
v = 5 # присвоить v значение 5
```

А этот комментарий содержит полезную информацию, которой нет в коде:

```
v = 5 # скорость в м/с
```

Удачно выбранные имена переменных могут сократить нужду в комментариях, однако, слишком длинные имена затрудняют восприятие. Поэтому здесь необходим компромисс.

2.10 Отладка

Если вы дошли до этого места, то синтаксические ошибки, которые вы допускаете, вероятнее всего вызваны неправильными именами переменных, как, например, `class` или `yield`, которые являются зарезервированными словами, или `odd~job` или `US$`, содержащими недопустимые символы.

Если вы запишите имя переменной с пробелом, то Python подумает, что это два операнда без оператора между ними:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

Сообщения о синтаксических ошибках не слишком информативны. Чаще всего это `SyntaxError: invalid syntax` и `SyntaxError: invalid token`

Если вы допускаете ошибки, обнаруживаемые во время выполнения, то чаще всего это будут ошибки использования переменной, которая еще не была создана.

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Здесь вы просто ошиблись при печатании переменной `principal`. Обратите внимание, что имена переменных чувствительны к регистру, поэтому `LaTeX` не то же самое, что `latex`.

И наиболее частой семантической ошибкой на этой стадии, вероятно, является порядок вычислений. Например, для вычисления 1/2π вы можете написать следующий код:

```
>>> pi = 3.1415926
>>> 1.0 / 2.0 * pi
```

Но сначала будет произведено деление, поэтому в итоге вы получите `0.5 *` π, что совсем не то, что вам нужно. Python не может угадать ваши намерения, поэтому сообщения об ошибке вы не получите. Вы просто получите неправильный ответ.

2.11 Словарь терминов

значение или **величина** (value): одна из основных порций данных, таких как число или строка, которыми манипулирует программа.

тип (type): категория значения. Типы, с которыми мы встречались до сих пор, были целый тип (`int`), числа с плавающей точкой (`float`) и строковый тип (`str`).

целое число (integer): тип, представляющий целые числа, как, например, `30` или `-5`.

числа с плавающей точкой (floating-point): тип, представляющий числа с дробной частью, например, `3.1415926` или `-0.0056`.

строка (string): тип, представляющий собой последовательность символов.

переменная (variable): имя, которое ссылается на некое значение.

предложение (statement): часть кода, которая представляет собой команду или действие. До сих пор мы видели только операции присваивания и вызов функции `print`.

присваивание (assignment): предложение, которое присваивает переменной некоторое значение.

диаграмма состояния (state diagram): графическое представление набора переменных и значений, на которые они ссылаются.

ключевое или **зарезервированное слово** (keyword): слово, используемое компилятором для разбора программы по частям; вы не можете использовать такие слова (как, например, `if`, `def` или `while`) в качестве имен переменных.

оператор (operator): специальный символ, представляющий собой простое вычисление, как, например, сложение, умножение или сцепление строк.

операнд (operand): одно из значений, с которым работает оператор.

деление с отбрасыванием дробной части (floor division): операция деления, отбрасывающая дробную часть (без округления).

выражение (expression): комбинация переменных, операторов и значений, которые, в итоге, представляют собой единое значение.

вычислить (evaluate): упростить выражение, совершая операции в установленном порядке, чтобы, в итоге, получилось единое значение.

приоритет операций (rules of precedence): набор правил, устанавливающий порядок выполнения вычислений в выражении.

сцепление (concatenation): соединение двух операндов концами.

комментарий (comment): информация в программе, предназначенная для других программистов (или для себя самого), которая не влияет на выполнение программы.

2.12 Упражнения

Упражнение 2.3 Представьте, что мы выполняем следующие операции присваивания:

```
width = 17        # ширина
height = 12.0     # высота
delimiter = '.'   # разделитель
```

Для каждого из следующих выражений определите его значение и тип этого значения:

```
1. width/2
2. width/2.0
3. height/3
4. 1 + 2 * 5
5. delimiter * 5
```

Используйте интерпретатор Python и функцию `type` для проверки ваших ответов.

Упражнение 2.4 Попрактикуйтесь в использовании интерпретатора Python в качестве калькулятора:

1. Объем сферы с радиусом `R` равен `4/3*(pi*R**3)`. Вычислите объем сферы радиусом `5`. Подсказка: вначале определите переменную `pi = 3.1415926`. Также имейте в виду, что ответ `392.6` неверный!
2. Стоимость одной книги $24.95, но магазин дает скидку 40%. Доставка стоит $3 за первую книгу и по $0.75 за каждую последующую. Какова будет цена покупки 60 книг с доставкой?

Если я выйду из дома в 6:52 утра, пробегу 1 милю не спеша (8 мин 15 сек на милю), затем 3 мили в более быстром темпе (7 мин 12 сек на милю), затем 1 милю снова не спеша, то в какое время я вернусь домой к завтраку.

Глава 3
Функции

3.1 Вызов функций

В программировании функцией называется именованная последовательность выражений, которые осуществляют некоторые вычисления. Когда вы определяете функцию, вы даете ей имя и последовательность выражений. Позже вы можете "вызвать" эту функцию по ее имени. Мы уже встречались с примером **вызова функции** (function call):

```
>>> type(32)
<class 'int'>
```

Имя этой функции – `type`. Выражение в скобках называется **аргументом** (argument) функции. Результатом этой функции является тип ее аргумента.

Часто говорят, что функция "берет" или "принимает" некий аргумент и "возвращает" результат. Такой результат называется **возвращаемым значением** (return value).

3.2 Функции, конвертирующие типы

В Python имеются встроенные (built-in) функции, которые конвертируют данные одного типа в другой тип. Функция `int` берет любое значение и конвертирует его в целочисленный тип; если же это невозможно, то она выводит сообщение об ошибке:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

Эта функция может также конвертировать дробные числа в целые, но при этом эти числа не округляются, просто отбрасывается их дробная часть:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Функция `float` конвертирует целые числа и строки в числа с плавающей точкой:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Наконец, функция `str` конвертирует свой аргумент в строку:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.3 Математические функции

В Python имеется модуль `math`, содержащий наиболее употребительные математические функции. **Модулем** (module) называется файл, содержащий набор схожих функций.

Чтобы можно было использовать функции из определенного модуля, его необходимо вначале импортировать в программу:

```
>>> import math
```

Эта команда создает **объект типа `module`** с именем `math`. Если вы примените функцию `print` к этому объекту, то сможете получить о нем некоторую информацию:

```
>>> print(math)
<module 'math' (built-in)>
```

Этот модульный объект содержит функции и переменные, определенные в этом модуле. Для доступа к любой функции этого модуля вам необходимо задать имя модуля и имя этой функции, разделенными точкой. Такой формат называется **точечная запись** (dot notation).

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

В первом случае вычисляется логарифм по основанию 10 для отношения сигнал/шум. В модуле `math` также имеется и функция `log` для вычисления логарифма по основанию `e`.

Во втором примере вычисляется синус угла, выраженного в радианах. Мы выбрали имена переменных так, чтобы они служили нам подсказкой тому, что `sin` и другие тригонометрические функции (`cos`, `tan` и т.д.) принимают аргумент в радианах. Чтобы перевести градусы в радианы, разделите градусы на 360 и умножьте на 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865475
```

Выражение `math.pi` получает значение `pi`, определенной в модуле `math`, которое соответствует значению π, округленного до 15 знаков.

Если вы еще не забыли ваш курс тригонометрии, то вы можете проверить предыдущий результат, сравнив его с квадратным корнем из двух, разделенного на два:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

3.4 Композиция

До сих пор мы видели различные элементы программы – переменные, выражения и предложения – сами по себе. Мы еще ничего не говорили о том, как можно их комбинировать.

Одним из наиболее сильных свойств программирования является способность брать маленькие блоки кода и комбинировать их. Это называется **композицией** (composition)**.** Например, аргументом функции может быть любое выражение, включая арифметические операторы:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

И даже вызов другой функции:

```
x = math.exp(math.log(x+1))
```

Почти везде, где вы могли бы подставить одиночное значение, вы можете подставить, практически, любое выражение, с одним только исключением: с левой стороны операции присваивания должна находится переменная. Любое другое выражение с левой стороны вызовет синтаксическую ошибку[5]:

```
>>> minutes = hours * 60 # правильно
>>> hours * 60 = minutes # неправильно!
SyntaxError: can't assign to operator
```

[5] Позже мы познакомимся с исключениями из этого правила.

3.5 Добавление новых функций

До сих пор мы пользовались только теми функциями, которые поставляются вместе с Python, однако имеется также и возможность добавлять новые функции. Чтобы создать, или как еще говорят, **определить функцию** (function definition), необходимо задать ей имя и порядок действий, которые будут выполняться при ее вызове. Пример:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` это ключевое слово, означающее определение новой функции. Имя этой новой функции `print_lyrics`. Правила для задания имен функций такие же, как и для имен переменных: буквы, числа и некоторые знаки пунктуации, но первым символом должна быть буква. Вы также не можете использовать ключевые слова для имен функций. Также вы не должны создавать функции и переменные с одним и тем же именем.

Пустые скобки после имени функции говорят о том, что она не принимает никаких аргументов.

Первая строчка определения функции называется **заголовком** (header); все остальное называется ее **телом** (body). Заголовок должен заканчиваться двоеточием; тело функции должно иметь отступ. По соглашению, отступ равняется 4-ем пробелам (см. раздел 3.13). Тело может содержать любое количество предложений.

Строки в теле нашей функции заключены в двойные кавычки (`"`). Можно также использовать и одинарные кавычки (`'`), результат будет тем же. Большинство людей используют одинарные кавычки (которые также называются *апострофом*), за исключением случаев, когда сам апостроф используется в строке.

Чтобы завершить ввод функции в интерпретаторе, вам необходимо ввести пустую линию (т.е. нажать <Enter>). В скриптах это делать не обязательно.

Когда вы определяете функцию, то одновременно с этим создается и переменная с точно таким же именем:

```
>>> print (print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

Значением `print_lyrics` является объект `function`, который имеет тип `class 'function'`.

Синтаксис для вызова новых функций такой же, что и для встроенных функций:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Как только вы определили новую функцию, вы можете использовать ее внутри другой функции. Например, чтобы повторить куплет из предыдущей песни, вы можете написать функцию `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

После этого можно вызвать `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Хотя в настоящей песне порядок слов другой.

3.6 Определение и использование

Объединив фрагменты кода из предыдущего раздела, наша программа примет следующий вид:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print "I sleep all night and I work all day."

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

В этой программе определяются две функции: `print_lyrics` и `repeat_lyrics`. Определения функций исполняются точно так же, как и другие предложения, но в их случае создается объект `function`. Предложения внутри функции не выполняются до тех пор, пока функция не будет вызвана. Само же по себе определение функции не производит никакого результата.

Как вы, возможно, и ожидали, вы должны создать функцию прежде, чем ее можно будет вызывать. Другими словами, определение функции должно быть выполнено прежде, чем функция будет выполнена в первый раз.

Упражнение 3.1 Переместите последнюю строчку нашей программы в самый верх так, чтобы вызов функции происходил раньше ее определения. Запустите программу и посмотрите, какие сообщения об ошибках вы получите.

Упражнение 3.2 Верните вызов функции обратно в самый низ, также поместите определение `print_lyrics` после определения `repeat_lyrics`. Что произойдет, когда вы запустите программу?

3.7 Поток вычислений

Чтобы быть уверенным в том, что функция определена до ее первого использования, вы должны знать **поток вычислений**[6] (flow of execution).

Вычисления всегда начинаются с первого предложения программы, но помните, что предложения внутри функций не выполняются до тех пор, пока функция не будет вызвана.

Вызов функции нарушает последовательный поток вычислений. Вместо того, чтобы выполнять следующее предложение, вычисление перемещается в тело функции, выполняются все предложения, заключенные в нем, затем вычисление снова возвращается к тому месту, откуда была вызвана функция.

Все это может показаться довольно простым до тех пор, пока вы не вспомните, что одна функция может вызывать другую функцию. Когда вычисления находятся в середине тела одной функции, программе может потребоваться выполнить код из другой функции. Но выполняя код из той другой функции, программе может потребоваться выполнить действия из еще одной функции!

К счастью, Python следит за всеми этими вызовами, и каждый раз, как только какая-либо функция заканчивает свою работу, программа возвращается как раз на то самое место, откуда эта функция была вызвана.

Какие выводы можно сделать на основании этого? Когда вы читаете программу, то не всегда нужно читать последовательно сверху вниз. Иногда бывает проще ее понять, если следовать потоку вычислений.

---

[6] Хотя термины "поток вычислений" (flow of execution) и "приоритет операций" (rules of precedence) похожи по совоему значению, но есть и различие: первый термин относится, как правило, ко всей программе, тогда как второй термин применяется обычно по отношению к определенному выражению, например, `1/2*pi` - *прим. пер.*

3.8 Параметры и аргументы

Некоторым из встроенных функций, которые мы уже видели, требуются аргументы. Например, когда вы вызываете `math.sin`, вы передаете какое-либо число в качестве аргумента. Некоторые функции принимают больше одного аргумента. Например, `math.pow` (вычисление степени) принимает два аргумента: основание степени и ее показатель.

Внутри функции аргументы присваиваются переменным, которые называются **параметрами** (parameter). Вот пример функции, определенной пользователем, которая принимает один аргумент:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Эта функция присваивает свой аргумент параметру с именем `bruce`. Когда функция вызвана, она выводит значение своего параметра (что бы там ни было) дважды.

Эта функция работает с любым значением, которое может быть напечатано.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

То же правило композиции, что применяется к встроенным функциям, работает и для функций, определенных пользователем, поэтому в качестве аргумента функции `print_twice` мы можем использовать любое выражение:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Аргумент вычисляется до того, как будет вызвана сама функция. Поэтому в данных примерах выражения `'Spam' * 4` и `math.cos(math.pi)` вычисляются только по одному разу.

В качестве аргумента вы также можете использовать и переменную:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Имя переменной, которую мы передали в качестве аргумента (`michael`) не имеет никакого отношения к имени параметра (`bruce`). Также в данном случае не имеет значения возвращаемое функцией значение.

3.9 Переменные и параметры являются локальными

Когда вы создаете переменную внутри функции, она имеет **локальное** (local) т.е. местное значение. Это означает, что такая переменная существует только внутри этой функции. Например:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Эта функция принимает два аргумента, сцепляет их и выводит результат дважды. Вот пример ее использования:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Когда функция `cat_twice` завершается, то переменная `cat` уничтожается. Если мы попробуем распечатать ее, мы получим ошибку:

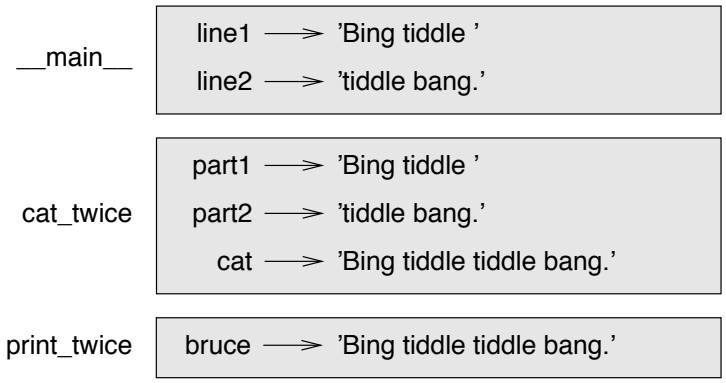
```
>>> print(cat)
NameError: name 'cat' is not defined
```

Параметры тоже локальны. Например, вне функции `print_twice` нет такой переменной, как `bruce`.

3.10 Стековые диаграммы

Чтобы графически отображать состояние всех переменных, которые мы используем, полезно рисовать **стековые диаграммы** (stack diagram). Как и диаграммы состояния, стековые диаграммы показывают значение каждой переменной, но они также показывают и функции, которым принадлежат эти переменные.

Каждая функция представлена тут **фреймом** (frame)**.** Фрейм это прямоугольник с именем функции сбоку и параметрами и переменными функции внутри его. Стековая диаграмма предыдущего примера выглядит так:



Фреймы расположены таким образом, что показывают какая функция вызвала какую и так далее. В этом примере `print_twice` была вызвана `cat_twice`, а `cat_twice` была вызвана `__main__`, что является специальным именем для самого верхнего фрейма. Когда вы создаете переменную вне функции, она принадлежит `__main__`.

Каждый параметр ссылается на то же самое значение, что и соответствующий ему аргумент. Поэтому `part1` имеет то же самое значение, что и `line1`, а у `part2` то же самое значение, что и у `line2`; `bruce` имеет то же самое значение, что и `cat`.

Если происходит ошибка, то Python выводит имя функции, в которой произошла ошибка, затем имя функции, вызвавшую функцию с ошибкой, затем имя функции, вызвавшей *my* функцию, и так далее, пока не дойдет до `__main__`.

Например, если вы попытаетесь получить доступ к переменной cat изнутри функции `print_twice`, вы получите ошибку `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: global name 'cat' is not defined
```

Этот список функций называется **отслеживанием** (traceback). Он говорит вам, в каком программном файле произошла ошибка, в какой строке, и какая функция выполнялась в это время. Он также показывает строчку кода, в которой произошла ошибка.

Порядок функций в отслеживании такой же, что и в фреймах стековой диаграммы. Функция, которая выполняется в настоящее время, находится в самом низу.

3.11 Функции, возвращающие результат и не возвращающие результата

Некоторые функции, которыми мы пользовались, такие, как математические функции, возвращают некоторый результат. За отсутствием лучшего названия, я называю их **результативные** (fruitful). Другие функции, как, например, `print_twice`, производят некоторое действие, но не возвращают никакого результата. Будем называть их **нерезультативные** (void)**.**

Когда вы вызываете результативную функцию, вы почти наверняка хотите что-то сделать с результатом; например, вы можете присвоить его переменной или использовать его как часть выражения:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Когда вы вызываете функцию в интерактивном режиме, Python выводит результат:

```
>>> math.sqrt(5)
2.2360679774997898
```

Но если вы вызываете результативную функцию в скрипте, то ее результат теряется навсегда!

```
math.sqrt(5)
```

Этот скрипт вычисляет квадратный корень из 5, но т.к. результат никуда не записывается и нигде не хранится, такой скрипт не слишком полезен.

Нерезультативные функции могут выводить что-то на экран или совершать другие действия, но они не возвращают значения. Если вы попытаетесь присвоить их результат переменной, то в итоге вы получите специальное значение, которое называется `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

Значение `None` это не то же самое, что и строка `'None'`. Это особое значение, которое имеет свой собственный тип:

```
>>> print(type(None))
<class 'NoneType'>
```

Все функции, которые мы с вами написали до сих пор, были нерезультативными. Мы начнем писать результативные функции через несколько глав.

3.12 Для чего нужны функции?

Возможно, что вы задаете себе вопрос: зачем утруждать себя разбиением программы на функции? Для этого есть несколько причин:

- Создание новой функции дает вам возможность дать определенное имя группе вычислений, что сделает вашу программу проще в чтении и отладке.
- Функции могут уменьшить размер вашей программы, избавив вас от необходимости повторять один и тот же код. Позже, если вы захотите внести в него изменения, вам нужно будет сделать это только в одном месте.

- Разбиение длинной программы на части позволяет вам отлаживать ее по частям, а затем собрать в единое целое.
- Хорошо написанные функции часто могут пригодиться и в других программах. Если вы напишете и отладите одну функцию, вы можете использовать ее в других программах.

3.13 Отладка

Если вы пользуетесь обычным текстовым редактором для написания ваших скриптов, вы легко можете столкнуться с проблемой, связанной с пробелами и знаками табуляции. Лучшим способом избежать подобных проблем является исключительное использование пробелов (т.е. не использовать табуляцию вовсе). Большинство текстовых редакторов, которые способны подсвечивать синтаксис Python, делают это по умолчанию, но некоторые этого не делают.

Знаки табуляции и пробелы обычно не видимы, поэтому их затруднительно отлаживать. Попытайтесь найти для использования такой редактор, который сам делает правильные отступы.

Также не забывайте сохранять ваши программы перед тем, как вы запускаете ее на исполнение. Некоторые рабочие окружения делают это автоматически, но не все. В последнем случае программа, которую вы отлаживаете в редакторе, и та, что запускается, это не одно и то же.

Отладка может занять довольно длительное время, если вы снова и снова запускаете неисправленную программу!

Убедитесь, что код, на который вы смотрите, это тот же самый код, который исполняется. Если вы в этом не уверены, поместите что-то вроде `print('hello')` в начале вашей программы и запустите ее снова. Если вы не увидите `hello`, значит вы запускаете не ту программу!

3.14 Словарь терминов

функция (function): именованная последовательность вычислений, которые совершают некоторые полезные действия. Функции могут принимать или не принимать аргументы, также они могут возвращать или не возвращать результат.

определение функции (function definition): строки, которые создают новую функцию, определяя ее имя и параметры, а также действия, которые будут выполнены.

объект `function` (function object): значение, созданное определением функции. Имя функции является переменной, которая ссылается на объект `function`.

заголовок (header): первая строка определения функции.

параметр (parameter): имя, используемое внутри функции, которое ссылается на значение, переданное в качестве аргумента.

вызов функции (function call): выражение, которое запускает функцию на исполнение. Оно состоит из имени функции и последующего списка ее аргументов.

аргумент[7] (argument): значение, передаваемое функции во время ее вызова. Это значение присваивается соответствующему параметру внутри функции.

локальная переменная (local variable): переменная, определенная внутри функции. Локальную переменную можно использовать только внутри функции, где она была создана.

возвращаемое значение (return value): результат, возвращаемый функцией. Если вызов функции используется как выражение, то возвращаемое значение будет значением этого выражения.

результативная функция (fruitful function): функция, которая возвращает результат.

---

[7] Часто в этой книге не делается существенного различия между *параметром* и *аргументом,* и они могут использоваться как синонимы. Когда же разница существенна, помните, что *параметр* это переменная внутри функции, а *аргумент* это то, что передается ей "снаружи" - *прим. пер*.

нерезультативная функция (void function): функция, которая не возвращает результата.

модуль (module): файл, который содержит схожие функции и другие данные.

импорт модуля (import statement): команда, которая читает файл модуля и создает объект `module`.

объект `module` (module object): значение, созданное командой `import`, дающее доступ к значениям (функциям и переменным), определенным в данном модуле.

точечная запись (dot notation): синтаксис, используемый для вызова функции, определенной в другом модуле; при этом указывается имя того модуля и функция, разделенные точкой (напр., `math.cos`).

композиция (composition): использование одного выражения в качестве составной части другого (более сложного) выражения.

поток вычислений (flow of execution): порядок, в котором выполняются действия в запущенной программе.

стековая диаграмма (stack diagram): графическое отображение стека функций, их переменных и значений, на которые они ссылаются.

фрейм (frame): прямоугольник в стековой диаграмме, представляющий собой вызов функции; он содержит локальные переменный и параметры функции.

отслеживание (traceback): список исполняемых функций, которые выводятся на экран, если происходит ошибка.

3.15 Упражнения

Упражнение 3.3 В Python есть встроенная функция `len`, которая возвращает длину строки. Так, значение `len('allen')` равняется 5.

Напишите функцию `right_justify` (выровнять по правому краю), которая берет строку `s` в качестве параметра и печатает ее так, чтобы оставить с левой стороны столько пробелов, чтобы последняя буква была на 70-й позиции.

```
>>> right_justify('allen')
                                                                allen
```

Упражнение 3.4 Объект `function` это значение, которое вы можете присвоить любой переменной или передать в качестве аргумента. Для примера, функция `do_twice` берет объект `function` в качестве аргумента и вызывает его дважды:

```
def do_twice(f):
    f()
    f()
```

Вот пример того, как `do_twice` используется для вызова функции `print_spam` дважды:

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. Запишите этот пример в скрипт и испытайте его.
2. Измените функцию `do_twice` таким образом, чтобы она принимала два аргумента – объект `function` и значение – и вызывала функцию дважды, передавая значение в качестве аргумента.
3. Напишите более общую версию функции `print_spam`, назвав ее `print_twice`, которая берет строку в качестве параметра и печатает ее дважды.
4. Используйте измененную версию `do_twice`, чтобы вызвать `print_twice` дважды, передав `'spam'` как аргумент.
5. Определите новую функцию под названием `do_four`, которая принимает объект `function` и значение, и вызывает эту функцию четыре раза, передавая принятое значение в качестве параметра. В теле функции должно быть только два предложения, не четыре.

Вы можете посмотреть на мое решение по адресу `thinkpython.com/code/do_four.py`

Упражнение 3.5 Это упражнение[8] можно выполнить, используя только те приемы, которые мы изучили до сих пор.

1. Напишите функцию, которая выводит сетку, подобную этой:

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - | - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

Подсказка: чтобы функция `print` могла вывести более одного значения в строке, вы можете разделить ее аргументы запятыми:

```
print('+', '-')
```

Чтобы заставить Python оставить линию незавершенной (т.е. чтобы следующее выводимое значение появлялось на той же самой строке), используйте следующее:

```
print('+', end="")
print('-')
```

Результатом этих двух строк будет следующий вывод:

```
+ -
```

Вызов `print()` сам по себе просто выводит пустую строку и переходит на новую строку.

2. Используйте предыдущую функцию, чтобы нарисовать похожую сетку с четырьмя строками и четырьмя столбцами.

Вы можете посмотреть на мое решение здесь `thinkpython.com/code/grid.py`.

[8] Основано на упражнении из книги *Practical C Programming*, Third Edition, O'Reilly (1997)

Глава 4
Углубленное изучение: разработка интерфейса

4.1 Модуль `TurtleWorld`

Специально для этой книги я написал набор модулей, который назвал `Swampy`. Один из этих модулей называется `TurtleWorld`. В нем содержатся функции для рисования линий на экране путем управления маленькой черепашкой. Вы можете скачать этот модуль по адресу `thinkpython.com/swampy`; здесь же я даю инструкции, как установить `Swampy` на вашей системе.

Откройте папку, в которой находится файл `TurlteWorld.py`, и создайте в ней файл с именем `polygon.py`. Введите в него следующий код:

```
from TurtleWorld import *
world = TurtleWorld()
bob = Turtle()
print(bob)
wait_for_user()
```

Первая строчка представляет собой из вариантов использования команды `import`, с которой мы уже встречались. В данном случае вместо того, чтобы создать объект `module`, она импортирует функции непосредственно, поэтому вы можете обращаться к ним без точечной записи.

Следующие две линии создают объекты `TurtleWorld` и `Turtle` и присваивают им имена `world` и `bob` соответственно. Команда `print(bob)` должна вывести что-то вроде

```
<TurtleWorld.Turtle object at 0x00FD1CD0>
```

Это означает, что `bob` ссылается на **экземпляр** (instance) `Turtle`, который определен в модуле `TurtleWorld`. В этом контексте "экземпляр" означает *один из*; `Turtle` является одним из многих возможных `Turtle`-ов.

`wait_for_user` заставляет `TurtleWorld` ожидать, чтобы пользователь сделал что-нибудь, хотя в этом конкретном случае пользователь не может сделать ничего особенного, только как закрыть окно.

`TurtleWorld` предоставляет несколько функций, управляющих движением черепашки. `fd` и `bk` двигают черепашку вперед и назад, а `lt` и `rt` поворачивают ее налево и направо соответственно. Также черепашка держит карандаш, который либо опущен вниз, либо поднят вверх. Если карандаш опущен вниз, то при движении черепашка оставляет след. Команды для поднятия и опускания карандаша – `pu` и `pd` соответственно.

Чтобы нарисовать прямой угол, добавьте эти строчки в программу (после создания `bob` и до вызова `wait_for_user`):

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

Первая линия заставляет `bob` двинуться на 100 шагов вперед. Вторая линия поворачивает его налево.

Когда вы запустите эту программу, то вы должны увидеть, как `bob` двигается на восток, а затем на север, оставляя после себя линии.

Теперь измените программу таким образом, чтобы у вас получился квадрат. Не переходите к следующему разделу, пока не выполните этого задания!

4.2 Простые повторения

Скорее всего вы написали код, подобный следующему (я не привожу строки для создания объектов `Turtle` и `wait_for_user`):

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
lt(bob)
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

Мы можем сделать то же самое более кратко с помощью инструкции `for`. Добавьте следующий пример в файл `polygon.py` и запустите его снова:

```
for i in range(4):
    print('Hello!')
```

Вы должны увидеть следующее:

```
Hello!
Hello!
Hello!
Hello!
```

Это простейший пример использования инструкции `for`; позже мы познакомимся с более сложными примерами. Но сейчас и этого должно быть достаточно, чтобы вы могли переписать код для рисования квадрата. Не читайте дальше, пока не выполните этого задания.

Вот код с инструкцией `for`, который рисует квадрат:

```
for i in range(4):
    fd(bob, 100)
    lt(bob)
```

Синтаксис для инструкции `for` похож на синтаксис для определения функции. У нее есть заголовок, заканчивающийся двоеточием, а также тело с отступом. Тело может содержать любое количество инструкций.

Инструкцию for иногда называют **циклом** (loop), т.к. вычисления проходят через все тело и возвращаются назад. В данном случае это происходит 4 раза.

На самом деле эта версия кода для рисования квадрата немного отличается от предыдущей, т.к. черепашка делает лишний поворот после того, как квадрат закончен. Этот лишний поворот отнимает дополнительное время, но если мы просто повторяем определенные действия в цикле, это упрощает наш код. Эта версия также отличается и тем, что черепашка остается в начальной позиции, поворачиваясь в направлении своего движения.

4.3 Упражнения

Следующая серия упражнений использует `TurtleWorld`. Они больше похожи на развлечение, но у них, все же, есть и определенная цель. Когда вы будете над ними работать, подумайте, какая еще у этих упражнений цель помимо забавы.

На все эти упражнения имеются ответы, но не смотрите их до тех пор, пока вы не выполните их сами, или, по крайней мере, не постараетесь как следует.

1. Напишите функцию `square`, которая берет параметр `t`, который, в свою очередь, является объектом `Turtle`. Она должна использовать этот объект для рисования квадрата.
2. Добавьте еще один параметр `length` (длина) к этой функции. Измените функцию так, чтобы длина сторон квадрата равнялась `length`. Запустите программу и испытайте ее с различными значениями `length`.

3. Функции `lt` и `rt` по умолчанию делают поворот на 90°, но они могут принимать и дополнительный аргумент, задающий угол поворота. Например, `lt(bob, 45)` повернет `bob` на 45° влево.
   Создайте копию функции `square` и переименуйте ее в `polygon`. Добавьте к ней еще один параметр `n` и измените тело функции так, чтобы она рисовала правильный `n`-сторонний многоугольник. Подсказка: внешний угол правильного `n`-стороннего многоугольника равен `360.0/n` градусов.
4. Напишите функцию `circle`, которая берет параметры `t` (объект Turtle) и радиус `r`, и рисует приблизительную окружность, вызывая функцию `polygon` с соответствующими значениями `length` и числом сторон. Испытайте функцию при различных значениях `r`. Подсказка: выясните длину окружности (`circumference`) и убедитесь, что `length * n = circumference`.
   Другая подсказка: если `bob` кажется вам слишком медленным, вы можете ускорить его, изменив его свойство `bob.delay`, которое представляет собой промежуток времени между двумя движениями. Так, например, `bob.delay = 0.01` должно заметно ускорить его движение.
5. Сделайте более общую версию `circle` под названием `arc`, которая берет дополнительный аргумент `angle` (угол), определяющий угол дуги окружности. `angle` должен измеряться в градусах, поэтому когда `angle=360`, `arc` должна нарисовать полную окружность.

4.4 Инкапсуляция

В первом упражнении от вас требовалось поместить код, рисующий квадрат, в определение функции, а затем вызвать эту функцию, передавая ей объект `Turtle` в качестве параметра. Вот как это выглядит:

```
def square(t):
    for i in range(4):
        fd(t, 100)
        lt(t)

square(bob)
```

Самые внутренние команды `fd` и `lt` имеют тут двойной отступ, т.к. они находятся внутри цикла `for`, который сам, в свою очередь, находится внутри определения функции. Следующая строка, `square(bob)`, не имеет отступа, что означает, что это уже конец как цикла `for`, так и определения функции.

Внутри функции `t` ссылается на тот же самый объект `Turtle`, что и `bob`, поэтому `lt(t)` имеет тот же самый эффект, что и `lt(bob)`. Почему бы тогда просто не вызвать параметр `bob`? Идея заключается в том, что `t` может содержать любой объект `Turtle`, не только `bob`. Поэтому вы можете создать другой объект `Turtle` и передать его как параметр функции `square`:

```
ray = Turtle()
square(ray)
```

Размещение кода в теле функции называется **инкапсуляцией** (encapsulation). Одним из преимуществ инкапсуляции является то, что вы как бы даете своему коду имя, который, к тому же, служит чем-то вроде документации. Другим преимуществом является то, что вы можете использовать код повторно. Ваш код будет более выразительным, если вы два раза вызовете функцию, нежели если вы два раза скопируете и вставите ее тело!

4.5 Обобщение

Следующим шагом будет добавление параметра `length` к функции `square`. Вот решение:

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)
```

```
square(bob, 100)
```

Добавление дополнительного параметра в функцию называется **обобщением** (generalization), т.к. это делает функцию более универсальной: в предыдущей версии квадрат был одного размера; в новой версии он может быть любого размера.

Следующим шагом также является обобщение. Вместо того, чтобы рисовать одни квадраты, `polygon` рисует правильные многоугольники с любым числом сторон. Вот мое решение:

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```

В данном случае рисуется семиугольник с длиной стороны равной 70. Если ваша функция содержит много аргументов, то легко в них запутаться и забыть, что они означают и каков их порядок. Поэтому допускается, а иногда даже и рекомендуется, добавлять имена параметров в список аргументов:

```
polygon(bob, n=7, length=70)
```

Это называется **ключевыми аргументами** (keyword argument), т.к. они состоят из имен параметров в качестве "ключевых слов" (не путайте с ключевыми словами Python, такими ка `while` и `def`).

Такой синтаксис упрощает чтение программы. Также это служит и напоминанием о том, как работают аргументы и параметры: когда вы вызываете функцию, значения аргументов передаются ее параметрам.

4.6 Разработка интерфейса

Следующим шагом будет написание функции `circle`, которая берет радиус *r* в качестве параметра. Вот простое решение, которое использует функцию `polygon` для рисования 50-стороннего многоугольника.

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

В первой строке вычисляется длина окружности с радиусом r по формуле $2\pi r$. Поскольку мы используем `math.pi`, нам необходимо импортировать модуль `math`. По соглашению, команда `import` обычно находится где-то вначале скрипта.

На самом деле мы рисуем многоугольник с большим числом сторон, который будет походить на окружность. Поэтому `n` это число сторон этого многоугольника, а `length` это длина каждой такой стороны. Таким образом, функция `polygon` рисует правильный 50-сторонний многоугольник, который приближается по виду к окружности радиуса *r*.

Одним из недостатков этого решения является то, что в нем число сторон многоугольника является константой. Это означает, что для очень больших окружностей длина стороны будет слишком большой, а для слишком маленьких окружностей мы будем тратить время на прорисовку ненужных деталей. Можно было бы обобщить эту функцию, сделав ее более универсальной, задав ей `n` в качестве дополнительного параметра. Это дало бы пользователю (т.е. тому, кто использует нашу функцию) больше выбора, но сделало бы интерфейс функции менее ясным.

Интерфейсом (interface) функции называется сумма всего того, как она используется: каковы ее параметры? что функция делает? какое значение она возвращает? Интерфейс можно назвать ясным, если он сделан "настолько простым, насколько это возможно, но не проще этого" (А. Эйнштейн).

В этом примере *r* относится к интерфейсу, т.к. определяет радиус рисуемой окружности. Значение `n` не совсем к этому относится, поскольку оно имеет вспомогательное значение и относится к деталям того, *как* окружность будет прорисована.

Вместо того, чтобы излишне усложнять интерфейс, гораздо лучше выбрать соответствующее значение n, зависящее от длины окружности:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Теперь число сторон приблизительно равняется трети длины окружности, поэтому длина каждой стороны приблизительно равняется 3, что достаточно для того, чтобы окружность выглядела плавной, и достаточно для того, чтобы сравнительно быстро рисовать окружность любого размера.

4.7 Рефакторинг (пересмотр программного кода)

Когда я писал функцию `circle`, я использовал для нее уже созданную функцию `polygon`, т.к. многоугольник с достаточно большим числом сторон приблизительно похож на окружность. Но в случае с функцией `arc` этого не получается: мы не можем использовать ни `polygon`, ни `circle` для того, чтобы нарисовать дугу.

В качестве одной из возможных альтернатив можно скопировать `polygon` и преобразовать ее в `arc`. Результат может выглядеть примерно так:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    for i in range(n):
        fd(t, step_length)
        lt(t, step_angle)
```

Вторая часть этой функции выглядит как `polygon`, но мы не можем использовать функцию `polygon` без изменения ее интерфейса. Мы можем сделать функцию `polygon` более общей, чтобы она принимала третий параметр `angle` (угол), но в этом случае имя `polygon` уже не будет соответствовать ее назначению! Вместо этого давайте назовем нашу новую универсальную функцию `polyline`:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Теперь мы можем переписать `polygon` и `arc`, чтобы они могли использовать `polyline`:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Наконец, мы можем переписать `circle`, чтобы она могла использовать `arc`:

```
def circle(t, r):
    arc(t, r, 360)
```

Такой процесс, при котором происходит пересмотр программного кода с целью улучшения интерфейса функций, а также для того, чтобы код можно было использовать повторно, называется **рефакторинг** (refactoring). В данном случае мы заметили, что код в функциях `polygon` и `arc` был похожим, поэтому мы "пересмотрели" его и преобразовали в `polyline`.

Если бы мы планировали заранее, то мы могли бы сразу написать функцию `polyline`, тогда нам не пришлось бы заниматься рефакторингом. Но довольно часто в начале какого-либо проекта вы не знаете еще достаточно, чтобы создать все необходимые интерфейсы. Как только вы станете писать код, вы начнете лучше понимать проблему. Иногда рефакторинг является знаком того, что вы научились чему-то новому.

4.8 Способ разработки

Способом разработки (development plan) называется определенный метод написания программы. Метод, который мы использовали в данной главе, называется "инкапсуляция и обобщение". Он делится на следующие шаги:

1. Начните с написания маленькой программы без функций.
2. Как только вы получите работающую программу, инкапсулируйте ее в функцию и дайте ей имя.
3. Сделайте функцию более универсальной, добавляя в нее больше параметров.
4. Повторяйте шаги 1-3, пока у вас не наберется несколько функций. Копируйте и вставляйте работающий код, чтобы избежать его повторного набора (и повторной отладки).
5. Ищите возможности улучшить программу путем рефакторинга. Например, если у вас в разный местах встречается похожий код, рассмотрите возможность его рефакторинга в одну универсальную функцию.

У этого процесса есть и свои недостатки. Мы их еще увидим позже. Но это достаточно хороший способ, если вы не знаете заранее, на какие функции разбить вашу программу. Такой подход позволяет вам проектировать по мере того, как вы продвигаетесь вперед.

4.9 Строки документации

Строкой документации (docstring) называется строка в самом начале функции, которая объясняет ее интерфейс. Вот пример:

```
def polyline(t, length, n, angle):
    """Чертит n линий заданной длины и угла
       (в градусах) между ними. t – объект turtle.
    """
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Строки документации заключаются в тройные кавычки. Еще они называются многострочными строками, т.к. могут занимать более одной линии.

К документации такого рода можно получить доступ из интерпретатора при помощи функции `help`:

```
>>> help(polyline)
Help on function polyline in module __main__:
polyline(t, length, n, angle)
    Чертит n линий заданной длины и угла
    (в градусах) между ними. t – объект turtle.
```

Хоть это и краткая документация, но она позволяет тому, кто собирается использовать данную функцию, получить о ней самую существенную информацию. Она кратко объясняет, что делает данная функция (без подробных деталей того, как она это делает). Она объясняет назначение каждого параметра, какое действие он оказывает на функцию, и какого типа должны быть эти параметры, если это не самоочевидно.

Написание подобного рода документации является важной частью разработки интерфейса. Хорошо разработанный интерфейс должен быть легок в объяснении. Если же вы затрудняетесь с тем, чтобы написать документацию к вашей функции, то, возможно, это означает то, что вам нужно переработать и улучшить интерфейс.

4.10 Отладка

Интерфейс подобен контракту между функцией и вызывающей программой. Программа соглашается предоставить ей необходимые параметры, а функция соглашается на некоторый вид работы.

Например, `polyline` требует для своей работы четыре аргумента. Первый из них должен иметь тип `Turtle`. Второй должен быть числом, причем, вероятно, положительным, хотя функция работает даже если число отрицательное. Третий аргумент обязан быть целым числом. Команда `range` выдаст ошибку в противном случае (хотя это зависит от версии Python, которой вы пользуетесь). Четвертый аргумент должен быть числом, и, понятно, что это число представляет из себя градусы.

Эти требования называются **входными условиями** (preconditions), что подразумевает их выполнение еще до того, как функция начнет свою работу. И, наоборот, условия в конце работы функции называются **выходными условиями** (postconditions). Выходные условия включают в себя ожидаемый эффект от работы функции (например, рисование линий) и любые побочные эффекты (например, движение черепашки или произведение других изменений).

Соблюдение входных условий является ответственностью вызывающей программы. Если программа не соблюдает (хорошо документированные!) входные условия, а функция отказывается работать нормально, то баг находится в вызывающей программе, а не в функции.

4.11 Словарь терминов

экземпляр (instance): член какого-либо множества. Например, в данной главе `TurtleWorld` является членом множества других `TurtleWorld`-ов.

цикл (loop): часть программы, которая выполняется по кругу.

инкапсуляция (encapsulation): процесс, при котором последовательность некоторых вычислений преобразуется в определение функции.

обобщение (generalization): процесс, при котором нечто специфичное (например, число) заменяется чем-то общим (например, переменной или параметром).

ключевой аргумент (keyword argument): аргумент, включающий в себя имя параметра в качестве "ключевого слова".

интерфейс (interface): описание того, как должна использоваться функция, включая имя, аргументы и возвращаемое значение.

рефакторинг или **пересмотр кода** (refactoring): процесс изменения готовой программы, при котором улучшается интерфейс функций и вносятся другие улучшающие изменения в код.

метод разработки (development plan): метод написания программы.

строка документации (docstring): строка, содержащаяся в определении функции и документирующая ее интерфейс.

входные условия (preconditions): требования, которые должны быть соблюдены вызывающей программой до того, как функция будет вызвана.

выходные условия (postconditions): требования, которые должны быть соблюдены функцией до того, как она закончит свою работу.

4.12 Упражнения

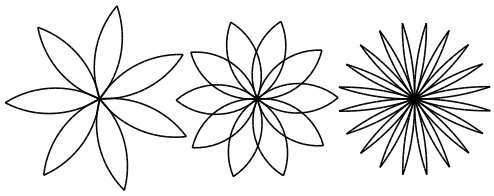
Упражнение 4.1 Скачайте код для этой главы: `thinkpython.com/code/polygon.py`

1. Напишите подходящие строки документации для функций polygon, `arc` и `circle`.
2. Нарисуйте стековую диаграмму, показывающую состояние программы, когда она выполняет `circle(bob, radius)`. Вы можете выполнять арифметические вычисления вручную или

добавить в программу вывод функции `print`.

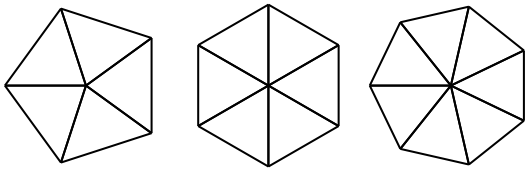
3. Версия функции `arc` из раздела 4.7 не дает слишком точных результатов, т.к. то подобие окружности, которое она создает, не является настоящей окружностью. В результате черепашка заканчивает свою работу и останавливается не совсем в том месте, где бы должна. Мое решение показывает на способ уменьшить эффект от этой ошибки. Посмотрите на код и попробуйте в нем разобраться. Если вы нарисуете диаграмму, она поможет вам увидеть, как работает программа.

Упражнение 4.2 Напишите обобщенные функции, способные рисовать фигуры, подобные этим:



Вы можете скачать решение отсюда: `thinkpython.com/code/flower.py`

Упражнение 4.3 Напишите функции, способные рисовать следующие фигуры:



Вы можете скачать решение отсюда: `thinkpython.com/code/pie.py`

Упражнение 4.4 Буквы алфавита можно составить из ограниченного набора основных элементов, таких, как вертикальные и горизонтальные линии и несколько кривых. Разработайте шрифт, который можно нарисовать, используя минимальный набор основных элементов, а затем напишите функции, которые рисуют буквы алфавита.

Вы должны написать по одной функции для каждой буквы, давая им имена, подобные этим: `draw_a`, `draw_b` и т.д. Поместите все эти функции в файл с именем `letters.py`. Вы можете скачать `"turtle typewriter"` (черепашка-принтер) с `thinkpython.com/code/typewriter.py`, что поможет вам протестировать ваш код.

Вы можете скачать мое решение с `thinkpython.com/code/letters.py`.

Глава 5
Условия и рекурсия

5.1 Оператор %

В Python имеется специальный оператор, который производит деление целых чисел, но его результатом является остаток от их деления. Этот оператор обозначается знаком процентов `%` . Он используется точно так же, как и другие операторы:

```
>>> quotient = 7 / 3    # частное
>>> print(quotient)
2.3333333333333335
>>> remainder = 7 % 3   # остаток
>>> print(remainder)
1
```

Поэтому 7 разделить на 3 будет 2 и 1 в остатке.

Этот оператор может оказаться чрезвычайно полезным. Например, очень легко проверить, делится ли одно число на другое – если `x%y` равняется нулю, то `x` делится нацело на `y`.

Также вы можете извлечь самую правую цифру из числа. Например, `x%10` выдает вам самую правую цифру числа `x` (по основанию 10). Подобным же образом, `x%100` выдает две последние цифры.

5.2 Булевы выражения

Булевым выражением (boolean expression) называется такое выражение, которое может принимать только два значения: либо `True` (истина), либо `False` (ложь). В следующих примерах используется оператор `==`, который сравнивает между собой два операнда и выдает `True`, если они равны, и `False` в противном случае:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` и `False` это специальные значения, относящиеся к типу `bool`. Это не строки:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Оператор == является одним из **операторов сравнения** (relational operator). Вот другие операторы сравнения:

```
x != y      # x не равен y
x > y       # x больше чем y
x < y       # x меньше чем y
x >= y      # x больше либо равен y
x <= y      # x меньше либо равен y
```

Хотя эти операции вам, вероятно, знакомы, в Python они обозначаются символами, которые немного отличаются от тех, что приняты в математике. Часто распространенной ошибкой является использование одиночного знака равенства = вместо двойного ==. Помните, что = это оператор присваивания, а == это оператор сравнения. Также в Python нет таких операторов, как =< или =>.

5.3 Логические операторы

Есть три **логических оператора** (logical operator): `and` (И), `or` (ИЛИ), `not` (НЕ). Семантика (значение) этих операторов близко их значению в английском (или русском) языке. Например, `x > 0 and x < 10` истинно тогда и только тогда, когда `x` больше `0` И меньше `10`.

`n%2 == 0 or n%3 == 0` истинно тогда, когда любое из этих двух условий истинно, т.е. когда `n` делится ИЛИ на `2`, ИЛИ на 3.

Наконец, оператор `not` производит логическое отрицание, поэтому `not(x > y)` истинно, если `x > y` ложно, другими словами, если `x` меньше либо равен `y`.

Строго говоря, все операнды логических операторов должны быть булевыми выражениями, но Python не относится слишком строго к данному правилу. Любое число, не равное нулю, он воспринимает как `True`:

```
>>> 17 and True
True
```

Это свойство может оказаться полезным, но имеются также и случаи, когда это же самое свойство ведет к трудноуловимым ошибкам. Поэтому старайтесь не пользоваться этой возможностью (разве только вы делаете это намеренно).

5.4 Условное выполнение

В практике программирования нам почти всегда требуется проверять некоторые условия и изменять поведение программы в зависимости от них. **Условный переход** (conditional statement) дает нам такую возможность. Вот простейший образец инструкции `if`:

```
if x > 0:
    print('x – число положительное')
```

Булево выражение после `if` называется **условием** (condition). Если оно истинно, то выполняется блок кода, который имеет отступ. Если оно ложно, то ничего не происходит.

Инструкция `if` имеет такую же структуру, что и определение функции: сначала идет заголовок, а затем с отступом – тело. Подобные конструкции еще называют **составным оператором** (compound statement).

Нет ограничения на количество предложений, которое может содержать тело, но должно быть, как минимум, одно. Однако, иногда возникает такая необходимость, чтобы в теле не было ни одного предложения (обычно такое происходит, когда вы пишите заготовку кода, и хотите просто зарезервировать место). В таком случае вы можете поместить туда команду `pass`, которая ничего не делает:

```
if x < 0:
    pass   # написать обработчик отрицательных величин
```

5.5 Альтернативное выполнение

Второй формой инструкции `if` является **альтернативное выполнение** (aleternative execution), в котором имеется две возможности и некоторое условие, определяющее, какая из возможностей получит управление:

```
if x%2 == 0:
    print('x число четное')
else:
    print('x число нечетное')
```

Здесь подразумевается, что `x` является целым числом. Если остаток от деления `x` на `2` равен `0`, тогда мы знаем, что число четное, и программа выводит соответствующую надпись. Если же условие оказывается ложным, то выполнение переходит на второй набор предложений. Так как условие может быть только

одним из двух – либо истинным, либо ложным – только одна из этих двух альтернатив будет выполнена. Такие альтернативы называются **ветвями** (branch), т.к. они разветвляют поток вычислений.

5.6 Последовательные условия

Иногда существует более двух возможностей выбора, поэтому двух ветвей может оказаться недостаточно. Один из способов осуществлять подобные вычисления называется **последовательные условия** (chained conditional):

```
if x < y:
    print('x меньше y')
elif x > y:
    print('x больше y')
else:
    print('x и y равны')
```

`elif` это сокращение от "else if" (иначе если). Опять же, только одна из всех ветвей будет выполнена. Нет ограничения на количество инструкций `elif`. Если имеется ветвь `else`, то она должна располагаться в самом конце. Но ее наличие не обязательно:

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Каждое условие проверяется по порядку. Если первое условие не выполняется, то проверяется следующее и т.д. Если очередное условие оказывается истинным, то оно и выполняется, и на этом проверка условий заканчивается. Даже если истинным будет более одного условия, выполнится только первое из них.

5.7 Вложенные условия

Одно условие может быть также вложено в другое. Мы можем переписать предыдущий пример следующим образом:

```
if x == y:
    print('x и y равны')
else:
    if x < y:
        print('x меньше y')
    else:
        print('x больше y')
```

Внешнее условие содержит две ветви. В первой ветви находится простое предложение. Во второй ветви содержится другое `if` условие, которое само разбивается на две ветви. Эти две ветви сами по себе являются простыми предложениями, хотя они могли бы быть и другими условиями `if`.

Хотя наличие отступов и делает структуру программы яснее, тем не менее, **вложенные условия** (nested conditional) очень быстро затрудняют чтение программы. В общем случае рекомендуется использовать их как можно меньше.

Логические операторы часто могут упростить вложенные условия. Например, мы можем переписать данный код следующим образом (подразумевая, что x – целое число):

```
if 0 < x:
    if x < 10:
        print('x - положительная цифра.')
```

Функция `print` выполняется только в том случае, если выполнены были оба условия. Мы можем добиться того же самого эффекта и с использованием оператора `and`:

```
if 0 < x and x < 10:
    print('x – положительная цифра.')
```

5.8 Рекурсия

Допускается, если одна функция вызывает другую. Более того, функция может вызывать даже сама себя. На первый взгляд может показаться не совсем ясным, для чего это нужно, но в реальности это одна из самых магических вещей, которые способна выполнять программа. Посмотрите, например, на следующую функцию:

```
def countdown(n):
    if n <= 0:
        print('Старт!')
    else:
        print(n)
        countdown(n-1)
```

Если `n` равно `0` или отрицательное число, то она сразу же выводит надпись "`Старт!`". В противном же случае она выводит значение `n` и потом вызывает функцию с именем `countdown` – саму себя – передавая ей в качестве аргумента `n-1`.

Что произойдет, если мы вызовем функцию подобным образом?

```
>>> countdown (n)
```

Выполнение `countdown` начнется с `n=3`, и так как `n` больше `0`, то программа выведет значение `3` и потом вызовет сама себя…

> Выполнение `countdown` начнется с `n=2`, и так как `n` больше `0`, то программа выведет значение `2` и потом вызовет сама себя…
>
> > Выполнение `countdown` начнется с `n=1`, и так как `n` больше `0`, то программа выведет значение `1` и потом вызовет сама себя…
> >
> > > Выполнение `countdown` начнется с `n=0`, и так как `n` не больше `0`, то программа выведет "`Старт!`" и вернется.
> >
> > `countdown` получит `n=1` и вернется.
>
> `countdown` получит `n=2` и вернется.

`countdown` получит `n=3` и вернется.

После этого мы снова окажемся в `__main__`. Поэтому вывод программы будет следующим:

```
3
2
1
Старт!
```

Функция, которая вызывает сама себя, называется **рекурсивной** (recursive). Процесс называется **рекурсией** (recursion).

В качестве другого примера мы можем написать программу, которая печатает строку `n` раз:

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

Если `n <= 0`, то по команде `return` происходит выход из функции. Выполнение немедленно возвращается в управляющую программу, оставшиеся строки функции не исполняются. Оставшаяся часть этой функции похожа на функцию `countdown`: если `n` больше `0`, она выводит `s` и вызывает сама себя для того, чтобы отобразить `s` еще `n-1` раз. Таким образом, количество выведенных линий равняется `1 + (n – 1)`, что, в конечном итоге, сводится к `n` повторениям.

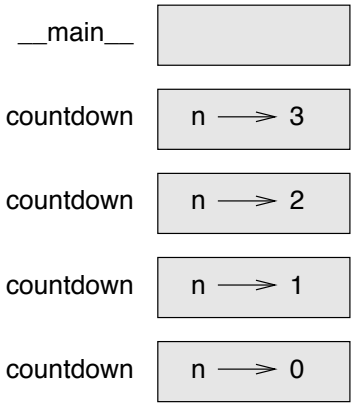
В простых примерах, подобных этому, вероятно, проще будет использовать цикл `for`, но позже мы познакомимся с примерами, которые трудно написать с этим циклом, но легко сделать с рекурсией, поэтому лучше начать пораньше.

5.9 Стековая диаграмма рекурсивных функций

В секции 3.10 мы пользовались стековой диаграммой, чтобы отобразить состояние программы во время выполнения вызова функции. Подобная же диаграмма поможет нам понять, как работает рекурсивная функция.

Каждый раз при вызове функции Python создает новую функцию (отображаемую фреймом), содержащую локальные переменные и параметры. Для рекурсивной функции в стеке может быть более одного фрейма в то же самое время.

Следующий рисунок демонстрирует стековую диаграмму функции `countdown` при `n=3`:



Как обычно, наверху стека находится фрейм с функцией `__main__`. Он пустой, потому что мы не создавали переменных в `__main__` и не передавали ему никаких параметров.

В четырех фреймах `countdown` находятся разные значения параметра `n`. Самый низ стека, где `n=0`, называется **базовым значением** (base case). Он не совершает рекурсивного вызова, поэтому за ним больше нет фреймов.

Начертите стековую диаграмму для `print_n`, вызванную с `s='Hello'` и `n=2`.

Напишите функцию `do_n`, которая в качестве своих аргументов принимает другую функцию, а также число `n`, и вызывает данную функцию `n` раз.

5.10 Бесконечная рекурсия

Если рекурсия никогда не доходит до базового значения, то она делает рекурсивные вызовы бесконечно, и программа никогда не закончится. Такое состояние называется **бесконечной рекурсией** (infinite recursion), и, как правило, считается ошибкой. Вот пример самой минимальной программы с бесконечной рекурсией:

```
def recurse():
    recurse()
```

В большинстве компьютерных систем программа с бесконечной рекурсией на самом деле не будет выполняться бесконечно. Python сообщит об ошибке, когда будет достигнута максимально допустимая глубина рекурсии:

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
```

```
                .
                .
                .
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

Это отслеживание немного больше, чем то, что мы видели в предыдущей главе. Когда происходит эта ошибка, тогда стековая диаграмма будет состоять из 1000 фреймов!

5.11 Ввод с клавиатуры

Все программы, которые мы до сих пор написали, немного скучноваты в том плане, что они не позволяют пользователю вводить данные, и каждый раз, когда мы их запускаем, они делают одно и то же.

В Python есть встроенная функция `input`, которая предназначена для того, чтобы принимать ввод с клавиатуры. Когда эта функция вызвана, программа останавливается и ожидает до тех пор, пока пользователь не введет что-нибудь. Когда же пользователь нажимает <Return> или <Enter>, программа возобновляет свою работу, и функция `input` возвращает в виде строки то, что ввел пользователь.

```
>>> inp = input()
И чего мы ждем?
>>> print(inp)
И чего мы ждем?
```

Перед тем, как запросить ввод у пользователя, желательно вывести приглашение, подсказывающее ему, что он должен ввести. `input` может принимать такое приглашение в качестве аргумента:

```
>>> name = input(Как вас зовут?\n')
Как вас зовут?
Артур, король бриттов!
>>> print(name)
Артур, король бриттов!
```

Последовательность `\n` в конце приглашения представляет из себя **символ новой строки** (new line). Это специальный символ, который вставляет разрыв в конце линии. Именно из-за него пользователь вводит свое имя на строке, которая ниже приглашения.

Если вам необходимо, чтобы пользователь ввел целое число, вы можете попытаться конвертировать возвращенное значение в тип `int`:

```
>>> prompt = 'Какова скорость ласточки?\n'
>>> speed = input(prompt)
Какова скорость ласточки?
17
>>> int(speed)
17
```

Но если пользователь введет нечто другое, нежели строку из цифр, то вы получите ошибку:

```
>>> speed = input(prompt)
Какова скорость ласточки?
Какой ласточки: африканской или европейской?
>>> int(speed)
Traceback (most recent call last):
    File "<pyshell#8>", line 1, in <module>
        int(speed)
ValueError: invalid literal for int() with base 10: 'Какой ласточки: африканской
или европейской?'
```

Позже мы познакомимся с тем, как обрабатывать подобные ошибки.

5.12 Отладка

Отслеживание, которое Python показывает при возникновении ошибки, может содержать множество информации, но она зачастую бывает перенасыщенной, особенно, когда в вашем стеке множество фреймов. Наиболее полезные части этой информации, как правило следующие:

- Вид произошедшей ошибки и
- Где она произошла

Как правило, синтаксические ошибки обнаружить проще всего, но бывают и некоторые нюансы. Например, ошибки, связанные с отступами, бывает достаточно трудно отследить, т.к. пробелы и знаки табуляции невидимы, и мы их, как правило, игнорируем:

```
>>> x = 5
>>>  y=6
    y=6
SyntaxError: unexpected indent
```

В этом примере проблема возникла из-за лишнего пробела вначале второй строки. Но сообщение об ошибке указывает на `y`, что несколько сбивает с толку. В общем случае сообщение об ошибке указывает на то, где была обнаружена проблема, но сама ошибка может быть в коде выше, иногда в предыдущей строке.

То же самое относится и к ошибкам во время выполнения. Предположим, вы пытаетесь вычислить отношение сигнал/шум в децибелах по формуле $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. В Python вы можете сделать это примерно так:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

Но когда вы запустите программу, вы получите сообщение об ошибке:

```
Traceback (most recent call last):
    File "snr.py", line 5, in ?
        decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

Сообщение указывает на строку 5, но в этой строке нет никакой ошибки. Чтобы отыскать реальную ошибку, полезно распечатать значение `ratio`, которое будет равняться 0. Проблема заключается в строке 4, потому что оператор `//` производит целочисленное деление. Заменив его на обычный оператор деления `/` , мы решим проблему.

В общем случае, сообщения об ошибках говорят, где проблема была обнаружена, но не где она реально произошла.

5.13 Словарь терминов

оператор % (modulus operator): оператор, производящий деление целых чисел и возвращающий остаток от деления.

булево выражение (bool expression): выражение, которое способно принимать только два значения: либо `True` (истина), либо `False` (ложь).

оператор сравнения (relational operator): один из операторов, который сравнивает операнды: `==`, `!=`, `<`, `>`, `<=` и `>=`.

логический оператор (logical operator): один из операторов, комбинирующий булевы выражения: `and` (И), `or` (ИЛИ), `not` (НЕ).

условный переход (conditional statement): предложение, контролирующее порядок выполнения вычислений в зависимости от определенного условия.

условие (condition): булево выражение в условном переходе, определяющее, какая ветвь программы будет выполняться.

составной оператор (compound statement): предложение, состоящее из заголовка и тела. Заголовок заканчивается двоеточием (:). Тело имеет отступ относительно заголовка.

альтернативное выполнение (alternative execution): вид условного перехода, в котором имеется две возможности выбора, и переход осуществляется на один из них.

ветвь (branch): одна из альтернативных последовательностей предложений в условном переходе.

последовательные условия (chained conditional): условный переход с несколькими альтернативными ветвями.

вложенные условия (nested conditional): условный переход, находящийся на одной из ветвей другого условного перехода.

рекурсия (recursion): процесс вызова функции, которая выполняется в настоящий момент.

базовое значение (base case): условная ветвь в рекурсивной функции, которая не совершает рекурсивного вызова.

бесконечная рекурсия (infinitive recursion): рекурсия, которая либо не имеет основания, либо никогда его не достигает. В конце концов бесконечная рекурсия вызывает ошибку во время исполнения.

символ новой строки (new line): специальная последовательность `\n`, означающая переход на новую строку. Воспринимается как один символ.

5.14 Упражнения

Упражнение 5.1 Последняя теорема Ферма утверждает, что среди целых чисел *a*, *b* и *c* нет таких, которые бы удовлетворяли равенство $a^n + b^n = c^n$ при n больше 2.

1. Напишите функцию `check_fermat`, которая принимает четыре параметра – `a`, `b`, `c` и `n` – и проверяет, выполняется ли теорема Ферма. Если `n` больше 2, и равенство соблюдается, программа должна вывести надпись "`Невероятно, Ферма ошибся!`" В противном случае программа должна вывести "`Нет, это не подходит`."
2. Напишите функцию, которая приглашает пользователя ввести данные `a`, `b`, `c` и `n`, конвертирует их в целые числа, а затем использует функцию `check_fermat` для проверки теоремы Ферма.

Упражнение 5.2 Если вам даны три отрезка, то вы либо можете, либо не можете составить из них треугольник. Например, если один отрезок длиной 12 дюймов, а два остальных по 1 дюйму, то очевидно, что вы не можете заставить короткие отрезки сойтись на середине длинного. Для любых трех отрезков есть простой тест на проверку, можно ли из них составить треугольник: если любой из отрезков длиннее суммы двух остальных, то из них нельзя составить треугольник; в противном случае можно[9].

1. Напишите функцию с названием `is_triangle`, которая берет три целых числа в качестве аргументов и выводит надписи "`Да`" или "`Нет`" в зависимости от возможности построить треугольник с такими сторонами.
2. Напишите функцию, которая предлагает пользователю ввести величины трех сторон предполагаемого треугольника, конвертирует их в целые числа, а затем использует функцию `is_triangle` для проверки возможности построения треугольника с заданными сторонами.

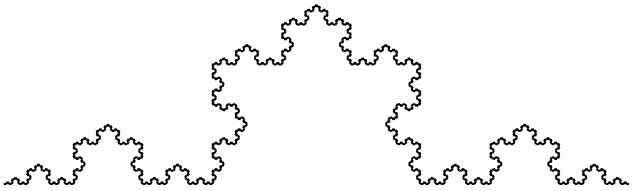
Следующие упражнения используют `TurtleWorld` из главы 4.

Упражнение 5.3 Прочитайте следующую функцию и попробуйте определить, что она делает. Потом запустите ее (как это сделать, описано в главе 4).

[9] Если сумма двух сторон равна третьей, то такой треугольник называется "вырожденным".

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    fd(t, length*n)
    lt(t, angle)
    draw(t, length, n-1)
    rt(t, 2*angle)
    draw(t, length, n-1)
    lt(t, angle)
    bk(t, length*n)
```

Упражнение 5.4 Кривая Коха это фрактал, который выглядит следующим образом:



Чтобы нарисовать кривую Коха длиной x, вам нужно сделать следующее:

1. Нарисовать кривую Коха длиной x/3.
2. Повернуть влево на 60°.
3. Нарисовать кривую Коха длиной x/3.
4. Повернуть направо на 120°.
5. Нарисовать кривую Коха длиной x/3.
6. Повернуть налево на 60°.
7. Нарисовать кривую Коха длиной x/3.

Единственным исключением является случай, когда x меньше 3. Тогда вы можете просто нарисовать прямую линию длиной x.

1. Напишите функцию под названием `koch`, которая берет объект `Turtle` и длину в качестве параметров и использует `Turtle`, чтобы нарисовать кривую Коха заданной длины.
2. Напишите функцию под названием `snowflake`, которая рисует три кривых Коха, делающих очертания снежинки. Вы можете посмотреть на мое решение `thinkpython.com/code/koch.py`.
3. Кривую Коха можно сделать более универсальной несколькими способами. Почитайте статью на википедии `wikipedia.org/wiki/Koch_snowflake`, посмотрите на примеры и сделайте тот, что вам понравится.

Глава 6
Результативные функции

6.1 Возвращаемые значения

Некоторые из встроенных функций, которые мы уже использовали, такие, как, например, математические функции, производят некоторые результаты. При вызове такой функции вычисляется некоторое значение, которое затем можно присвоить некоторой переменной или использовать как часть выражения.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

Все функции, которые мы с вами писали до сих пор не являются результативными; они печатают некоторое значение или передвигают черепашку, но их возвращаемое значение равняется `None`.

В этой главе мы (наконец-то) начнем писать результативные функции. Первым примером будет функция `area`, которая возвращает площадь круга заданного радиуса:

```
def area(radius):
    temp = math.pi * radius**2
    return temp
```

Мы уже встречались с командой `return`, но в результативных функциях эта команда включает в себя некоторое выражение. Эта команда означает: "Немедленно вернись из функции и используй следующее выражение в качестве возвращаемого значения". Выражение может быть сколь угодно сложным, поэтому мы можем переписать данную функцию в более сжатом виде:

```
def area(radius):
    return math.pi * radius**2
```

С другой стороны, **временные переменные** (temporary variable), подобные `temp`, часто облегчают процесс отладки.

Иногда возникает необходимость использовать команду `return` несколько раз, например, в каждой ветви условного перехода:

```
def absolute_value(x):   # вычисление модуля числа
    if x < 0:
        return -x
    else:
        return x
```

Т.к. команда `return` находится в альтернативных ветвях условия, только одна из них будет выполнена. Как только выполняется какая-либо команда `return`, функция немедленно прекращает свою работу, и весь остальной код, содержащийся в ней, игнорируется. Код, который следует после выражения с командой `return`, или любой другой код, который ни при каких условиях не будет выполнен, называется **мертвым кодом** (dead code).

Создавая результативную функцию нужно убедиться, что при любых возможных ситуациях вычисление пройдет через одну из команд `return`. Например:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Эта функция некорректна, т.к. если `x` будет равен `0`, то это не удовлетворит ни одно из двух условий, а функция закончит работу не выполнив команды `return`. Если вычисления доходят до конца функции, то возвращаемое значение будет `None`, что не является тем же самым, что и значение `0`.

```
>>> print absolute_value(0)
None
```

Кстати, в Python в модуле `math` имеется встроенная функция `abs`, которая вычисляет модуль введенного значения.

Упражнение 6.1 Напишите функцию `compare` (сравнение), которая выводит `1` при `x > y`, `0` при `x == y`, и `1` при `x < y`.

6.2 Пошаговая разработка

Когда вы пишите большие функции, то, как правило, тратите значительное время на их отладку.

Для написания сложных программ вы можете попробовать метод, который называется **пошаговая разработка** (incremental development). Суть метода состоит в том, чтобы не доводить программу до состояния, когда вы будете тратить значительное время на поиск и исправление ошибок. Вместо этого вы добавляете и испытываете маленькие порции кода за раз.

Давайте возьмем в качестве примера ситуацию, когда вам необходимо найти расстояние между двумя

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

точками, заданными координатами (x_1, y_1) и (x_2, y_2). Расстояние находится по теореме Пифагора:

Прежде всего нужно посмотреть, как функция `distance` должна выглядеть в Python. Другими словами, каковы ее входные значения (параметры) и выходные значения (возвращаемое значение).

В данном случае на входе мы имеем две точки, которые можно представить в виде четырех чисел. Возвращаемое значение это расстояние, которое будет иметь дробный тип (`float`). Не забудьте импортировать модуль `math`.

Мы уже можем записать заготовку нашей функции:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Разумеется, данная версия не вычисляет расстояние; она всегда возвращает 0. Но у нее правильный синтаксис, ее можно запустить, что означает, что вы можете испытать ее до того, как начнете ее усложнять.

Для испытания данной функции вызовите ее с простыми аргументами:

```
>>> distance(1, 2, 4, 6)
0.0
```

Я выбрал эти значения не случайно. Здесь горизонтальное значение равняется 3, а вертикальное – 4. Поэтому ответ должен быть 5 (гипотенуза треугольника со сторонами 3-4-5). Испытывая какую-либо функцию, всегда полезно знать заведомо верный ответ.

На этой стадии мы убедились, что функция имеет правильный синтаксис, поэтому мы можем начать добавлять код в ее тело. Поэтому следующим разумным решением будет нахождение разностей $x_2 - x_1$ и $y_2 - y_1$. Поэтому следующая версия хранит эти значения во временных переменных и распечатывает их:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx =', dx)
    print('dy =', dy)
    return 0.0
```

Если функция работает, то она должна вывести `'dx = 3'` и `'dy = 4'`. Если у нас получился такой результат, мы знаем, что функция принимает правильные аргументы и правильно производит первые вычисления. Если же нет, то нам нужно проверить всего несколько строчек.

На следующем шаге мы подсчитываем сумму квадратов `dx` и `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Опять же, вы можете запустить программу и проверить ее вывод, который должен равняться 25. Наконец, извлечем квадратный корень `math.sqrt`, чтобы вычислить окончательное значение, и вернем результат:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Если функция работает правильно, значит мы выполнили свою задачу. Если же нет, то, возможно, вам нужно будет распечатать значение `result` перед тем, как вы выполните команду `return`.

Окончательная версия функции не выводит никакого результата во время своей работы, она только возвращает значение. Команды `print`, которые мы использовали в ней, полезны на стадии отладки, но как только мы убедились, что функция работает правильно, их лучше сразу удалить. Такой код называется **отладочным кодом** (scaffolding code), т.к. он используется во время отладки, но не является частью готовой программы.

Когда вы только начинаете писать функцию, вы добавляете лишь одну или две строки за раз. С приобретением опыта вы обнаружите, что становитесь способны писать больше кода за раз. В любом случае, пошаговая разработка поможет вам сэкономить время на отладке.

Вот ключевые моменты этого процесса:

1. Начните с работающей программы и делайте постепенные изменения. Как только вы обнаружите ошибку, вам легко найти место, где она могла произойти.
2. Используйте временные переменные для хранения промежуточных результатов, тогда вы сможете распечатывать их значения для проверки.
3. Как только вы получите работающую программу, можно удалить отладочный код, а также объединить короткие выражения в более длинные, если только это не затрудняет чтение программы.

Упражнение 6.2 Используйте метод пошаговой разработки, чтобы написать функцию `hypotenuse`, которая возвращает длину гипотенузы прямоугольного треугольника, имея длины двух катетов в качестве аргументов. Записывайте каждую стадию этого процесса разработки по мере продвижения.

6.3 Композиция

Вы уже знаете, что одну функцию можно вызывать из другой. Напомним, что такая возможность называется композицией. В качестве примера мы напишем функцию, которая принимает координаты двух точек – центра окружности, и точки, лежащей на ее периметре, – и вычисляет площадь получившегося круга. Обозначим координаты центра как `xc` и `yc`, а точку на периметре как `xp` и `yp`. Первым шагом будет нахождение радиуса окружности, который есть ни что иное, как расстояние между этими двумя точками. Мы только что написали функцию `distance`, которая как раз и вычисляет это расстояние:

```
radius = distance(xc, yc, xp, yp)
```

Следующим шагом будет нахождение площади круга заданного радиуса. Мы тоже только что написали такую функцию:

```
result = area(radius)
```

Инкапсулируя эти шаги в функцию, получим:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Временные переменные `radius` и `result` полезны при разработке и отладке, но в окончательной версии программы они не нужны, поэтому мы можем переписать ее в более сжатом виде, используя композицию:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Булевы функции

Функции могут возвращать булевы значения, что достаточно удобно, когда мы хотим спрятать сложные вычисления внутри функции. Пример:

```
def is_divisible(x, y): # делится ли x на y
    if x % y == 0:
        return True
    else:
        return False
```

Принято давать булевым функциям имена, которые звучат как да/нет вопросы. `is_divisible` возвращает `True` или `False` в зависимости от того, делится ли `x` на `y`. Вот пример:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Результат оператора == тоже булевого типа, поэтому мы можем написать функцию более лаконично, возвращая его непосредственно:

```
def is_divisible(x, y):
    return x % y == 0
```

Булевы функции часто используются в условных переходах:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

У вас может возникнуть искушение написать что-то подобное этому:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

Но оператор сравнения здесь не является обязательным.

Упражнение 6.3 Напишите функцию `is_between(x, y, z)`, возвращающую `True`, если `x <= y <= z`, или `False` в противном случае.

6.5 Больше примеров рекурсий

Мы изучили еще сравнительно мало из того, на что способен Python, но вам, вероятно, интересно будет узнать, что то, что мы уже изучили, является *достаточным* языком программирования, что означает, что все, что только может быть вычислено, может быть выражено при помощи тех средств, которые мы уже изучили. Любая когда-либо написанная программа может быть переписана с использованием только тех средств, которые мы уже изучили (на самом деле нам еще понадобятся команды, контролирующие устройства ввода-вывода, такие как клавиатура, мышь, диски и т.п., но это все).

Первым, кто доказал это необычное утверждение, был Алан Тьюринг, один из первых компьютерных специалистов (некоторые утверждают, что он был математиком, но многие компьютерные специалисты начинали как математики). Это утверждение известно под названием *тезис Тьюринга*. За более полной и

точной информацией по тезису Тьюринга я рекомендую вам обратиться к книге Michael Sipser *Введение в теорию вычислений.*

Чтобы дать вам представление о том, что вы можете сделать при помощи тех средств, которые мы до сих пор изучили, мы выведем несколько рекурсивных определений математических функций. Рекурсивное определение подобно тавтологическому определению, т.е. определение содержит ссылку на тот предмет, который оно определяет. От тавтологического определения мало пользы:

замечательный – прилагательное, используемое для описания чего-нибудь замечательного.

Если бы вы увидели подобное определение в словаре, вы, вероятно, оказались бы в замешательстве. С другой стороны, если вы посмотрите на определение функции, вычисляющей факториал, выраженного символом !, то можете увидеть что-то подобное этому:

```
0! = 1
n! = n(n-1)!
```

Это определение говорит нам, что факториал `0` равен `1`, а факториал любого другого значения, `n`, равен `n`, умноженное на факториал `n-1`.

Таким образом, `3!` равно `3` умножить на `2!`, что, в свою очередь, равно `2` умножить на `1!`, что, в свою очередь, равно `1` умножить на `0!` Все вместе это означает: `3!` равно `3` умножить на `2`, умножить на `1`, умножить на `1`, что равняется `6`.

Если вы способны написать рекурсивное определение чего-либо, то, как правило, вы можете также написать и программу на Python для вычисления его значения. Прежде всего нужно решить, какие должны быть параметры. В случае с `factorial`, ясно, что это должно быть целое число:

```
def factorial (n):
```

Если аргумент равен `0`, то все, что нам остается, это вернуть `1`:

```
def factorial(n):
    if n == 0:
        return 1
```

В противном случае, и это интереснее всего, нам нужно осуществлять рекурсивный вызов, чтобы найти факториал `n-1` и, затем, умножить его на `n`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1) # рекурсия
        result = n * recurse     # результат
        return result
```

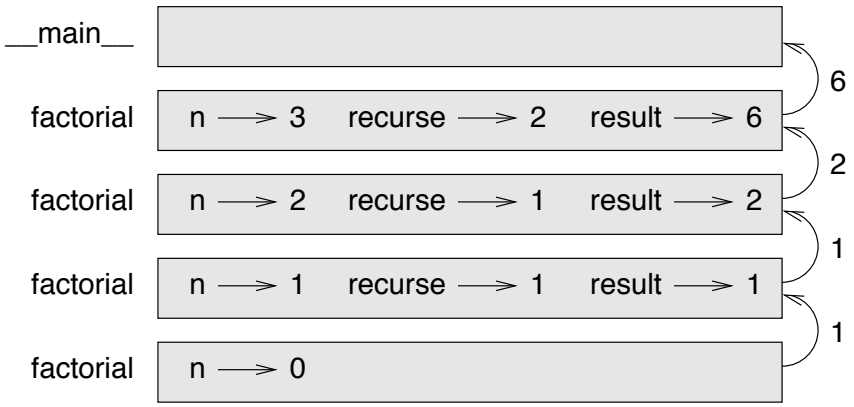
Порядок выполнение этой программы подобен порядку выполнения программы `countdown`, которую мы видели в разделе 5.8. Если мы вызовем `factorial` со значением 3:

Т.к. `3` не равно `0`, мы переходим на вторую ветвь и вычисляем факториал `n-1`...

> Т.к. `2` не равно `0`, мы переходим на вторую ветвь и вычисляем факториал `n-1`...
>
> > Т.к. `1` не равен `0`, мы переходим на вторую ветвь и вычисляем факториал `n-1`...
> >
> > > Т.к. `0` равно `0`, мы переходим на первую ветвь и возвращаем `1` без дальнейшего вызова рекурсии.
> >
> > Возвращенное значение (`1`) умножаем на `n` (равное `1`) и возвращаем результат.
>
> Возвращенное значение (`1`) умножаем на `n` (равное `2`) и возвращаем результат.

Возвращенное значение (`2`) умножаем на `n` (равное `3`), и результат, равный `6`, становится результатом, который и возвращает функция, начавшая весь этот процесс.

Вот как выглядит стековая диаграмма всех этих вызовов:



Из рисунка видно, что возвращаемые значения передаются обратно по стеку. В каждом фрейме возвращаемое значение и является результатом (`result`), т.е. произведением `n` и `recurse`.

В последнем фрейме нет локальных переменных `result` и `recurse`, т.к. та ветвь, в которой они создаются, не выполняется.

6.6 Шаг веры

Одним из способов прочитать программу является следование за потоком вычислений, однако, вскоре он превращается в лабиринт. Альтернативой этому является то, что я называю "шаг веры". Когда вы доходите до вызова функции, то вместо того, чтобы следовать за порядком ее вычислений, вы *предполагаете* (assume), что функция работает правильно и возвращает правильный результат.

На самом деле вы уже практиковали подобный шаг веры, когда пользовались встроенными функциями. Когда вы вызываете `math.cos` или `math.exp`, вы не исследуете код этих функций. Вы просто предполагаете, что они работают правильно, т.к. люди, написавшие встроенные функции – хорошие программисты.

То же самое верно и в отношении вызова ваших функций. Например, в разделе 6.4 мы написали функцию `is_divisible`, которая выясняет, делится ли одно число на другое. Как только мы убедились, что функция работает правильно – исследуя ее код и тестируя – мы можем просто пользоваться ею, не заглядывая в ее тело повторно.

То же самое верно и в отношении рекурсивных программ. Когда вы доходите до рекурсивного вызова, то вместо того, чтобы следовать за потоком вычислений, вы должны просто предположить, что рекурсивный вызов работает (производит корректный результат) и потом просто спросить себя: "Если я могу вычислить факториал `n-1`, то могу ли я вычислить также и факториал `n`?" В данном случае ответ очевиден: можете, умножая его на `n`.

Само собой, предположение о том, что функция работает правильно, когда вы еще не закончили ее написание, выглядит несколько странно, но именно поэтому я и называю это шагом веры!

6.7 Еще один пример

После вычисления факториала наиболее часто для демонстрации рекурсии используют функцию для вычислений чисел Фибоначчи[10], имеющую следующее определение:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
```

Переведя в код на Python это будет выглядеть следующим образом:

[10] Смотрите `wiki.org/wiki/Fibonacci_number`.

```
def fibonacci (n):
    if n == 0:
        return 0
    elif n==1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Если вы попытаетесь проследовать за потоком вычислений даже при сравнительно малых значениях `n`, ваш мозг взорвется. Но в соответствии с принципом "шаг веры", если вы предположите, что два рекурсивных вызова работают корректно, тогда вам станет ясно, что вы получаете верный результат, суммируя их.

6.8 Проверка типов

Что произойдет, если мы вызовем функцию `factorial` и передадим ей 1.5 в качестве аргумента?

```
>>> factorial(1.5)
RuntimeError: maximum recursion depth exceeded in comparison
```

Выглядит, как бесконечная рекурсия. Но как такое может быть? Основание рекурсии достигается, когда `n = 0`. Но если `n` не является целым числом, то мы можем *пропустить* основание, и рекурсия будет выполняться бесконечно.

При первом рекурсивном вызове значение `n = 0.5`. При следующем оно равно `-0.5`. Дальше оно становится все меньше и меньше (все более и более отрицательным), но никогда не станет равным `0`.

У нас есть два выбора. Мы можем сделать функцию `factorial` более общей, чтобы она работала и с дробными числами, или мы можем заставить `factorial` проверять тип введенного аргумента. Первый выбор называется гамма-функцией[11], но мы не будем рассматривать его в данной книге. Поэтому остановимся на втором варианте.

Мы можем воспользоваться встроенной функцией `isinstance`, которая проверяет тип аргумента. Одновременно мы можем убедиться и в том, что наш аргумент является положительным числом:

```
def factorial (n):
    if not isinstance(n, int):
        print('Факториал только для целых чисел.')
        return None
    elif n < 0:
        print('Факториал не предназначен для отрицательных чисел.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Первая проверка определяет нецелые числа. Вторая проверка отлавливает отрицательные целые числа. В обоих случаях программа выводит сообщение об ошибке и возвращает `None`, сигнализирующее о том, что что-то не в порядке:

```
>>> factorial('fred')
Факториал только для целых чисел.
None
>>> factorial(-2)
Факториал не предназначен для отрицательных чисел.
None
```

Если же мы удачно миновали обе проверки, то мы знаем, что `n` либо положительное число, либо `0`. Это доказывает, что рекурсия не будет бесконечной.

---

[11] Смотрите `wikipedia.org/wiki/Gamma_function`.

Эта программа демонстрирует прием, иногда называемый **проверкой параметров** (guardian). Первые две проверки выполняются для защиты функции от неправильно введенных параметров, способных вызвать ошибку. Такая защита может помочь убедиться в том, что у нас корректный код.

6.9 Отладка

Разбивание большой программы на меньшие функции естественным образом создает точки отладки. Если функция не работает, то на то может быть три причины:

- Проблема с аргументами, которые передаются функции: входные условия не выполнены.
- Проблема в самой функции: выходные условия не соблюдаются.
- Проблема с возвращаемым значением или тем, как оно используется.

Для исключения первой возможности, вы можете добавить вызов функции `print` в самое начало вашей функции для распечатки значений параметров (а, возможно, и их типов). Также вы можете написать код, который будет явным образом проверять входные условия.

Если с параметрами все в порядке, добавьте вызов `print` перед каждой командой `return`, чтобы видеть, какое значение возвращается. Если возможно, проверьте результат вручную. Попробуйте вызвать функцию со значениями, результат которых можно легко проверить (как в разделе 6.2).

Если вам кажется, что функция работает нормально, попробуйте ее вызывать и убедитесь, что ее возвращаемое значение используется корректно (и используется ли!).

Добавьте вызов `print` в начале и в конце функции, чтобы поток вычислений был более наглядным. Например, вот версия `factorial` с вызовом `print`:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

Здесь переменная `space` представляет из себя строку из пробелов, которые определяют величину отступа при выводе. Вот результат работы `factorial(5)`:

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
120
```

Если вы запутались, следуя за порядком выполнения действий, то такой тип вывода может вам помочь. Хотя написание отладочного кода и отнимает дополнительно время, такой код может сэкономить вам время при отладке.

6.10 Словарь терминов

временная переменная (temporary variable): переменная, используемая для хранения промежуточных результатов в сложных вычислениях.

мертвый код (dead code): часть программы, которая не будет выполнена ни при каких условиях; часто такой код находится после команды `return`.

`None`: особое значение, возвращаемое функцией, в которой нет команды `return` или `return` не содержит аргументов.

пошаговая разработка (incremental development): метод разработки программ, призванный избежать отладки, при котором в программу добавляются и испытываются лишь небольшие порции кода единовременно.

отладочный код (scaffolding): код, использующийся при разработке программы, но не являющийся частью окончательной версии.

проверка параметров (guardian): прием в программировании, при котором условные переходы используются для проверки и реагирования на обстоятельства, способные вызвать ошибку.

6.11 Упражнения

Упражнение 6.4 Нарисуйте стековую диаграмму следующей программы. Что эта программа выводит на экран?

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x=x+1
    return x * y

def c(x, y, z):
    sum = x + y + z
    pow = b(sum)**2
    return pow

x=1
y=x+1
print(c(x, y+3, x+y))
```

Упражнение 6.5 Функция Акермана A(m, *n*) определена следующим образом[12]:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Напишите функцию под названием `ack`, вычисляющую функцию Акермана. Используйте вашу функцию для вычисления значения `ack(3, 4)`, что должно равняться `125`. Что будет с большими значениями `m` и `n`?

Упражнение 6.6 Палиндром это слово, которое одинаково читается слева направо и справа налево, например, "`noon`" и "`redivider`". С точки зрения рекурсии, слово является палиндромом, если первая и последняя буквы равны, и середина (то, что остается) является палиндромом.

[12] См. `wikipedia.org/wiki/Ackermann_function`

Следующие функции принимают строку в качестве аргумента и возвращают первую, последнюю и среднюю букву, а также середину слова:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

Мы еще посмотрим на то, как они работают, в главе 8.

1. Сохраните эти функции в файле с названием `palindrome.py` и проверьте их. Что произойдет, если вы вызовете `middle` со строкой, состоящей из двух букв? Из одной буквы? А как насчет пустой строки, которая записывается как `''` и не содержит ни одной буквы?
2. Напишите функцию под названием `is_palindome`, которая принимает строку в качестве аргумента и возвращает `True`, если она является палиндромом, и `False` в противном случае. Помните, что вы можете воспользоваться встроенной функцией `len` для проверки длины строки.

Упражнение 6.7 Число *a* является степенью числа *b*, если оно делится на *b* и *a*/*b* является степенью *b*. Напишите функцию под названием `is_power`, которая принимает параметры *a* и *b* и возвращает `True`, если *a* является степенью *b*.

Упражнение 6.8 Наибольшим общим делителем (greatest common diviser – GCD) чисел *a* и *b* называется наибольшее число, на которое они оба делятся без остатка[13].

Одним из способов нахождения GCD двух чисел является алгоритм Евклида, основанный на том наблюдении, что если *r* является остатком от деления *a* на *b*, то *gcd(a,b) = gcd(b,r)*. В качестве основания мы можем принять *gcd(a,0) = a*.

Напишите функцию `gcd`, которая принимает параметры a и b и возвращает их наибольший общий делитель. Смотрите также `wikipedia.org/wiki/Euclidean_algorithm`.

[13] Это упражнение основано на примере из книги Abelson и Sussman *Структура и интерпретация компьютерных программ*.

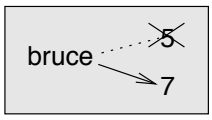
Глава 7
Итерация (повтор)

7.1 Многократное присваивание

Вероятно вы уже заметили, что допускается одной и той же переменной многократно присваивать разные значения. Новое присваивание вынуждает существующую переменную ссылаться на новое значение (и перестать ссылаться на прежнее значение):

```
>>> bruce = 5
>>> print (bruce)
5
>>> bruce = 7
print (bruce)
7
```

Вот как выглядит **многократное присваивание** (multiply assignment) в диаграмме состояния:



Совершая многократное присваивание, особенно необходимо помнить о разнице между операцией присваивания и состоянием равенства. Т.к. Python использует знак равенства (=) для присваивания, у вас может возникнуть искушение интерпретировать выражение, подобное *a = b* как выражение равенства, которое таковым не является.

Прежде всего необходимо уяснить, что равенство является симметрическим отношение, а присваивание нет. Например, в математике если *a* = 7, то 7 = *a*. Но в Python выражение *a* = 7 допустимо, а 7 = *a* нет.

Более того, в математике состояние равенства, будь оно верно или неверно, не меняется с течением времени. Если *a = b* сейчас, то *a* всегда будет равно *b*. Но в Python операция присваивания может сделать равными две переменные, но они не обязаны оставаться такими постоянно:

```
a = 5
b = a # a и b теперь равны
a = 3 # a и b больше не равны
```

В третьей строчке *a* меняет свое значение, но *b* остается прежним, поэтому они больше не равны.

Хотя множественное присваивание и полезно во многих случаях, вы должны быть с ним осторожны. Если значения переменных часто меняются, это может затруднить чтение и отладку кода.

7.2 Обновление

Одним из самых частых видов многократного присваивания является **обновление** (update), при котором новое значение переменной зависит от старого.

```
x = x + 1
```

Это означает "получить значение *x*, прибавить к нему единицу, обновить *x*, присвоив ему новое значение".

Если вы попытаетесь обновить несуществующую переменную, то получите сообщение об ошибке, потому что Python вычисляет правую часть выражения прежде левой.

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Прежде чем обновлять переменную, вам необходимо **инициализировать** (initialize) ее, обычно каким-нибудь простым значением:

```
>>> x = 0
>>> x = x+1
```

Обновление переменной добавлением 1 называется **инкремент** или **приращение** (increment), а вычитанием единицы – **декремент** или **отрицательное приращение** (decrement).

7.3 Команда `while`

Компьютеры часто используются для автоматизации задач, требующих производить многократные повторения. Повторение без того, чтобы допускать ошибки это то, что хорошо могут делать компьютеры, и плохо люди.

Мы уже видели две программы, `countdown` и `print_n`, использующих рекурсию для выполнения повторений. По-другому повторение еще называется **итерация** (iteartion). Поскольку итерация используется очень часто, язык Python предоставляет несколько конструкций для их осуществления. Одну из них – `for` – мы уже видели в разделе 4.2. Мы еще вернемся к ней позже.

Другая конструкция это команда `while`. Вот версия программы `countdown`, которая использует `while`:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print('Старт!')
```

Вы почти что можете читать эту программу, как если бы она была написана на обычном английском языке.

Говоря более формально, команда `while` работает следующим образом:

1. Вычислить условие на значение `True` или `False`.
2. Если условие ложно, то выйти из цикла `while` и продолжить выполнение со следующего за ним предложения.
3. Если условие истинно, то выполнить тело цикла и затем вернуться к шагу 1.

Напомним, что такие вычисления, ходящие по кругу, называются циклом. Тело цикла должно изменять одну или более переменных, так, чтобы в конце концов условие стало ложным и цикл завершился. В противном случае, цикл будет выполняться бесконечно, что называется **бесконечным циклом** (infinite loop). С точки зрения компьютерного специалиста инструкция на бутылке с шампунем "намылить, промыть, повторить" является также примером бесконечного цикла.

В случае с `countdown` мы можем доказать, что цикл закончится, потому что мы знаем, что значение `n` – конечно, и мы видим, что оно становится меньше и меньше по мере выполнения цикла, так что рано или поздно, но оно сравняется с `0`. В некоторых других случаях это не так то и просто определить:

```
def sequence(n):
    while n != 1:
        print(n)
        if n%2 == 0:
            n = n/2     # n - четное число
        else:
            n = n*3+1   # n - нечетное число
```

Условием этого цикла является `n != 1`, таким образом цикл будет продолжаться до тех пор, пока `n` не сравняется с `1`, что сделает условие ложным.

Каждый раз при выполнении тела цикла, программа выводит значение `n` и проверяет, четное ли оно или нет. Если четное, то `n` делится на `2`. Если нечетное, то значение `n` заменяется значением выражения `n*3+1`. Например, если начальное `n` будет равно `3`, то программа выдаст следующие результаты: `3`, `10`, `5`, `16`, `8`, `4`, `2`, `1`.

Т.к. `n` иногда то увеличивается, то уменьшается, нет очевидного доказательства, что `n` когда-либо сравняется с `1` и программа остановится. Для некоторых частных значений `n` мы можем это доказать. Например, если начальное значение является квадратом какого-либо числа, то значение `n` каждый раз будет четным числом, пока не сравняется с `1`.

Вопрос заключается в том, сможем ли мы доказать, что цикл рано или поздно завершится *при любом положительном значении* `n`. До сих пор[14] никто не мог ни доказать этого, ни опровергнуть!

Упражнение 7.1 Перепишите функцию `print_n` из раздела 5.8, используя итерацию вместо рекурсии.

7.4 Инструкция `break`

Иногда вы не знаете, когда вам нужно выходить из цикла, до тех пор, пока не пройдете, скажем, половину его тела. В подобном случае вы можете использовать инструкцию `break`, чтобы выйти из цикла.

Предположим, для примера, вам требуется принимать ввод данных от пользователя до тех пор, пока он не завершит его (введет слово `'done'`). Вы можете написать следующее:

```
while True:
    line = input('> ')
     if line == 'done':
         break
     print(line)

print('Done!')
```

Условие цикла равно `True`, что всегда истинно, поэтому цикл будет выполняться до тех пор, пока не встретится инструкция `break`.

Каждый раз программа выдает пользователю приглашение в виде угловой скобки. Если пользователь набирает слово `done`, то инструкция `break` заставляет программу выйти из цикла. В противном случае программа выводит то, что ввел пользователь и возвращается к началу цикла. Вот пример:

```
> not done
not done
> done
Done!
```

Такой способ написания цикла `while` довольно распространен, т.к. вы можете проверять условие в любом месте цикла, а не только в самом начале. Также вы можете выражать условие для выхода в утвердительной форме ("остановиться, когда произойдет следующее…"), а не в негативной ("продолжать до тех пор, пока не произойдет следующее...").

7.5 Квадратные корни

Циклы часто используются программами для получения числовых результатов, которые начинаются с приблизительного ответа и последовательно улучшаются.

Например, одним из способов вычисления квадратного корня может являться метод Ньютона. Предположим, вы хотите вычислить квадратный корень из *a*. Если вы начнете с почти любого приближенного значения, *x*, то вы можете вычислить более точное приближенное значение по следующей формуле:

$$y = \frac{x + a/x}{2}$$

Например, если *a* = 4 и *x* = 3:

[14] См. `wikipedia.org/wiki/Collatz_conjecture`.

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print(y) 2.1666666666666665
```

Этот результат ближе к правильному ответу, который равен 2. Если вы повторите процесс с уже новым приближенным значением, по получите еще более точный ответ:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.0064102564102564
```

После еще нескольких приближений ваше значение еще более приблизится к точному:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.0000102400262145
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.0000000000262146
```

Хоть мы и не знаем наперед, *сколько* шагов займет получение точного ответа, но мы знаем, *когда* это произойдет, т.к. значение перестанет изменяться:

```
>>>  x = y
>>>  y = (x + a/x) / 2
>>>  print(y) 2.0
>>>  x = y
>>>  y = (x + a/x) / 2
>>>  print(y) 2.0
```

Когда `x == y`, мы можем остановиться. Вот пример цикла, который начинает с приближенного значения, `x`, затем последовательно улучшает его до тех пор, пока оно не перестанет изменяться:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Этот пример будет правильно работать с большинством введенных значений. Но в общем случае, сравнение между собой данных типа `float` может привести к ошибке, т.к. часто они выражают лишь приближенное значение: большинство рациональных чисел, таких, как 1/3, и иррациональных чисел, таких, как $\sqrt{2}$, нельзя точно выразить при помощи типа `float`.

Вместо того, чтобы сравнивать между собой *x* и *y* на точное равенство, гораздо безопаснее использовать встроенную функцию `abs`, вычисляющую абсолютное значение (модуль), применяя ее для вычисления разности между *x* и *y*:

```
if abs(y-x) < epsilon:
    break
```

где `epsilon` принимает значение 0.0000001, что и определяет точность вычисления.

Упражнение 7.2 Инкапсулируйте этот цикл в функцию `squqre_root`, которая принимает *a* в качестве параметра, выбирает подходящее значение *x* и возвращает приближенное значение квадратного корня из *a*.

7.6 Алгоритмы

Метод Ньютона может служить примером алгоритма: это чисто механический процесс решения определенной категории задач (в данном случае, извлечение квадратного корня).

Дать определение алгоритму не так-то и легко. Можно попробовать начать с определения чего-то, что не является алгоритмом. В школе мы все учили таблицу умножения, чтобы уметь умножать однозначные числа. В действительности, вы заучили наизусть 100 специфических решений. Такое знание нельзя назвать алгоритмическим. Но если вы были "ленивы", то, возможно, схитрили, и вместо заучивания таблицы умножения на 9, воспользовались правилом для нахождения произведения n и 9: первая цифра равняется n-1, а вторая 10-n. Такой способ подходит для умножения любого однозначного числа на 9. Это и есть алгоритм!

Подобным же образом и методы сложения, вычитания, умножения и деления столбиком также являются алгоритмами. Одно из свойств алгоритма заключается в том, что они совершенно не требуют интеллекта для выполнения. Они являются чисто механическими процессами, в которых каждый следующий шаг вытекает из предыдущего в соответствии с простым набором правил.

На мой взгляд, достоин сожаления тот факт, что людей в школах учат больше исполнять алгоритмы, нежели развивать свои интеллектуальные способности.

Напротив, сам процесс разработки алгоритмов это интересное и непростое занятие, являющееся центральной частью того, что мы зовем программированием.

Некоторые из тех вещей, которые люди делают совершенно естественно, без какого-либо напряжения мысли, очень трудно выразить алгоритмически. Понимание естественного языка может служить хорошим примером. Мы все способны на это, но до сих пор еще никто не объяснил, *как* мы делаем это, по крайней мере, в форме алгоритма.

7.7 Отладка

Как только вы начинаете писать большую программу, вы тотчас обнаруживаете, что стали тратить больше времени на отладку. Больше кода означает больше шансов совершить ошибку и больше места для ошибок.

Одним из способов сократить время отладки является метод "отладки делением пополам". Например, если у вас есть 100 строчек кода, и вы проверяете их за один раз, это забирает у вас 100 шагов.

Вместо этого вы можете разделить задачу на две части. Найдите место в середине вашей программы или возле него, там, где есть промежуточное значение, которое вы можете проверить. Добавьте туда вызов функции `print` или что-то иное, что может помочь проверить вашу программу, и запустите ее.

Если вывод программы неверен, то ошибка должна быть в первой части программы. Если же он верен, то ошибка во второй части.

Каждый раз, когда вы осуществляете такую проверку, вы сокращаете вдвое число строк кода для поиска ошибки. После шести шагов (что определенно меньше 100) у вас останется для проверки одна или две строчки кода, как минимум, теоретически.

На практике не всегда бывает ясно, что такое "середина программы", и не всегда возможно осуществить там проверку. Не стоит подсчитывать линии и находить точную середину. Вместо этого подумайте о тех местах в программе, где может скрываться ошибка, а также о тех местах, в которых легко осуществить проверку.

7.8 Словарь терминов

многократное присваивание (multiple assignment): присваивание одной и той же переменной разных значений во время исполнения программы.

обновление (update): присваивание, при котором новое значение зависит от предыдущего.

инициализация (initialization): присваивание, которое дает начальное значение некоторой переменной, которую собираются обновлять позже.

инкрементация или **приращение** (increment): обновление, увеличивающее значение переменной (часто на 1).

декрементация или **отрицательное приращение** (decrement): обновление, которое уменьшает значение переменной.

итерация или **повторение** (iteration): повторяющееся выполнение набора инструкций, использующее либо рекурсивный вызов функции, либо цикл.

бесконечный цикл (infinite loop): цикл, в котором условие для его завершения никогда не выполняется.

7.9 Упражнения

Упражнение 7.3 Для испытания работы алгоритма извлечения квадратного корня, приведенного в этой главе, вы можете сравнить его со встроенной функцией `math.sqrt`. Напишите функцию `test_square_root`, которая выводит таблицу, подобную этой:

```
1.0   1.0              1.0              0.0
2.0   1.41421356237    1.41421356237    2.22044604925e-16
3.0   1.73205080757    1.73205080757    0.0
4.0   2.0              2.0              0.0
5.0   2.2360679775     2.2360679775     0.0
6.0   2.44948974278    2.44948974278    0.0
7.0   2.64575131106    2.64575131106    0.0
8.0   2.82842712475    2.82842712475    4.4408920985e-16
9.0   3.0              3.0              0.0
```

Здесь первая колонка это само число *a*; вторая колонка это квадратный корень числа *a*, вычисленный с помощью функции из упражнения 7.2; третья колонка это квадратный корень, вычисленный встроенной функцией `math.sqrt`; четвертая колонка это модуль значения разности между двумя приближениями.

Упражнение 7.4 Встроенная функция `eval` берет строку и вычисляет ее так, как это делает интерпретатор Python. Например:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.23606797749979
>>> eval('type(math.pi)')
<class 'float'>
```

Напишите функцию, под названием `eval_loop`, которая выводит приглашение пользователю, принимает от него данные и выводит результат, используя `eval`.

Она должна продолжать выполняться до тех пор, пока пользователь не введет слово `'done'`, а затем она должна вернуть значение последнего вычисленного выражения.

Упражнение 7.5 Замечательный математик Srinivasa Ramanujan обнаружил бесконечный ряд[15], который можно использовать для вычисления числа π:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Напишите функцию `estimate_pi`, использующую данную формулу для вычисления приближенного значения π, и возвращающую затем результат. Она должна использовать цикл `while` для вычисления элементов суммы, пока последний элемент не будет меньше, чем `1e-15` (это запись 10^{-15} на языке Python). Вы можете проверить результат, сравнив его с `math.pi`.

Вы также можете посмотреть на мое решение: `thinkpython.com/code/pi.py`.

[15] См. `wikipedia.org/wiki/Pi`

Глава 8
Строки

8.1 Строка как последовательность

Строка представляет из себя **последовательность** (sequence) символов. Вы можете получить доступ к символам, используя прямоугольные скобки:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Во второй строке символ номер 1 из `fruit` присваивается переменной `letter`.

Выражение в квадратных скобках называется **индексом** (index). Он указывает на то, какой символ из последовательности вам нужен, отсюда и его название.

Но, возможно, это не совсем то, что вы ожидали:

```
>>> print(letter)
a
```

Большинство людей полагает, что слово `'banana'` начинается с буквы 'b', не с буквы 'a'. Но для компьютерного специалиста индекс показывает величину смещения от начала строки, а смещение первого символа равняется 0.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

Таким образом, `b` – нулевой символ, `a` – первый, `n` – второй и т.д.

В качестве индекса вы можете использовать любое выражение, включая переменные и операторы, но значение этого выражения должно оставаться целым, иначе вы получите сообщение об ошибке:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 Функция `len`

`len` это встроенная функция, возвращающая число символов в строке:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Чтобы получить доступ к последнему символу строки, вам, вероятно, захочется сделать что-то подобное следующему:

```
>>> length = len(fruit)       # длина строки
>>> last = fruit[length]      # последний символ
IndexError: string index out of range
```

Причиной того, что мы получаем `IndexError`, является то, что в `'banana'` нет символа с индексом 6. Т.к. мы начинаем считать с нуля, все шесть букв пронумерованы цифрами от 0 до 5. Поэтому для доступа к последнему символу мы должны вычесть 1 из длины `length`:

```
>>> last = fruit[length-1]
>>> print (last)
a
```

Вы можете использовать также и отрицательные индексы, которые начинают отсчет с конца строки. Выражение `fruit[-1]` означает последний символ, `fruit[-2]` – второй от конца и т.д.

8.3 Перебор элементов строки с помощью цикла `for`

Есть множество вычислений, которые заключаются в просматривании символов строки один за одним. Часто они начинаются с самого начала, выбирают каждый символ по очереди, делают что-либо с ним, и продолжаются, пока не достигнут конца строки. Такой вид операций называется **перебор** (traversal) элементов. Одним из способов перебора элементов заключается в использовании цикла `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Этот цикл перебирает всю строку и отображает каждый символ на отдельной строке. Условием цикла является `index < len(fruit)`, поэтому когда `index` равняется длине строки, условие не будет соблюдаться, и тело цикла не будет выполняться. Последним обработанным символом будет тот, у которого индекс равен `len(fruit)-1`, т.е. последний символ строки.

Упражнение 8.1 Напишите функцию, которая берет строку в качестве аргумента и отображает ее задом наперед. Подсказка: помните, что для того, чтобы следующий за функцией `print` вывод осуществлялся на той же самой строке без переноса на новую, вам нужно использовать дополнительный параметр `end=""`. Например:

```
print('hello ', end="")
print('world!')
```

Выведет на экран следующее:

```
hello world!
```

Другим способом осуществления перебора является использование цикла `for`:

```
for char in fruit:
    print(char)
```

При выполнении этого цикла каждый новый символ присваивается переменной `char`. Цикл продолжается до тех пор, пока не закончатся все символы в строке.

В следующем примере показано, как использовать сцепление строк и цикл `for` для генерации алфавитных последовательностей. В книге Robert McCloskey *Make Way for Ducklings* встречаются имена утят Jack, Kack, Lack, Mack, Nack, Ouack, Pack и Quack. Следующий цикл выводит эти имена в алфавитном порядке.

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

Вывод выглядит следующим образом:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Вообще-то, это не совсем правильно, т.к. имена Ouack и Quack выводятся с ошибкой.

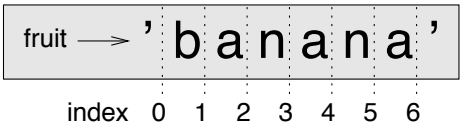
Упражнение 8.2 Модифицируйте программу, чтобы исправить эту ошибку.

8.4 Срезы строк

Часть строки называется **срезом** (slice). Выбор среза подобен выбору символа:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

Оператор `[n:m]` возвращает часть строки, начинающуюся с `n`-ного символа и заканчивающуюся m-ым, включая первый и исключая второй. Такое поведение не совсем интуитивно понятно. Но вам будет проще, если вы будете рассматривать индексы, указывающие на места *между* символами, а не на сами символы, как на следующей диаграмме:



Если вы не укажете первый индекс (тот, что стоит до двоеточия), срез начнется от самого начала строки. Если же вы не укажете последний индекс, то срез будет продолжаться до конца строки:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Если первый индекс больше либо равен второму, то в результате вы получите **пустую строку** (empty string), обозначаемую двумя апострофами:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Пустая строка не содержит символов, и ее длина равна 0, но во всем остальном это такая же строка, как и прочие.

Упражнение 8.3 Переменная `fruit` содержит некоторую строку. Что, в таком случае, означает выражение `fruit[:]`?

8.5 Строки принадлежат к неизменяемым типам данных

Возможно, вы думаете, что можете использовать оператор `[]` с левой стороны операции присваивания, чтобы изменить определенный символ в строке, например, так:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

Сообщение об ошибке гласит: "Объект не поддерживает операцию присваивания для элементов". В данном случае "объект" это строка, а "элемент" это тот символ, которому вы пытались присвоить новое значение. На данной стадии будем считать, что *объект* это то же самое, что и значение, но позже мы уточним определение. А *элемент* это одно из значений в последовательности.

Причина ошибки, которую мы получили, кроется в том, что строки принадлежат к **неизменяемым типам данных** (immutable), что означает, что вы не можете изменить существующую строку. Самое большее, что вы можете сделать, это создать другую строку – модифицированную версию оригинальной:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
```

```
Jello, world!
```

В данном примере символ `J` сцепляется со срезом строки `greeting`. На первоначальную строку это никак не влияет.

8.6 Поиск

Что делает следующая функция?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

В некотором смысле, `find` (*найти*) является противоположным действием по отношению к оператору `[]`. Вместо того, чтобы взять индекс и извлечь соответствующий символ, эта функция берет символ и находит соответствующий ему индекс. Если же такой символ не найден, то функция возвращает `-1`.

Это первый раз, когда мы встречаемся с инструкцией `return` внутри цикла. Если выполняется условие `word[index] == letter`, функция немедленно выходит из цикла и возвращается.

Если же данный символ не содержится в слове, то цикл выполняется до конца, и выполняется вторая инструкция `return`, возвращающая `-1`.

Такой вид вычислений – перебор некоей последовательности с возвращение определенного значения тогда, когда найдено то, что нужно, – называется **поиск** (search).

Упражнение 8.4 Измените функцию `find` таким образом, чтобы у нее был третий параметр – индекс в слове, с которого нужно начинать поиск.

8.7 Цикл и подсчет

Следующая программа подсчитывает, сколько раз буква '`a`' встречается в строке:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Эта программа демонстрирует другой вид вычислений, называемых **счетчиком** (counter). Переменная `count` вначале инициализируется значением `0`, а затем ее значение увеличивается каждый раз, как будет найдена новая буква '`a`'. Когда цикл заканчивается, `count` содержит результат – общее количество букв '`a`'.

Упражнение 8.5 Инкапсулируйте этот код в функцию `count`, и затем сделайте ее более общей, чтобы она принимала строку и символ в качестве своих аргументов.

Упражнение 8.6 Перепишите эту функцию таким образом, чтобы вместо перебора строки она использовала бы версию нашей функции `find` с тремя параметрами из предыдущего раздела.

8.8 Методы строк `string`

Метод (method) в чем-то похож на функцию – он берет аргументы и возвращает значение – но его синтаксис отличается. Например, метод `upper` берет строку и возвращает другую строку, у которой все буквы в верхнем регистре.

Вместо синтаксиса вызова функции `upper(word)` используется синтаксис вызова метода `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

Такая форма точечной записи требует имя метода (в данном случае `upper`) и имя строки, к которой этот метод применяется (в данном случае `word`). Пустые скобки означают, что метод используется без параметров.

Вызов (invocation) **метода** всегда осуществляется на определенный объект. В данном случае мы можем сказать, что мы вызвали метод `upper` на объект `word`.

Кстати, у объектов типа `string` имеется встроенный метод `find`, который делает то же самое, что и одноименная функция, которую мы написали:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

В этом примере мы вызвали метод `find` на объект `word` и передали символ `'a'`, который мы ищем, в качестве параметра.

На самом деле встроенный метод `find` более универсален, чем наша функция. Он может искать подстроки, а не только одиночные символы:

```
>>> word.find('na')
2
```

В качестве второго аргумента он может брать индекс, с которого начинать поиск:

```
>>> word.find('na', 3)
4
```

В качестве третьего аргумента он может брать индекс, где прекращать поиск:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

В данном случае поиск возвращает ошибку, т.к. `'b'` не содержится между индексами `1` и `2` (не включая `2`).

Упражнение 8.7 У объектов `string` имеется встроенный метод `count`, который подобен нашей функции из предыдущего упражнения. Прочитайте документацию к этому методу и используйте его для подсчета количества букв `'a'` в слове 'banana'.

8.9 Оператор `in`

Команда `in` является булевым оператором, который берет две строки и возвращает `True`, если первая содержится как подстрока во второй:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

Например, следующая функция выводит все символы из `word1`, которые содержатся также и в `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

Если удачно подобрать имена переменных, то программу на Python иногда можно читать почти как естественный текст на английском языке. Вы можете прочитать этот цикл следующим образом: “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Вот что у вас получится, если вы сравните слова 'apples' и 'oranges':

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 Сравнение строк

Операции сравнения работают также и на строках. Посмотреть на два слова, равны ли они друг другу, можно так:

```
if word == 'banana':
    print('Все правильно, bananas.')
```

Другие операции сравнения можно использовать для того, чтобы вывести слова в алфавитном порядке:

```
if word < 'banana':
    print('Ваше слово, ' + word + ', по алфавиту стоит до banana.')
elif word > 'banana':
    print('Ваше слово, ' + word + ', по алфавиту стоит после banana.')
else:
    print('Все правильно, bananas.')
```

Буквы верхнего и нижнего регистра Python сортирует не так, как это делают люди. У него все буквы ВЕРХНЕГО регистра идут впереди всех букв нижнего регистра, поэтому:

```
Ваше слово, Pineapple, по алфавиту стоит до banana.
```

Для решения этой проблемы обычно используют конвертацию строк в стандартный формат, например, все в нижнем регистре, а уже затем совершается их сравнение.

8.11 Отладка

Используя индексы для перебора значений в последовательности не так-то и просто определить правильно начальные и конечные значения. Вот функция, которая должна сравнивать два слова, и возвращать True, если одно из них является зеркальным отображением другого, но в ней есть две ошибки:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
    i=0
    j = len(word2)
    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
    return True
```

Первое if проверяет, имеют ли оба слова одинаковую длину. Если нет, то мы можем сразу вернуть правильный результат False. В противном случае выполняется остальная часть функции, и мы знаем, что оба слова имеют одинаковую длину. Это образец того, как используется проверка условия из раздела 6.8.

i и j – индексы: i перебирает word1 в прямом направлении, а j перебирает word2 в обратном. Если мы обнаружим два символа, которые не совпадают, то мы можем немедленно вернуть False. Если же весь цикл закончится, и все символы совпадают, то мы возвращаем True.

Если мы запустим эту функцию с двумя словами "pots" и "stop", то, вероятно, ожидаем получить True, но нас ожидает ошибка индексирования:

```
>>> is_reverse('pots', 'stop') ...
File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

Для обнаружения такого вида ошибок я бы, первым делом, распечатал значения индексов непосредственно перед строками, в которых возникает ошибка.

```
while j > 0:
    print(i, j) # print here
    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

Теперь, если я снова запущу программу, у меня будет больше информации:

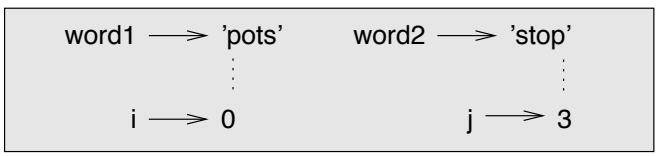
```
>>> is_reverse('pots', 'stop')
04
...
IndexError: string index out of range
```

При первом прохождении цикла значение `j = 4`, что является ошибкой, т.к. у слова "pots" нет элемента с таким индексом. Индекс последнего символа равняется 3, поэтому начальное значение j должно быть `len(word2)-1`.

Если я исправлю ошибку и снова запущу программу, я получу следующий результат:

```
>>> is_reverse('pots', 'stop')
3
2
1
True
```

В этот раз мы получаем правильный ответ, но, похоже, что цикл выполнился только 3 раза, что подозрительно. Для того, чтобы лучше понять, что происходит, полезно нарисовать диаграмму состояния. Во время первого прохода фрейм для `is_reverse` выглядит так:



Для наглядности я подрисовал пунктирные линии, чтобы было видно, какие символы в word1 и word2 соответствуют индексам i и j.

Упражнение 8.8 Продолжите эту диаграмму, выполняя программу на бумаге, изменяя значения i и j во время прохождения цикла. Найдите и исправьте вторую ошибку в этой функции.

8.12 Словарь терминов

объект (object): нечто, на что может ссылаться переменная; на данной стадии вы можете под "объектом" понимать "значение" и наоборот.

последовательность (sequence): упорядоченный набор данных; другими словами, набор значений, где каждое из них определяется по его индексу (целое число).

элемент (item): одно из значений в последовательности.

индекс (index): целое число, используемое для доступа к элементу в последовательности (например, доступ к символу в строке).

срез (slice): часть строки, заключенная между двух индексов (начало и конец среза).

пустая строка (empty string): строка, которая не содержит ни одного символа, и чья длина равна `0`; отображается двумя апострофами (`''`).

неизменяемый тип данных (immutable): свойство последовательности, при котором ее элементам нельзя присваивать значения (можно только считывать).

перебор (traverse): проход по элементам последовательности с выполнением похожих операций над каждым элементом.

поиск (search): прием в программировании, при котором последовательность перебирается до тех пор, пока не будет найдено то, что необходимо, или станет ясно, что этого там нет.

счетчик (counter): переменная, используемая для подсчета чего-либо; как правило инициализируется нулем, а затем ее значение увеличивается.

метод (method): функция, ассоциируемая с некоторым объектом, которую можно вызвать при помощи точечной записи.

вызов на объект (invocation): вызов (применение) метода к некоторому объекту.

8.13 Упражнения

Упражнение 8.9 Срез строки может принимать третий индекс, означающий "шаг". Шаг 2 означает выбор каждого второго элемента, шаг 3 – каждый третий и т.д.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

Шаг `-1` означает перебор строки в обратном порядке, поэтому `[::-1]` выводит строку в обратном порядке. Используйте такую запись и перепишите функцию `is_palindrom` из раздела 6.6 одной строкой.

Упражнение 8.10 Прочитайте документацию по методам строк `docs.python.org/lib/stringmethods.html`

Попробуйте поэкспериментировать с ними, чтобы понять, как они работают. В частности, методы `strip` и `replace` могут оказаться очень полезными.

В документации используется синтаксис, который может вас смутить. Например, в такой записи `find(sub[, start[, end]])` квадратные скобки означают опциональные (т.е. не обязательные) аргументы. Так что `sub` является необходимым параметром, а `start` опциональным. Но если вы используете и `start`, то тогда `end` становится опциональным.

Упражнение 8.11 Следующие функции *предназначены* для того, чтобы определять, содержит ли строка любые буквы нижнего регистра, но, как минимум, в одной из них содержится ошибка. Выясните, что *на самом деле* делает каждая функция, если ей передается строка в качестве параметра.

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'
```

```
def any_lowercase3(s):
    for c in s:
        flag = c.islower()
            return flag

def any_lowercase4(s):
    flag = False
        for c in s:

flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Упражнение 8.12 ROT13 является слабой формой шифрования, заключающейся в "ротации", т.е. вращении или сдвиге каждой буквы слова на 13 позиций[16]. Сдвиг буквы заключается в перемещении ее по алфавиту, а если необходимо, то и возвращаясь по кругу. Например, 'A' сдвинутое на 3 превращается в 'D', а 'Z', сдвинутое на 1 превращается в 'A'.

Напишите функцию `rotate_word`, принимающую строку и целое число в качестве параметров, которая возвращает новую строку из букв, сдвинутых относительно оригинальных на данное значение.

Например, "`cheer`" сдвинутое на `7` превращается в "`jolly`", а "`melon`", сдвинутое на `-10` превращается в "`cubed`".

Вам может пригодиться встроенная функция `ord`, которая конвертирует символы в их численные значения, а также `chr`, конвертирующая численные значения в символы.

В интернете нередко можно встретить потенциально оскорбительные шутки, зашифрованные алгоритмом ROT13. Если вы не относитесь к тем, кого легко этим оскорбить, то можете попытаться найти такие шутки и раскодировать их.

[16] См. `wikipedia.org/wiki/ROT13`

Глава 9
Углубленное изучение: играем словами

9.1 Чтение списков слов

Для выполнения упражнений этой главы нам понадобится список английских слов. В интернете можно найти много таких списков, но для наших целей мы будем пользоваться списком, составленным Grady Ward для проекта Moby[17], и опубликованный как общественное достояние (public domain). Этот список состоит из 113809 слов и является официальным списком слов, пригодных для составления и решения кроссвордов. В проекте Moby файл со списком слов имеет название 113809of.fic. Я прилагаю копию этого файла в архиве `Swampy`, который я переименовал в `words.txt`.

Это файл в простом текстовом формате, поэтому вы можете открыть его в любом текстовом редакторе, но вы также можете читать его и средствами Python. Встроенная функция `open` принимает имя файла в качестве параметра возвращает **объект `file`**, который вы можете использовать для чтения файла.

```
>>> fin = open('words.txt')
>>> print(fin)
<_io.TextIOWrapper name='words.txt' mode='r' encoding='cp1252'>
```

Имя `fin` часто присваивается объекту `file`, с которого считываются данные. Режим '`r`' указывает на то, что файл открыт только для чтения (reading), в противоположность '`w`' – для записи (writing).

Объект `file` предоставляет несколько методов для чтения содержимого, в том числе `readline`, который считывает символы с из файла, пока не встретит символ новой строки (другими словами, считывает одну строку), и возвращает результат как `string`:

```
>>> fin.readline()
'aa\n'
```

Первым словом в этом списке идет слово "`aa`", что представляет собой вид вулканической лавы. Символ `\n` это символ новой строки, который отделяет это слово от следующего.

Объект `file` следит за тем, где производится чтение. Поэтому если вы вызовете метод `readline` снова, то получите следующее слово:

```
>>> fin.readline()
'aah\n'
```

Да, в английском языке есть слово "`aah`", поэтому не нужно на меня так смотреть. Если же вас раздражают непечатные символы (в данном случае `\n`), то вы можете от них избавиться, применив метод `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print(word)
aahed
```

Объект `file` можно использовать и в цикле. Следующая программа читает файл `words.txt` и выводит каждое слово на отдельной строке:

```
fin = open('words.txt')
    for line in fin:
        word = line.strip()
        print(word)
```

Упражнение 9.1 Напишите программу, которая читает список `words.txt` и выводит только те слова, которые длиннее 20 символов (не учитывая символ новой строки).

---

[17] `wikipedia.org/wiki/Mobi_Project`

9.2 Упражнения

К этим упражнениям есть ответы в следующем разделе. Но вы должны, как минимум, попытаться выполнить каждое из них прежде, чем заглядывать в ответ.

Упражнение 9.2 В 1939 году Ernest Vincent Wright опубликовал роман *Gadsby*, содержащий 50 тысяч слов, ни в одном из которых не было буквы "e". Поскольку буква "e" в английском языке встречается очень часто, то сделать это было не так-то и легко.

Да что там роман, даже одно предложение без этой буквы составить трудно, особенно поначалу. Но с часами тренировки и внимательностью приходит необходимый навык.

Ну ладно, довольно об этом.

Напишите функцию `has_no_e`, которая возвращает `True`, если данное слово не содержит "`e`".

Модифицируйте предыдущую программу так, чтобы она выводила только слова, в которых не содержится буквы "`e`", и подсчитайте процентное соотношение таких слов к их общему количеству.

Упражнение 9.3 Напишите функцию `avoids`, которая берет в качестве параметров слово и строку запрещенных букв, и выдает `True`, если слово не содержит ни одной запрещенной буквы.

Модифицируйте эту программу так, чтобы она выдавала пользователю приглашение, ожидая от него ввода строки с запрещенными буквами, а затем выводила только те слова, в которых эти буквы не используются. Можете ли вы найти такую комбинацию из 5 запрещенных букв, которые выводят наименьшее число слов в вашей программе?

Упражнение 9.4 Напишите функцию `uses_only`, которая принимает слово и строку букв, и, затем, выдает `True`, если это слово содержит только буквы из этой строки. Можете ли вы составить какое-нибудь английское предложение, используя только буквы `acefhlo`, кроме *Hoe alfalfa*?

Упражнение 9.5 Напишите функцию `uses_all`, которая берет слово и строку требуемых букв, и возвращает `True`, если слово содержит все необходимые буквы хотя бы один раз. Сколько таких слов, которые содержат все гласные `aeiou`? А как насчет `aeiouy`?

Упражнение 9.6 Напишите функцию `is_abecederian`, которая возвращает `True`, если все буквы в слове следуют алфавитному порядку (повторяющиеся буквы тоже допускаются). Сколько вообще таких слов?

9.3 Поиск

У всех упражнений в предыдущем разделе есть нечто общее между собой, а именно: их можно выполнить с использованием техники поиска, с которой мы познакомились в разделе 8.6. Вот простейший пример:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

Цикл `for` перебирает все символы в `word`. Если мы встретим "`e`", то мы можем немедленно возвратить `False`; в противном случае, мы идем к следующему символу. Если же цикл закончится нормально, то это означает, что мы не встретили буквы "`e`" в этом слове, поэтому можно возвращать `True`.

Функция `avoids` более универсальна, чем `has_no_e`, хотя имеет ту же самую структуру:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Мы можем возвратить `False` как только встретим запрещенную букву; если же мы дойдем до конца цикла, мы возвращаем `True`.

Функция `uses_only` похожа на предыдущую за единственным исключением – в ней условие противоположно:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Вместо списка запрещенных букв у нас имеется список допустимых букв. Если мы находим в `word` букву, которая не содержится в `availble`, мы можем вернуть `False`.

Функция `uses_all` похожа на предыдущую, за исключением того, что мы меняем роли слова и строки символов местами:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Вместо того, чтобы перебирать буквы в `word`, мы перебираем требуемые буквы. Если любая из требуемых букв не появится в `word`, мы можем вернуть `False`.

Если вы действительно рассуждали как компьютерный специалист, то могли заметить, что `uses_all` является частным случаем решенной прежде проблемы, поэтому вы могли бы написать следующее:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Это пример метода разработки программ, называемый **распознание проблемы** (problem recognition), означающий, что вы распознаете проблему, над которой вы работаете, как частный случай решенной прежде проблемы, и применяете разработанное прежде решение.

9.4 Циклы с индексами

В предыдущем разделе я использовал цикл `for` в функциях только потому, что мне нужно было только взглянуть на символы в строках. Я не собирался их каким-то образом обрабатывать. Мне не нужно было использовать их индексы.

В функции `is_abecederian` нам уже нужно сравнивать соседние буквы, что несколько трудно реализовать при помощи цикла `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

В качестве альтернативы можно использовать рекурсию:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Но можно использовать и цикл с `while`:

```
def is_abecedarian(word):
    i=0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

Цикл начинается с `i=0` и заканчивается, когда `i=len(word)-1`. Каждый раз при прохождении цикла сравнивается `i`-й символ (его можно рассматривать в качестве текущего символа) с `i+1` символом (следующий символ).

Если следующий символ меньше (т.е. в алфавите стоит раньше) текущего, это значит, что мы обнаружили нарушение алфавитного порядка, поэтому возвращаем `False`.

Если мы достигнем конца цикла без проблем, это значит, что наше слово прошло проверку. Для того, чтобы убедиться в том, что цикл заканчивается корректно, рассмотрите, для примера, слово `'flossy'`. Длина этого слова равна `6`, поэтому последний раз цикл исполняется, когда `i=4`, т.е. индекс предпоследней буквы. Во время этого последнего прохода предпоследняя буква, `s`, сравнивается с последней, `y`, что как раз то, что нам нужно.

Вот версия `is_palindrome` (см. упражнение 6.6), использующая два индекса; один начинается с самого начала и идет вперед; другой начинается с самого конца и идет назад:

```
def is_palindrome(word):
    i=0
    j = len(word)-1
    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
    return True
```

Если же вы заметили, что это частный случай прежде решенной проблемы, вы можете записать:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Предполагается, что вы делали упражнение 8.8.

9.5 Отладка

Тестирование программ – нелегкая задача. Функции этой главы сравнительно легко протестировать, потому что вы можете вручную проверить результаты. Но даже при все этом, выбор набора слов, который мог бы выявить все возможные ошибки, находится где-то между "трудно" и "невозможно".

Если взять в качестве примера `has_no_e`, то очевидно, что есть два случая для проверки: слова, в которых есть `'e'`, должны возвращать `False`, а слова, в которых `'e'` нет, должны возвращать `True`. В обоих случаях у вас не должно возникать особых проблем.

Но и в каждом из этих случаев есть свои менее очевидные ситуации, которые необходимо проверить. Среди тех слов, в которых есть `'e'`, вы должны проверять те, которые начинаются на `'e'`, заканчиваются на `'e'`, и у которых `'e'` встречается где-то посередине. Вы должны испытать длинные слова, короткие слова и очень короткие слова, такие, как, например, пустую строку. Пустая строка является примером **особого случая** (special case). Это один из не вполне очевидных случаях, где может скрываться ошибка.

Кроме тестов со словами, которые вы задаете сами, вы можете испытать вашу программу списком слов типа `words.txt`. Просматривая вывод, вы можете обнаружить ошибку, если она имеется, но будьте внимательны: вы можете заметить один вид ошибки (слова, которых не должно там быть, а они есть) и пропустить другую (слова, которые должны там быть, а их нет).

В общем случае, тестирование может помочь вам отыскать ошибки, но совсем не просто придумать набор подходящих тестов. Даже если вы это и сделаете, вы не можете до конца быть уверены, что в вашей программе нет ошибок.

"Тестирование программы может выявить в ней ошибку, но не может доказать, что ошибок в ней нет." (c) Edsger W. Dijkstra.

9.6 Словарь терминов

объект `file` (file object): значение, представляющее собой открытый файл.

распознание проблемы (problem recognition): способ решения проблем, выражая их как частный случай прежде решенной проблемы.

особый случай (special case): тест с нетипичным или неочевидным значением (который труднее правильно обработать).

9.7 Упражнения

Упражнение 9.7 Это упражнение основано на загадке из радиопередачи *Car Talk*:[18]

"Найдите слово с тремя последовательными двойными буквами. В качестве примера я могу привести вам пару слов, которые почти подходят, но не совсем. Одно из них это слово `committee`: `committee`. Оно бы подходило, если бы не было в нем `i` посередине. Другое слово это `Mississippi`: `Mississippi`. Если бы вы убрали две `i` посередине, то оно тоже подошло бы. Но все же есть такое слово, в котором имеются последовательно три пары букв. Я знаю только одно такое слово. Конечно, их может быть и 500, но мне известно только одно. Что это за слово?"

Напишите программу для нахождения этого слова. Мое решение находится здесь: `thinkpython.com/code/cartalk.py`

Упражнение 9.8 Это другой пример загадки из *Car Talk*:[19]

"Однажды, ведя машину по дороге, я взглянул на счетчик километров. Как большинство спидометров он показывал 6-значные числа, т.е. только мили. Например, если бы мой спидометр показывал бы `300000` миль, то я бы увидел `300000`.

Но то, что я увидел тогда, было достаточно интересным. Я заметил, что последние 4 цифры образовывали палиндром. Например, `5445` это палиндром. Поэтому мой спидометр мог показывать `315445`.

Через оду милю уже 5 последних цифр образовывали палиндром. Например, это могло быть `365456`. Спустя еще одну милю 4 средние цифры из 6 были палиндромом. А теперь вы готовы услышать следующее? Через еще одну милю уже все 6 цифр образовывали палиндром.

Вопрос: что показывал спидометр, когда я взглянул на него в первый раз?"

Напишите программу, которая тестирует все 6-значные числа и выводит любое число, которое отвечает всем этим требованиям. Вы можете сравнить с моим решением на `thinkpython.com/code/cartalk.py`.

Упражнение 9.9 Вот еще одна загадка из *Car Talk*, которую можно решить при помощи поиска:[20]

"Недавно я посещал свою маму и внезапно осознал, что те две цифры, из которых состоит мой возраст, если поменять их местами, показывают ее возраст. Например, если ей 73, то мне 37. Нам стало интересно, как часто в течение нашей жизни такое происходило, но разговор пошел о других темах, и мы к этому уже не возвращались.

---

[18] `www.cartalk.com/content/puzzler/transcripts/200725`

[19] `www.cartalk.com/content/puzzler/transcripts/200803`

[20] `www.cartalk.com/content/puzzler/transcripts/200813`

Придя же домой я выяснил, что в нашей жизни это происходило уже 6 раз. Также я выяснил, что если нам повезет, то это произойдет еще раз через несколько лет, а если совсем повезет, то это случится и еще один раз. Другими словами, это может произойти 8 раз. Вопрос: сколько мне сейчас лет?"

Напишите программу, отыскивающую решение этой загадки. Подсказка: вам может пригодиться метод строк `zfill`.

Смотрите мое решение на `thinkpython.com/code/cartalk.py`.

Глава 10
Списки

10.1 Список как последовательность

Подобно строкам, **списки** (list) являются последовательностями значений. В случае со строками значениями являются символы; в списках же они могут быть любого типа. Значения в списке называются **элементами** (elements, items).

Есть несколько способов для создания нового списка; простейший из них это заключение его элементов в квадратные скобки `[` и `]`:

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

В первом примере мы видим список, состоящий из четырех целых чисел. Во втором примере это список из трех строк. Элементы списка не обязательно должны иметь одинаковый тип. Следующий список содержит строку, дробное число, целое число и (внимание!) другой список:

```
['spam', 2.0, 5, [10, 20]]
```

Список внутри другого списка называется **вложенным** (nested).

Список, не содержащий элементов, называется пустым списком; его можно создать при помощи пустых скобок, `[]`.

Как вы, вероятно, и ожидали, вы можете присваивать спискам имена переменных:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 Списки принадлежат к изменяемым типам данных

Синтаксис для доступа к элементам списка точно такой же, как и для доступа к символам строки – оператор `[]`. Выражение внутри скобок определяет индекс. Помните, что индекс начинается с `0`:

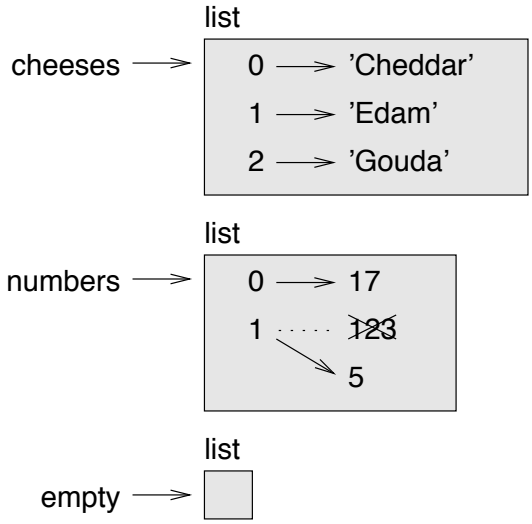
```
>>> print(cheeses[0])
Cheddar
```

В противоположность строкам, списки принадлежат к изменяемым типам данных. Когда оператор `[]` появляется с левой стороны операции присваивания, он определяет тот элемент списка, которому осуществляется присваивание.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Элемент списка `numbers` с индексом `1`, у которого было значение `123`, теперь получил значение `5`.

Вы можете представлять себе список в виде отношения между индексами и элементами. Такое отношение называется **отображением** (mapping); каждый из индексов "отображается" на один из элементов. Вот диаграмма состояния, показывающая `cheeses`, `numbers` и `empty`:



Списки представлены прямоугольниками со словами "`list`" снаружи и элементами списка внутри. `cheeses` ссылается на список с тремя элементами с индексами `0`, `1` и `2`. `numbers` содержит два элемента; диаграмма показывает, что значение второго элемента было изменено с `123` на `5`. `empty` ссылается на список, в котором нет элементов.

Индексы списков работают таким же образом, как и индексы строк:

- Любое целое выражение может использоваться в качестве индекса.
- Если вы попытаетесь прочитать или записать несуществующий элемент, вы получите ошибку `IndexError`.
- Если индекс имеет отрицательное значение, то отсчет идет от конца списка к началу.

Оператор `in` также работает и со списками:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 Перебор элементов списка

Чаще всего перебор элементов списка осуществляется с помощью цикла `for`. Синтаксис такой же, как и для строк:

```
for cheese in cheeses:
    print(cheese)
```

Такой список подходит, если вам нужно только прочитать элементы списка. Если же вам нужно записать или обновить значения элементов, то вам потребуются индексы. Часто для этого используется комбинация функций `range` и `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Этот цикл перебирает весь список и обновляет каждый его элемент. `len` возвращает количество элементов списка. `range` возвращает список индексов от `0` до `n-1`, где n это длина списка. Каждый раз при выполнении цикла `i` получает индекс следующего элемента. Операция присваивания в теле цикла использует `i` для чтения старого значения элемента и присваивания ему нового значения.

Тело цикла for никогда не будет выполняться, если его применить к пустому списку:

```
for x in []:
    print('Это никогда не случится.')
```

Хотя один список может содержать другой список, вложенный список считается как один элемент. Длина этого списка равна 4:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 Операции со списками

Оператор + сцепляет списки:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Сходным образом, оператор * повторяет список данное количество раз:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Первый пример повторяет [0] четыре раза; второй пример повторяет список [1,2,3] три раза.

10.5 Срезы списков

Операции со срезами работают и на списках:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Если вы опустите первый индекс, то срез начнется с самого начала. Если опустите второй, то срез будет продолжаться до конца. Если опустите оба, то срез будет копией всего списка.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Поскольку списки можно изменить, то часто бывает полезно сделать их копию перед тем, как совершать операции, которые их затрагивают.

Оператор среза, находящийся с левой стороны операции присваивания, может обновить сразу несколько элементов:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 Методы списков

Python предоставляет несколько методов, работающих со списками. Например, append добавляет новый элемент к концу списка:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

Метод `extend` берет список в качестве аргумента и добавляет все его элементы:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

В данном примере список `t2` остается неизменным.

Метод `sort` сортирует все элементы списка по порядку, начиная с меньших значений, и заканчивая большими:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Все методы списков являются нерезультативными функциями. Они модифицируют список и возвращают `None`. Если вы случайно напишите `t = t.sort()`, то, вероятно, будете удивлены полученным результатом.

10.7 Отображение, фильтрация и сокращение

Для того, чтобы суммировать все числа, содержащиеся в списке, можно воспользоваться следующим циклом:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

Вначале переменная `total` инициализируется нулем. Каждый раз при прохождении цикла `x` получает один элемент из списка. Оператор += предоставляет краткий способ обновления переменной. Это называется **операцией приращения** (augment assignment statement):

```
total += x
```

равнозначно

```
total = total + x
```

При выполнении цикла в `total` накапливается сумма элементов; когда переменная используется таким образом, она называется **аккумулятором** (accumulator)**.**

Поскольку суммирование элементов списка используется достаточно часто, Python предоставляет встроенную функцию, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Операция, подобная этой, при которой последовательность элементов преобразуется в единое значение, иногда называется **сокращением** (reduce)**.**

Иногда вам необходимо перебрать один список с одновременным построением другого. Например, следующая функция берет список строк, и возвращает новый список, содержащий строки, у которых первый буквы заглавные:

```
def capitalize_all(t):
    res = []
        for s in t:
            res.append(s.capitalize())
    return res
```

Переменная `res` инициализируется пустым списком; каждый раз при прохождении цикла мы добавляем к нему новый элемент. Поэтому `res` это другой вид аккумулятора.

Операция, подобная `capitalize_all` иногда называется **отображением** (map), т.к. она "отображает" функцию (в данном случае это метод `capitalize`) на каждый элемент последовательности.

Другим примером часто используемой операции является выбор только определенных элементов из списка. Например, следующая функция берет список строк в качестве параметра, и возвращает только те из них, все буквы которых – заглавные:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` это метод строк, который возвращает `True`, если строка состоит только из букв верхнего регистра.

Операция, подобная `is_upper`, называется **фильтрацией** (filter), потому что она отбирает лишь некоторые элементы, а остальные отфильтровывает.

Наиболее частые операции со списками можно выразить при помощи комбинации операций отображения, фильтрации и сокращения. Поскольку эти операции используются довольно часто, в Python имеются средства для их осуществления, включая встроенную функцию `map` (отображение), а также операцию, которую можно назвать "заполнение списка" (list comprehension), но мы не будем ее здесь обсуждать.

Упражнение 10.1 Напишите функцию, которая принимает список с числами и возвращает другой список с элементами, представляющими из себя их кумулятивную сумму; другими словами, новый список, где `i`-й элемент равен сумме первых `i+1` элементов первого списка. Например, кумулятивная сумма `[1,2,3]` равна `[1,3,6]`.

10.8 Удаление элементов

Есть несколько способов для удаления элементов из списка. Если вам известен индекс элемента, можно использовать метод `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

Метод `pop` модифицирует список и возвращает удаляемый элемент. Если вы не укажете индекс, он удаляет и возвращает последний элемент.

Если вам не нужно удаляемое значение, вы можете использовать оператор `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Если вам известен элемент, который вы собираетесь удалить, а не его индекс, вы можете воспользоваться `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

Значение, возвращаемое `remove`, равно `None`.

Для удаление более одного элемента, вы можете объединить `del` и срез:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
```

```
>>> print(t)
['a', 'f']
```

Как обычно, срез выбирает диапазон элементов, указанных в нем, за исключением последнего.

10.9 Списки и строки

Строка является последовательностью символов, а список – последовательностью значений, но список символов это не то же самое, что и строка. Для конвертации строки в список символов можно воспользоваться функцией `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Поскольку `list` является именем встроенной функции, вы не должны создавать переменную с таким именем (хотя это и допустимо). Также я избегаю использования буквы `l` (латинская эль), потому что ее можно спутать с `1` (цифра один). Вот почему я использую `t`.

Функция `list` разбивает строку на индивидуальные символы. Если же вы хотите разбить строку на слова, вы можете использовать метод `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
```

Этот метод принимает опциональный аргумент, называемый **разделителем** (delimiter), который определяет, какой символ используется для разделения слов. В следующем примере в качестве разделителя используется дефис:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` это метод, обратный `split`. Он берет список строк и соединяет их элементы. `join` это метод строк, поэтому вы должны вызывать его на разделитель и передавать ему список в качестве параметра:

```
>>> t = ['pining', 'for', 'the', 'fjords'
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

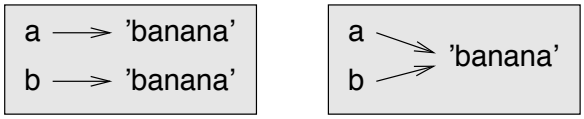
В данном случае разделителем являлся пробел, поэтому `join` и вставлял пробелы между словами. Если вы хотите соединить слова не оставляя пробелов между ними, вы можете использовать пустую строку, `''`, в качестве разделителя.

10.10 Объекты и значения

Если мы выполним следующие операции присваивания:

```
a = 'banana'
b = 'banana'
```

то мы знаем, что как `a`, так и `b` ссылаются на строку, но мы не знаем, ссылаются ли они на *ту же самую* строку. Существует две возможности:



В первом случае `a` и `b` ссылаются на различные объекты, которые имеют одинаковое значение. Во втором случае они ссылаются на тот же самый объект.

Чтобы проверить, ссылаются ли две переменные на тот же самый объект, вы можете использовать оператор `is`:

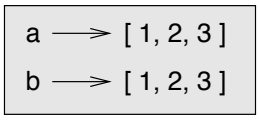
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

В данном примере Python создает лишь один строковый объект, и обе переменные, `a` и `b`, ссылаются на него, т.е. для строк верной будет вторая диаграмма.

Но если вы создаете два одинаковых списка, то вы создаете два разных объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Поэтому диаграмма состояния выглядит следующим образом:



В данном случае мы можем сказать, что два списка **эквивалентны** (equivalent), т.к. они состоят из одинаковых элементов, но они не **идентичны** (identical), потому что они не являются одним и тем же объектом. Если два объекта идентичны, то они также и эквивалентны, но если они эквивалентны, но они не обязательно идентичны.

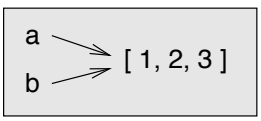
До сих пор мы использовали понятия "объект" и "значение" не делая между ними разницы, но теперь более точным будет говорить, что объект имеет значение. Если вы выполните `[1,2,3]`, то получите объект `list`, значениями которого будет последовательность целых чисел. Если другой объект `list` состоит из точно таких же элементов, мы говорим, что он имеет точно такое же значение, но не является тем же самым объектом.

10.11 Создание синонимов

Если a ссылается на некий объект, и вы выполняете присваивание `b = a`, тогда обе переменные ссылаются на тот же самый объект.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Диаграмма состояния выглядит следующим образом:



Связывание переменной с объектом называется **ссылкой** (reference). В данном примере имеется две ссылки на один объект.

У объекта с более чем одной ссылкой имеется более одного имени, поэтому говорят, что у объекта имеются **синонимы** (aliase).

Если объект, имеющий синонимы, может изменяться, то изменения, примененные к одному синониму влияют на другой синоним:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Хотя такое поведение и может оказаться полезным, оно, тем не менее, легко может привести к ошибкам. В общем случае, безопаснее воздерживаться от использования синонимов, когда вы работаете с изменяемыми объектами.

С неизменяемыми объектами, такими, как, например, строки, использование синонимов не является проблемой. В данном примере:

```
a = 'banana'
b = 'banana'
```

почти никогда не бывает разницы, ссылаются ли `a` и `b` на одну и ту же строку или нет.

10.12 Аргументы списков

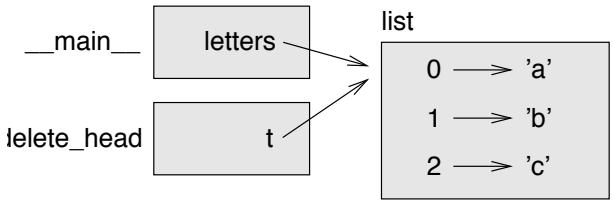
Когда вы передаете список в функцию, она получает ссылку на этот список. Если функция изменяет этот список, то изменения становятся видны и в программе, вызвавшей эту функцию. Например, функция `delete_head` удаляет первый элемент списка:

```
def delete_head(t):
    del t[0]
```

Вот пример использования:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

Параметр `t` и переменная `letters` являются синонимами к одному и тому же объекту. Стековая диаграмма выглядит следующим образом:



Важно различать операции, которые изменяют списки от операций, которые создают новые списки. Например, метод `append` изменяет список, тогда как оператор + создает новый:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

Разница становится важной, когда вы пишете функцию, предназначенную изменять список. Например, следующая функция *не удаляет* первый элемент списка:

```
def bad_delete_head(t):
    t = t[1:] # Неправильно!
```

Оператор среза создает новый список, а операция присваивания заставляет переменную `t` ссылаться на него, но ни то, ни другое никак не затрагивает список, переданный в функцию в качестве аргумента.

Альтернативным способом будет написание функции, которая создает и возвращает новый список. Например, `tail` возвращает весь список кроме первого элемента:

```
def tail(t):
    return t[1:]
```

Эта функция оставляет первоначальный список нетронутым. Вот как она может использоваться:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Упражнение 10.2 Напишите функцию `chop`, которая принимает список и модифицирует его, удаляя первый и последний элементы, и возвращает `None`.

Затем напишите функцию `middle`, которая берет список и возвращает новый без первого и последнего элементов первоначального.

10.13 Отладка

Невнимательное обращение со списками (и другими изменяемыми объектами) ведет к долгим часам отладки. Вот несколько типичных ошибок и способов их предотвратить:

1. Не забывайте, что большинство методов списков изменяют аргумент и возвращают `None`. Это прямо противоположно работе методов строк, которые возвращают новую строку, а старую оставляют нетронутой.
   Если вы привыкли работать со строками так:

   ```
   word = word.strip()
   ```

   то у вас может возникнуть искушение работать сходным образом и со списками:

   ```
   t = t.sort() # Неправильно!
   ```

   Поскольку `sort` возвращает `None`, то, вероятнее всего, вы получите ошибку, когда в следующий раз обратитесь к списку `t`.
   Перед тем, как использовать в программе методы и операторы списков, внимательно прочитайте документацию и испытаете их в интерактивном режиме. Описание методов и операторов, которые применяются к спискам и другим последовательностям, можно найти на `docs.python.org/lib/typesseq.html`. Методы и операторы, которые применяются только к изменяемым последовательностям, задокументированы на `docs.python.org/lib/typesseq-mutable.html`.

2. Выберете определенный способ и придерживайтесь его.
   Часть проблем при использовании списков возникает от того, что существует множество разных способов сделать одно и то же. Например, чтобы удалить из списка элемент, можно воспользоваться операторами `pop`, `remove`, `del` или, даже, срезом.
   Для добавления элемента вы можете использовать либо метод `append`, либо оператор `+`. Имея список `t` и элемент `x`, следующие конструкции будут корректными:

   ```
   t.append(x)
   ```

   ```
   t = t + [x]
   ```

   А эти будут неправильными:

   ```
   t.appent([x])      # Неправильно!
   ```

   ```
   t = t.append(x)    # Неправильно!
   ```

   ```
   t + [x]            # Неправильно!
   ```

   ```
   t = t + x          # Неправильно!
   ```

   Испытайте каждый из этих примеров в интерактивном режиме и убедитесь, что вы понимаете, что они делают. Обратите внимание, что только последний пример вызывает ошибку при выполнение. Остальные же являются допустимыми, однако, они не делают то, что нам нужно.

3. Делайте копии и избегайте синонимов.
   Если, для примера, вы хотите использовать метод `sort`, который изменяет аргумент, но вам, также, необходимо сохранить и оригинальный список, вы можете сделать его копию:

   ```
   orig = t[:]
   t.sort()
   ```

   В данном примере вы могли бы также использовать и встроенную функцию `sorted`, которая возвращает новый, отсортированный список и не затрагивает оригинальный. Но также вам не следует и использовать имя `sorted` в качестве переменной!

10.14 Словарь терминов

список (list): последовательность значений.

элемент (element): одно из значений списка (или другой последовательности).

индекс (index): целое значение, указывающее на элемент списка.

вложенный список (nested list): список, являющийся элементом другого списка.

перебор списка (list traversal): последовательный доступ к каждому элементу списка.

отображение (как существительное – mapping): отношение, при котором каждый элемент одного набора соответствует элементам другого набора. Например, список является отображением индексов к элементам.

аккумулятор (accumulator): переменная, используемая в цикле для суммирования или аккумулирования результата.

операция приращения (augmented assignment): выражение, которое обновляет значение переменной при помощи оператора типа +=.

сокращение (reduce): прием в программировании, который перебирает последовательность и аккумулирует элементы в единое значение.

отображение (как действие – map): прием в программировании, при котором перебирается последовательность и над ее элементами совершаются определенные действия.

фильтрация (filter): процесс, при котором из списка выбираются лишь элементы, удовлетворяющие некоторому критерию.

объект (object): нечто, на что может ссылаться переменная; объект имеет тип и значение.

эквивалентный (equivalent): имеющий то же самое значение.

идентичный (identical): являющийся тем же самым объектом (что подразумевает также и эквивалентность).

ссылка (reference): связь между переменной и ее значением.

синоним (aliase): если на один объект ссылается несколько переменных, то синонимом называется одна из таких переменных.

разделитель (delimiter): символ или строка, используемые для того, чтобы указать на место, где строка должна разделяться (на слова).

10.15 Упражнения

Упражнение 10.3 Напишите функцию `is_sorted`, которая берет список в качестве параметра и возвращает `True`, если список отсортирован в восходящем порядке, и `False` в противном случае. Вы можете принять (в качестве входного условия), что элементы списка можно сравнить с помощью операторов сравнения <, > и т.п.

Например, `is_sorted([1,2,2])` должно возвратить `True`, а `is_sorted('b', 'a')` должно возвратить `False`.

Упражнение 10.4 Два слова являются анаграммами, если вы можете переставить буквы одного, чтобы составить другое. Напишите функцию `is_anagram`, которая берет две строки и возвращает `True`, если они являются анаграммами.

Упражнение 10.5 Так называемый, парадокс дня рождения:

1. Напишите функцию `has_duplicates`, которая берет список и возвращает `True`, если любой его элемент встречается более одного раза. Она не должна изменять оригинальный список.
2. Если в вашем классе 23 студента, каковы шансы того, что у двоих из вас день рождения выпадает на тот же самый день? Вы можете подсчитать вероятность этого, генерирую случайные числа для 23 дней рождения и проверяя их на совпадения. Подсказка: сгенерировать случайные дни рождения можно при помощи функции `randint`, входящей в модуль `random`.

Вы можете прочитать об этой задаче на `wikipedia.org/wiki/Birthday_paradox`, а также посмотреть мое решение на `thinkpython.com/code/birthday.py`.

Упражнение 10.6 Напишите функцию `remove_duplicates`, которая берет список и возвращает новый список, в котором элементы не повторяются. Подсказка: они не обязательно должны сохранять свой порядок.

Упражнение 10.7 Напишите функцию, которая читает `words.txt` и создает список одним элементом на каждое слово. Напишите две версии этой функции: в одной должен использоваться метод `append`, в другой конструкция `t = t + [x]`. Определите, какая функция выполняется дольше. Почему?

Можете посмотреть на мое решение `thinkpython.com/code/wordlist.py`.

Упражнение 10.8 Чтобы проверить, содержится ли слово в списке, можно воспользоваться оператором `in`, но это будет медленно, т.к. поиск происходит по порядку во всех словах.

Так как слова в списке отсортированы по алфавиту, данный процесс можно ускорить, используя бисекционный поиск (метод деления пополам), подобный тому, когда вы ищите слово в словаре. Вы начинаете с середины и смотрите, находится ли искомое слово в первой половине списка. Если да, то вы ищите в первой половине подобным же образом. Если нет, то вы ищите во второй половине.

Каждый раз вы делите оставшуюся часть пополам. Если список слов состоит из 113809 слов, вам потребуется, максимум, 17 шагов чтобы либо найти слово, либо определить, что его нет в списке.

Напишите функцию `bisect`, которая принимает отсортированный список и искомое значение, и возвращает индекс этого значения в списке либо `None`, если его там нет.

Или вы можете прочитать документацию по модулю `bisect` и пользоваться им!

Упражнение 10.9 Два слова называются "обратной парой", если каждое из них является обратной версией другого (например, "pot" и "top"). Напишите программу, которая находит все обратные пары в списке слов.

Упражнение 10.10 Два слова образуют так называемый "интерлок" (interlock), если из всех их букв можно составить третье слово[21]. Например, из слов "shoe" и "cold" можно составить "schooled".

1. Напишите программу, которая будет находить все пары слов, составляющих интерлок. Подсказка: не перечисляйте (enumerate) все пары!
2. Можете ли вы найти слова, которые образуют тройной интерлок; т.е. новое слово получается, если из исходного брать каждую третью букву, начиная с первого, второго или третьего символа?

[21] Данное упражнение навеяно примером из `puzzlers.org`

Глава 11
Словари

Словарь (dictionary) похож на список, но он более универсальный. Если в списке индексы должны быть целыми числами, то в словаре они могут быть (почти) любого типа.

Вы можете представлять себе словарь в виде отображения набора индексов, которые называются **ключами** (key), на набор значений. Каждый ключ отображается на значение (другими словами, каждому ключу соответствует определенное значение). Связь между ключом и значением называется **пара ключ-значение** (key-value pair) и, иногда, **элемент** (item).

В качестве примера мы создадим простой словарь, который устанавливает (отображает) отношения между английскими и испанскими словами, поэтому как ключи, так и значения будут являться строками.

Функция `dict` создает новый пустой словарь. Поскольку `dict` является именем встроенной функции, вам не следует создавать переменную с таким именем.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

Фигурные скобки, `{}`, представляют пустой словарь. Для добавления новых элементов (т.е. пар ключ-значение) в словарь вы можете использовать квадратные скобки:

```
>>> eng2sp['one'] = 'uno'
```

Эта строка создает элемент, в котором ключ `'one'` отображается на значение `'uno'`. Если мы снова распечатаем этот словарь, то увидим пару ключ-значение с двоеточием посреди них.

```
>>> print(eng2sp)
{'one': 'uno'}
```

Этот формат вывода одновременно является и форматом ввода. Например, вы можете создать новый словарь с тремя элементами:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Но если вы распечатаете `eng2sp`, вас может ожидать сюрприз:

```
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Порядок пар ключ-значение может отличаться от того, что вы вводили. На самом деле, если вы пробуете этот пример на вашем компьютере, то можете получить и другой результат. В общем случае, порядок элементов в словаре непредсказуем.

Но это не является проблемой, потому что элементы в словаре никогда не индексируются целыми числами. Вместо этого вы используете ключ, чтобы посмотреть на соответствующее ему значение:

```
>>> print(eng2sp['two'])
'dos'
```

Ключ `'two'` всегда отображается на значение `'dos'`, поэтому порядок элементов в словаре не имеет никакого значения.

Если такого ключа нет в словаре, вы получите сообщение об ошибке:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

Функция `len` работает и со словарями. Она возвращает количество пар ключ-значение:

```
>>> len(eng2sp)
3
```

Оператор `in` также работает со словарями. Он показывает, содержится ли в словаре такой *ключ*:

```
>>> 'one' in eng2sp
True
```

```
>>> 'uno' in eng2sp
False
```

Чтобы проверить, содержится ли в словаре определенной значение, вы можете использовать метод `values`, который возвращает все значения словаря в качестве списка, а уже потом применить оператор `in` к получившемуся списку:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Оператор `in` использует разный алгоритм работы в словарях и списках. Для списков он использует алгоритм поиска, подобный тому, что мы видели в разделе 8.6. По мере того, как список увеличивается в размере, пропорционально увеличивается и время поиска. В случае со словарями Python использует алгоритм **хеширования** (hashtable), который имеет одно замечательное свойство: скорость работы оператора `in` не зависит от количества элементов в словаре. Я не буду объяснять, как такое возможно, но вы можете прочитать об этом на википедии `wikipedia.org/wiki/Hash_table`.

Упражнение 11.1 Напишите функцию, которая читает список слов в `words.txt` и сохраняет их в качестве ключей словаря. Нет разницы, что вы будете использовать в качестве значений (для определенности это может быть цифра 0). После этого вы можете использовать оператор `in` для быстрой проверки того, содержится ли в словаре определенное слово.

Если вы делали упражнение 10.8, можете сравнить скорость работы вашей новой программы с той, что использовала списки, а также с той, что использовала бисекционный поиск.

11.1 Словари в качестве счетчиков

Представьте, что вам дана строка, и вы хотите подсчитать, сколько раз в ней встречается каждая буква. Есть несколько способов сделать это:

1. Вы можете создать 26 переменных для каждой буквы английского алфавита. Затем вы можете перебрать всю строку, и для каждой буквы увеличивать соответствующий ей счетчик, возможно, используя связанные условия
2. Вы можете создать список с 26 элементами. Затем вы можете конвертировать каждую букву в соответствующий ей номер (используя для этого встроенную функцию `ord`), использовать этот номер в качестве индекса в списке для увеличения соответствующего счетчика.
3. Вы можете создать словарь с символами в качестве ключей и счетчиками в качестве соответствующий значений. Если вы встретите определенный символ в первый раз, вы добавите его в словарь. После этого вы будете увеличивать значение соответствующего элемента.

Каждый из этих вариантов совершает одинаковые вычисления, но реализация этих вычислений будет разная.

Реализация (implementation) это способ совершения вычислений; некоторые реализации лучше других. Для примера, преимуществом реализации данной программы с использованием словаря является то, что нам не нужно знать заранее, какие буквы нам встретятся. Нам нужно лишь предоставить место для тех букв, которые на самом деле встретятся.

Вот как может выглядеть подобный код:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Такая функция называется **гистограммой** (histogram), что является статистическим термином для подсчета определенных значений (частоты их повторения).

Первая строчка функции создает пустой словарь. Цикл `for` перебирает всю строку. Каждый раз при прохождении цикла если символ `c` не присутствует в словаре, мы создаем новый элемент с ключом `c` и начальным значением `1` (поскольку мы уже видели этот символ один раз). Если же `c` уже находится в словаре, мы просто увеличиваем `d[c]` на единицу.

Вот как это работает:

```
>>> h = histogram('brontosaurus')
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Гистограмма показывает, что буквы '`a`' и '`b`' встречаются по одному разу, '`o`' два раза и т.д.

Упражнение 11.2 У словарей есть метод `get`, который принимает ключ и некоторое значение по умолчанию. Если ключ находится в словаре, `get` возвращает соответствующее значение; в противном случае, возвращается значение по умолчанию. Пример:

```
>>> h = histogram('a')
>>> print(h)
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Используйте метод `get` для написания более краткой версии `histogram`. Вы должны избавиться от оператора `if`.

11.2 Циклы и словари

Если вы используете словарь в цикле `for`, то он перебирает его ключи. Например, `print_hist` выводит каждый ключ и соответствующее ему значение:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Опять же, ключи не выводятся в определенном порядке.

Упражнение 11.3 У словарей имеется метод `keys`, который возвращает неупорядоченный список ключей.

Модифицируйте `print_hist` так, чтобы выводить ключи с соответствующими им значениями в алфавитном порядке (сортировка должна осуществляться по ключам, не по значениям).

11.3 Поиск ключей по их значениям

Если у вас есть словарь `d` и ключ `k`, то легко найти соответствующее значение `v = d[k]`. Такую операцию можно назвать **поиск по словарю** (lookup).

Но что делать, если у вас есть `v`, а вы хотите найти `k`? Здесь возникают две проблемы: в словаре может быть более одного ключа со значением `v`. В зависимости от разрабатываемого вами приложения, вам может потребоваться возможность выбрать один из них, или получить список со всеми ключами, имеющими данное значение. Другая проблема заключается в том, что не существует простого синтаксиса для осуществления **обратного поиска** (reverse lookup). Вы должны просмотреть все значения.

Вот функция, которая берет значение и возвращает первый ключ, который отображается на это значение:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise ValueError
```

Эта функция демонстрирует еще один способ поиска, но она использует инструкцию, с которой мы еще не встречались, `raise`. Эта команда вызывает ошибку; в данном случае это будет ошибка `ValueError`, которая указывает на ошибку со значением параметра.

Если мы выйдем из цикла, это будет означать, что `v` не содержится в словаре в качестве одного из значений, поэтому мы и вызываем ошибку.

Вот пример удачного обратного поиска:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print(k)
r
```

А вот неудачный поиск:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    File "<stdin>", line 5, in reverse_lookup
ValueError
```

Когда вы искусственно вызываете ошибку, это будет выглядеть так же, как это делает и сам Python: выводится отслеживание и сообщение об ошибке.

Команда `raise` может принимать в качестве дополнительного параметра детальное описание ошибки, например:

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

Обратный поиск происходит намного медленнее, чем прямой поиск по ключам словаря; если вам необходимо осуществлять его часто, или если ваш словарь увеличивается в размерах, то производительность вашей программы будет падать.

Упражнение 11.4 Модифицируйте `revers_lookup` так, чтобы она возвращала список *всех* ключей, которые отображаются на `v`, или пустой список, если такого значения нет в словаре.

11.4 Словари и списки

Списки могут быть значениями в словаре. Например, если у вас имеется словарь, который отображает буквы на частоту их использования, то вы можете инвертировать его, т.е. создать словарь, который отображает частоту использования на конкретные буквы. Поскольку может быть несколько букв с одинаковой частотой, каждое значение в инвертированном словаре должно представлять из себя список букв.

Вот функция, которая инвертирует словарь:

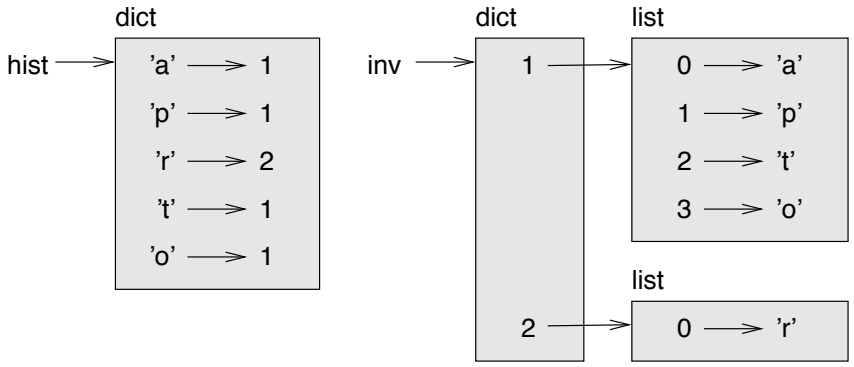
```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

Каждый раз при прохождении цикла `key` получает ключ из `d`, а переменной `val` присваивается соответствующее значение. Если `val` не содержится в `inv`, это означает, что мы еще с ним не встречались, поэтому мы создаем новый элемент и инициализируем его **одноэлементным множеством** (singleton), т.е. списком, содержащим единственный элемент. В противном случае это означает, что мы уже встречались с этим значением, поэтому мы добавляем соответствующий ключ к списку.

Вот пример:

```
>>> hist = histogram('parrot')
>>> print(hist)
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print(inv)
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

А вот как выглядит диаграмма для `hist` и `inv`:



Словари представлены прямоугольниками с типом `dict` сверху и парами ключ-значение внутри. Если значения являются целыми или дробными числами, а также строками, то я обычно рисую их внутри прямоугольника. Если же значения являются списками, то я изображаю их снаружи, чтобы не усложнять восприятие диаграммы.

Хотя списки могут быть значениями в словаре, как было показано выше, они не могут быть его ключами. Вот что произойдет, если вы, все же, попытаетесь это сделать:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Как я упомянул ранее, словари созданы таким образом, что используют таблицу хешей, а это означает, что их ключи должны быть **хешируемыми** (hashable).

Хеш (hash) это функция, которая берет любое значение и возвращает целое число. Словари используют эти значения, называемые хешами, чтобы хранить и искать пары ключ-значение.

Эта система прекрасно работает лишь в том случае, если ключи принадлежат к неизменяемому типу данных. Если же ключи вдруг будут изменяемыми, как, например, списки, то много неприятных вещей может произойти. Например, когда вы создаете пару ключ-значение, Python хеширует ключ и хранит его в соответствующем месте. Если вы модифицируете ключ и захешируете его снова, то вы окажетесь уже в другом месте. В таком случае у вас может оказаться два разных значения для одного ключа, или вы можете оказаться неспособными отыскать ключ. В любом случае, словарь не будет работать правильно.

Именно поэтому ключи должны быть хешируемыми, и именно поэтому изменяемые типы, подобные спискам, не могут быть ключами словаря. Самым простым способом обойти данное ограничение является использование кортежей, с которыми мы познакомимся в следующей главе.

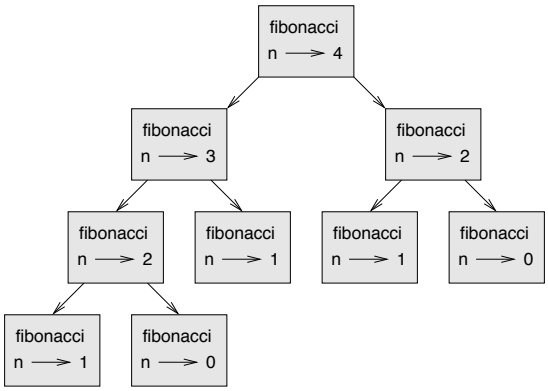
А поскольку сами словари относятся к изменяемому типу, то они не могут использоваться в качестве ключей других словарей, но *могут* использоваться в качестве их значений.

Упражнение 11.5 Прочитайте документацию к методу словарей, называемому `setdefault`, и используйте его для написания более краткой версии `invert_dict`.

11.5 Memo

Если вы играли с функцией `fibonacci` из раздела 6.7, то, вероятно, обратили внимание, что чем больший аргумент вы задаете, тем большее время требуется функции для вычисления значения. Более того, время выполнения возрастает очень быстро.

Чтобы понять, почему так происходит, рассмотрите **граф вызова** (call graph) функции `fibonacci` с `n=4`:



Граф вызова показывает функции, представленные фреймами, с линиями, соединяющими каждый фрейм с тем, который его вызывает. На самом верху этого графа `fibonacci` с `n=4` вызывает `fibonacci` с `n=3` и `n=2`. В свою очередь, `fibonacci` с `n=3` вызывает `fibonacci` с `n=2` и `n=1`. И так далее.

Подсчитайте, сколько раз вызываются `fibonacci(0)` и `fibonacci(1)`. Такое решение проблемы является неэффективным, и оно становится тем хуже, чем больше становится аргумент.

Для решения этой проблемы можно вести учет тех значений, которые уже были подсчитаны, и хранить их в словаре.

Значения, вычисленные ранее и хранящиеся для последующего использования, называются **memo**[22]**.**

Вот пример реализации функции `fibonacci`, использующей memo:

```
known = {0:0, 1:1}
def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

`known` это словарь, который ведет учет всех чисел Фибоначчи, которые мы уже знаем. Его начальными значениями являются две пары: `0:0` и `1:1`.

Когда происходит вызов `fibonacci`, она проверяет `known`. Если результат уже содержится там, то она немедленно его возвращает. В противном случае она должна вычислить новое значение, записать его в словарь, а затем вернуть.

Упражнение 11.6 Запустите эту версию `fibonacci` и сравните скорость ее работы с оригинальной (той, что не использует memo) при различных параметрах.

11.6 Глобальные переменные

В предыдущем примере переменная `known` была создана за пределами функции, поэтому она принадлежит особой структуре, называемой `__main__`. Переменные в `__main__` иногда называют **глобальными** (global), потому что они доступны из любой функции. В противоположность локальным

[22] См. `wikipedia.org/wiki/Memoization`

переменным, которые исчезают, как только заканчивает работу функция, где они содержатся, глобальные переменные существуют в течение всего времени работы главной программы.

Довольно часто глобальные переменные используются в качестве **флагов** (flag), т.е. булевых переменных, которые показывают, истинно ли некое состояние. Например, некоторые программы используют флаги, называемые `verbose`, для контроля количества выводимой при работе информации:

```
verbose = True
def example1():
    if verbose:
        print('Выполняется функция example1')
```

Если вы попытаетесь присвоить глобальной переменной новое значение, то вас может ожидать сюрприз. В следующем примере мы пытаемся запомнить, была ли уже когда-либо вызвана функция `example2`:

```
been_called = False     # True если функция уже вызывалась
def example2():
    been_called = True  # НЕПРАВИЛЬНО
```

Но если вы запустите данную функцию, то обнаружите, что значение `been_called` не изменилось. Проблема заключается в том, что функция `example2` создает новую локальную переменную с именем `been_called`. Эта локальная переменная исчезает сразу же после того, как функция завершает свою работу, и не оказывает совершенно никакого влияния на глобальную переменную с тем же именем.

Чтобы иметь возможность присваивать глобальной переменной новые значения внутри функции, вы должны **объявить** (declare) глобальную переменную перед ее использованием:

```
been_called = False
def example2():
    global been_called
    been_called = True
```

Команда `global` как бы говорит интерпретатору следующее: "В этой функции когда я говорю `been_called`, я имею в виду глобальную переменную; не создавай новую локальную с таким же именем".

Вот пример, который пытается обновить глобальную переменную:

```
count = 0
def example3():
    count = count + 1   # НЕПРАВИЛЬНО
```

Если вы его запустите, то получите сообщение об ошибке:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python предполагает, что `count` является локальной переменной, что означает, что вы читаете ее прежде, чем записать в нее. Опять же, решением проблемы будет объявление `count` глобальной переменной:

```
def example3():
    global count
    count += 1
```

Если глобальное значение относится к изменяемым типам, то вы можете изменять его без объявления:

```
known = {0:0, 1:1}
def example4():
    known[2] = 1
```

Поэтому вы можете добавлять, удалять и замещать элементы глобального списка или словаря без их объявления внутри функции. Но если вам требуется присвоить глобальной переменной новое значение, ее необходимо объявить:

```
def example5():
    global known
        known = dict()
```

11.7 Отладка

Когда вы работаете с большими объемами данных, то ручной вывод и проверка данных для отладки становится неудобным. Вот несколько советов для отладки больших объемов данных:

Уменьшите входные данные: если возможно, сократите объем данных. Например, если программа читает большой текстовый файл, то начните с чтения первых 10 строк. Вы можете как редактировать сам файл, так и (что лучше) модифицировать программу таким образом, чтобы она читала только первые *n* строк.

Если в программе будет ошибка, то вы можете уменьшить *n* до еще меньшего значения, чтобы легче было отыскать ошибку. Затем вы можете постепенно увеличивать объем данных.

Проверяйте общее количество данных и их типы: вместо того, чтобы выводить и проверять все данные, рассмотрите возможность вывода только их общего количества: например, общее число элементов в словаре или в списке.

Частой ошибку во время выполнения вызывает переменная, которая имеет неправильный тип. Для отладки подобных ошибок часто бывает достаточно просто вывести тип переменной.

Создавайте самопроверочные тесты: иногда вы можете написать код, который будет автоматически проверять наличие ошибки. Например, если вы подсчитываете среднее арифметическое списка чисел, вы можете проверить, чтобы результат не был больше максимального числа в списке, и не меньше минимального.

Другой вид проверки сравнивает результаты двух различных вычислений на предмет того, являются ли они последовательными. Это называется "проверкой на последовательность".

Форматируйте вывод отладочной информации: Это облегчает обнаружение ошибок. Мы видели это на примере из раздела 6.9. Модуль `pprint` предоставляет функцию `pprint`, которая выводит встроенные типы в более читаемом виде.

Помните, время, которое вы тратите на написание отладочного кода, экономит время, которое вы тратите на отладку.

11.8 Словарь терминов

словарь (dictionary): отображение множества ключей на соответствующие им значения.

пара ключ-значение (key-value pair): изображение отношения между ключом и значением.

элемент (item): другое название пары ключ-значение.

ключ (key): объект, который находится в словаре в качестве первой части пары ключ-значение.

значение (value): объект, который находится в словаре в качестве второй части пары ключ-значение. В данном случае этот более специфичный термин, нежели слово "значение", которое мы использовали ранее.

реализация (implementation): способ выполнения вычислений.

таблица хешей (hashtable): алгоритм, используемый Python для реализации словарей.

хешируемый (hashable): тип, который имеет функция, выполняющая хеширование. Неизменяемые типы, такие как целые числа, дроби и строки являются хешируемыми; изменяемые типы, такие как списки и словари, ими не являются.

поиск по словарю (lookup): операция со словарем, которая берет ключ и находит соответствующее ему значение.

обратный поиск (reverse lookup): операция со словарем, которая берет определенное значение и находит один или более ключей, которые отображаются на это значение.

одноэлементное множество (singleton): список (или любая другая последовательность) с единственным элементом.

граф вызова (call graph): диаграмма, которая показывает каждый фрейм, созданный во время выполнения программы, который соединяется с другими фреймами стрелками; стрелки обозначают, какой фрейм кем был вызван.

гистограмма (histogram): набор счетчиков.

memo: уже вычисленное значение, которое хранится на тот случай, чтобы предотвратить повторное вычисление.

глобальная переменная (global value): переменная, определенная вне функции; доступ к глобальным переменным может осуществляться из любой функции.

флаг (flag): булева переменная, используемая для указания на то, является ли некоторое условие истинным.

объявление (declaration): команда наподобие `global`, которая говорит интерпретатору что-нибудь о переменной.

11.9 Упражнения

Упражнение 11.7 Если вы выполняли упражнение 10.5, то у вас уже есть функция `has_duplicates`, которая берет в качестве параметра список и возвращает `True`, если любой объект в нем встречается более одного раза.

Используйте словарь, чтобы написать более быструю и простую версию `has_duplicates`.

Упражнение 11.8 Два слова образуют "сдвинутую пару" (rotate pair), если вы можете "сдвинуть" одно из них, чтобы получить другое (см. `rotate_word` в упражнении 8.12).

Напишите программу, которая читает файл `words.txt` и находит все слова, образующие между собой сдвинутые пары.

Упражнение 11.9 Вот еще одна загадка из передачи *Car Talk*[23]:

Эта загадка была прислана слушателем по имени Dan O'Leary. Недавно ему встретилось слово, состоящее из пяти букв и всего одного слога, которое обладало уникальными свойствами: если вы удалите первую букву, то оставшиеся образуют с оригинальным слово омофон (слова одинакового произношения, которые могут иметь разное значение и написание). Если вы вернете первую букву на место, но удалите вторую букву, то в результате опять получите омофон с оригинальным словом. Что это за слово?

Сейчас я приведу вам пример, который не совсем точный. Давайте посмотрим на пятибуквенное слово `'wrack'`: `WRACK`. Если вы удалите первую букву, у вас останется четырехбуквенное слово '`RACK`'. Слова `wrack` и `rack` произносятся одинаково. Если вы вернете на место `w` и удалите `r`, то у вас получится слово `'wack'`. Такое слово есть, но оно произносится по-другому, не так, как `wrack` и `rack`.

Но есть, как минимум, одно слово, о котором Дэну и нам известно, что оно образует два омофона, если удалить либо первую, либо вторую букву. Итак, что же это за слово?

Для проверки того, находится ли слово в `words.txt`, вы можете использовать программу из упражнения 11.1.

Для проверки того, являются ли два слова омофонами, вы можете воспользоваться словарем произношений проекта CMU. Вы можете скачать его с сайта `www.speech.cs.cmu.edu/cgibin/cmudict` или `thinkpython.com/code/c06d`. Также вы можете скачать `thinkpython.com/code/pronounce.py`, что предоставляет функцию `read_dictionary`, которая читает словарь произношений и возвращает словарь, используемый в Python, который отображает каждое слово на его произношение.

Напишите программу, которая выводит все слова, подходящие для решения этой загадки. Вы можете посмотреть на мое решение `thinkpython.com/code/homophone.py`.

---

[23] `www.cartalk.com/content/puzzler/transcripts/200717`

Глава 12
Кортежи

12.1 Кортежи принадлежат к неизменяемому типу

Кортеж представляет из себя последовательность значений. Эти значения могут быть любого типа, они индексируются целыми числами, поэтому в этом отношении кортежи сильно похожи на списки. Важным отличием является то, что кортежи не изменяются.

С точки зрения синтаксиса, кортеж представляет из себя список значений, разделенных запятыми:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Хотя это и не обязательно, но, тем не менее, принято заключать кортежи в скобки:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Чтобы создать кортеж, состоящий из одного элемента, вы должны добавить в конце запятую:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

Одиночное значение в скобках (без заключительной запятой) не является кортежем:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Другим способом создания кортежа является использование встроенной функции `tuple`. Если ей не дать аргументов, то она создает пустой кортеж:

```
>>> t = tuple()
>>> print(t)
()
```

Если в качестве аргумента предоставить какую-нибудь последовательность (строку, список или кортеж), то результатом будет являться кортеж с элементами этой последовательности:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Т.к. `tuple` служит именем встроенной функции, вы не должны создавать переменную с таким же именем.

Большинство операторов, работающих со строками, работают также и с кортежами:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Оператор среза также выбирает диапазон элементов:

```
>>> print(t[1:3])
('b', 'c')
```

Но если вы попытаетесь изменить один из элементов кортежа, вы получите сообщение об ошибке:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Вы не можете изменять элементы кортежа, но вы можете заменять один кортеж другим:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

12.2 Кортежи и операция присваивания

Часто бывает необходимо поменять местами значения двух переменных. При использовании обычной операции присваивания вам необходима временная переменная. Например, чтобы поменять местами *a* и *b*, можно использовать следующее:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Но эта конструкция занимает несколько строк. **Операция присваивания кортежа** (tuple assignmetn) более лаконична:

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

С левой стороны находится кортеж переменных; с правой стороны находится кортеж выражений. Каждое значение присваивается соответствующей ему переменной. Все выражения, находящиеся с правой стороны, вычисляются до того, как происходит присваивание.

Количество переменных с левой стороны должно быть равным количеству значений с правой стороны:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Если говорить более обобщенно, то справа могут находиться любые последовательности (строка, список, кортеж). Например, чтобы разделить e-mail на имя пользователя и домен, вы можете использовать следующий код:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Возвращаемым значением метода `split` будет список из двух элементов; первый элемент присваивается переменной `uname`, второй – `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

12.3 Кортежи в качестве возвращаемого значения

Строго говоря, функция может возвращать лишь одно значение, но если это значение будет кортежем, то результат будет такой же, как и если бы возвращалось несколько значений. Например, если вы хотите разделить одно целое число на другое, вычислив их частное и остаток, то не слишком эффективно будет вычислять сначала `x/y`, а потом `x%y`. Гораздо лучше вычислить оба значения одновременно.

Встроенная функция `divmod` принимает два аргумента и возвращает кортеж из двух значений, одно из которых будет частным, а другое остатком. Вы можете хранить результат в виде кортежа:

```
>>> t = divmod(7, 3)
>>> print(t)
(2, 1)
```

Или вы можете использовать операцию присваивания кортежей, чтобы хранить значения по отдельности:

```
>>> quot, rem = divmod(7, 3)
>>> print(quot)
2
>>> print(rem)
1
```

Вот пример функции, которая возвращает кортеж:

```
def min_max(t):
    return min(t), max(t)
```

Встроенные функции `man` и `mix` находят максимальный и минимальный элементы последовательности. `min_max` вычисляет оба таких значения и возвращает их в качестве кортежа из двух элементов.

12.4 Кортежи с переменным числом аргументов

Функции могут принимать переменное количество аргументов. Параметр, начинающийся со звездочки, `*`, **собирает** (gather) аргументы в кортеж. Например, `printall` принимает любое количество аргументов и выводит их на печать:

```
def printall(*args):
    print(args)
```

Собирательный параметр может иметь любое имя, но принято называть его `args`. Вот как работает функция:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Кроме собирания оператор `*` осуществляет также и **разбивку** (scatter). Если у вас есть последовательность значений, и вы хотите передать их вашей функции как несколько аргументов, то вы также можете использовать оператор `*`. Например, функция `divmod` принимает ровно два аргумента, она не работает с кортежами:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Но если вы "разобьете" кортеж на отдельные элементы, то это сработает:

```
>>> divmod(*t)
(2, 1)
```

Упражнение 12.1 Многие из встроенных функций имеют аргументы переменной длины. Например, `min` и `max` могут принимать различное число аргументов:

```
>>> max(1,2,3)
3
```

Но с функцией `sum` это не проходит:

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

Напишите функцию `sumall`, которая принимает любое количество аргументов и возвращает их общую сумму.

12.5 Списки и кортежи

Встроенная функция `zip` берет две или более последовательностей и объединяет их в список[24] кортежей таким образом, что каждый кортеж состоит из одного элемента с каждой последовательности.

Вот пример работы `zip` для строки и списка:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> for e in zip(s, t):
        print(e)
('a', 0)
('b', 1)
('c', 2)
```

[24] Говоря технически, `zip` возвращает итератор кортежей, но для большинства случаев он ведет себя также как если бы это был просто список.

Результатом этого кода является, по существу, итератор списка кортежей, где каждый кортеж содержит символ из строки и соответствующий ему элемент из списка.

Если последовательности будут разной длины, то результат будет иметь длину меньшей последовательности:

```
>>> for e in zip('Anne', 'Elk'):
        print(e)
('A', 'E')
('n', 'l')
('n', 'k')
```

Вы можете использовать присваивание кортежам в цикле `for` для перебора списка кортежей:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Каждый раз при прохождении цикла Python выбирает следующий кортеж из списка и присваивает его элементы переменным `letter` и `number`. Результат выглядит следующим образом:

```
0 a
1 b
2 c
```

Если вы объедините `zip`, `for` и присваивание кортежу, у вас получится полезный код для перебора двух и более последовательностей одновременно. Например, `has_match` берет две последовательности, `t1` и `t2`, и возвращает `True`, если имеется хотя бы один индекс `i`, такой, что `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Если вам необходимо перебрать элементы последовательности и ее индексы, можете воспользоваться встроенной функцией `enumerate`:

```
for index, element in enumerate('abc'):
    print(index, element)
```

Вывод будет выглядеть как и в предыдущем случае:

```
0 a
1 b
2 c
```

12.6 Словари и кортежи

У словарей имеется довольно эффективный метод `items`, который возвращает список кортежей, в котором каждый кортеж является парой ключ-значение:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> type(t)
<class 'dict_items'>
>>> print(t)
dict_items([('a', 0), ('c', 2), ('b', 1)])
>>> for e in t:
        print(e)
('a', 0)
('c', 2)
('b', 1)
```

Как вы и должны были ожидать от словаря, его элементы не находятся в каком-либо порядке.

И обратно, вы можете воспользоваться списком кортежей для инициализации нового словаря:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print(d)
{'a': 0, 'c': 2, 'b': 1}
```

Объединение `dict` и `zip` служит лаконичным способом для создания словаря:

```
>>> d = dict(zip('abc', range(3)))
>>> print(d)
{'a': 0, 'c': 2, 'b': 1}
```

Метод словарей `update` также берет список кортежей и добавляет их как пары ключ-значение к существующему словарю.

Объединение `items`, присваивание кортежу и `for` предоставляет вам способ перебора ключей и значений словаря:

```
for key, val in d.items():
    print(val, key)
```

Мы снова получаем уже знакомый результат:

```
0 a
1 b
2 c
```

Довольно часто кортежи используются в качестве ключей в словарях (главным образом потому, что вы не можете использовать там списки). Например, телефонный справочник может отображать связи между парами имя-фамилия и телефонными номерами. Предполагая, что у нас есть определенные значения `first`, `last` и `number`, мы можем написать следующее:

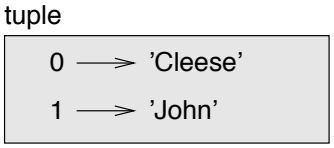
```
directory[last,first] = number
```

Выражение в квадратных скобках представляет из себя кортеж. Для перебора элементов словаря можно использовать операцию присваивание кортежу.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Этот цикл перебирает ключи словаря, которые являются кортежами. Каждый элемент кортежа присваивается переменным `last` и `first` соответственно. Затем функция `print` выводит имя, фамилию и соответствующий им телефонный номер.

Есть два способа изображения кортежей на диаграмме состояния. Более подробный способ показывает индексы и элементы точно так же, как они выглядят и в списке. Например, кортеж `('Cleese', 'John')` будет выглядеть следующим образом:



Но если диаграмма больше, то вы, возможно, захотите опустить некоторые детали. Например, диаграмма телефонного справочника может выглядеть следующим образом:

Здесь кортежи изображены, используя синтаксис Python. Кстати, эти телефонные номера взяты со службы BBC, поэтому, пожалуйста, не звоните по ним!

12.7 Сравнение кортежей

Операторы сравнения можно использовать как при работе с кортежами, так и при работе с другими последовательностями. Python начинает со сравнения первого элемента каждой последовательности. Если они одинаковы, сравниваются следующие элементы и т.д. до тех пор, пока не будет обнаружена разница. При сравнении элементы, следующие после рассматриваемых в данный момент, не принимаются во внимание, даже если они сравнительно большие:

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Функция `sort` работает подобным же образом.

Данное свойство используется в программировании под названием **DSU идиомы**[25] (decorate-sort-undecorate) или, по-другому, *преобразование Шварца* (Schwartzian tramsform) :

Decorate – построение списка кортежей с одним или более ключами для сортировки, которые предшествуют элементам последовательности,

Sort – сортировка построенного списка, а затем

Undecorate – извлечение отсортированных элементов последовательности.

Для примера, предположим, что у вас имеется список слов, который вам необходимо отсортировать по длине слов, начиная с самых длинных:

```
def sort_by_length(words):
    t = []
    for word in words:
        t.append((len(word), word))
    t.sort(reverse=True)
    res = []
    for length, word in t:
        res.append(word)
    return res
```

Первый цикл создает список кортежей, где каждый кортеж состоит из пары (длина слова, слово).

Далее, `sort` сравнивает первый элемент – длину, а второй элемент рассматривается лишь если необходимо разорвать связи. Если при этом в качестве дополнительного аргумента используется ключевое слово `reverse=True`, то сортировка осуществляется по возрастанию длины.

Второй цикл перебирает список кортежей и создает список слов в порядке убывания их длины.

Упражнение 12.2 В примере, данном выше, при сравнении слов разрушаются связи, так что слова одинаковой длины выводятся в обратном алфавитном порядке. Но для некоторых приложений вы, вероятно, захотите, чтобы связи были нарушены в случайном порядке. Измените этот код так, чтобы слова одинаковой длины выводились в случайном порядке. Подсказка: вам поможет функция `random` из модуля `random`.

12.8 Последовательности последовательностей

Я заострил ваше внимание на списках, состоящих из кортежей, но почти все примеры данной главы также применимы и к спискам из списков, кортежам из кортежей и кортежам из списков. Чтобы не

[25] В настоящее время DSU-сортировка считается устаревшим методом. См., например, http://wiki.python.org/moin/HowTo/Sorting/ - *прим. пер.*

перечислять все возможные комбинации, иногда проще говорить о последовательностях, состоящих из других последовательностей.

Во многих случаях разные типы последовательностей (строки, списки и кортежи) можно взаимозаменять. Итак, в каких случаях и почему вы должны предпочитать одни последовательности другим?

Начнем с очевидного: строки более ограничены, чем другие последовательности, потому что их элементами могут быть только символы. Также они неизменяемы. Если вам необходима возможность изменять символы в строке, а не просто создавать новые строки, то, вероятно, вам более подойдут списки, состоящие из символов.

Списки используются чаще, чем кортежи, главным образом потому, что они изменяемы. Но есть несколько случаев, когда предпочтительнее использовать кортежи:

1. В некоторых конкретных случаях, таких, как, например, инструкция `return`, проще создать кортежи, чем списки. В других случаях списки могут оказаться предпочтительнее.
2. Если вы планируете использовать последовательности в качестве ключей словаря, вам необходим неизменяемый тип, такой как строка или кортеж.
3. Если вы передаете последовательность функции в качестве аргумента, использование кортежей уменьшает потенциальный риск в виде непредсказуемого поведения из-за использования синонимов.

Т.к. кортежи принадлежат к неизменяемому типу, у них отсутствуют такие методы, как `sort` и `reverse`, которые модифицируют существующие списки. Но в Python имеются встроенные функции `sorted` и `reversed`, которые принимают любую последовательность в качестве параметра и возвращают новый список с теми же самыми элементами, но в другом порядке следования.

12.9 Отладка

Списки, словари и кортежи имеют одно общее название – **структуры данных** (data structures). В этой главе мы уже начали встречаться с составными структурами данных, как, например, списками кортежей, или словарями, содержащими кортежи в качестве ключей, а списки в качестве значений. Составные структуры данных полезны, но потенциально подвержены ошибкам, которые я называю **ошибки формы** (shape errors), т.е. ошибки, возникающие тогда, когда в структуре данных используется неправильный тип, размер или композиция. Например, если вы ожидаете список с одним элементом целого типа, а я дам вам просто целое число (не находящееся в списке), то это может вызвать ошибку.

Чтобы упростить отладку подобных ошибок, я написал модуль под названием `structshape`, предоставляющий функцию, также называемую structshape, которая принимает любую структуру данных и возвращающая строку, в которой содержится информация о форме данной структуры. Вы можете скачать модуль по адресу `thinkpython.com/code/structshape.py`.

Вот результат ее работы в случае простого списка:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print(structshape(t))
list of 3 int
```

Более изощренная программа могла бы вывести и "list of 3 int*s*", но проще не учитывать множественное число. А вот список, состоящий из списков:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print(structshape(t2))
list of 3 list of 2 int
```

Если элементы списка имеют разный тип, то `structshape` группирует их по порядку и типу:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> print(structshape(t3))
list of (3 int, float, 2 str, 2 list of int, int)
```

А вот пример словаря с тремя элементами, которые отображают целые числа на строки:

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> d = dict(lt)
>>> print(structshape(d))
dict of 3 int->str
```

Если вы запутались в ваших структурах данных, модуль `structshape` может вам помочь.

12.10 Словарь терминов

кортеж (tuple): один из типов данных, представляющий из себя последовательность из элементов.

операция присваивания кортежу (tuple assignment): операция присваивания с последовательностью с правой стороны и кортежем переменных с левой. Сначала вычисляется правая часть, а затем ее элементы присваиваются соответствующим переменным, стоящим в левой части.

собирание (gather): операция собирания аргумента переменной длины в один кортеж.

разбивание (scatter): операция, в результате которой последовательность трактуется как список аргументов.

DSU (decorate-sort-undecorate), **Schwartsian transform** (**преобразование Шварца**): идиома программирования, в которой сначала создается список кортежей, затем происходит сортировка, а потом некоторая часть выводится в качестве результата.

структура данных (data structure): набор некоторых данных, часто организованных в списки, словари, кортежи и т.п.

форма структуры данных (shape of a data structure): информация о типе, размере и композиции структуры данных.

12.11 Упражнения

Упражнение 12.3 Напишите функцию под название `most_frequent`, которая берет строку и выводит ее символы в убывающем порядке частоты их использования в строке. Добудьте и испытайте различные тексты на разных языках, чтобы определить, какие буквы чаще всего встречаются в разных языках. Сравните ваши результаты с таблицей из википедии `wikipedia.org/wiki/Letter_frequencies`.

Упражнение 12.4 Еще больше анаграмм!

1. Напишите программу, которая читает список слов из файла (см. раздел 9.1) и выводит все наборы слов, которые являются анаграммами. Вот пример того, как это может выглядеть:
   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```
   Подсказка: вы можете создать словарь, который будет отображать набор букв на список слов, которые написаны с использованием этих букв. Вопрос заключается в том, каким образом вы представите эти буквы, чтобы их можно было использовать в качестве ключей словаря?
2. Измените предыдущую программу так, чтобы она выводила сначала самый большой список из слов-анаграмм, затем следующий по величине и т.д.
3. В игре *Scrabble* (игра на составление слов, русский аналог – игра "Эрудит") вы получаете "бинго", если выставите все семь фишек с буквами с вашей подставки так, что вместе с буквой на игровой доске они образуют слово из 8 букв. Какие наборы из 8 букв представляют из себя наиболее вероятные бинго?
4. Два слова образуют "пару-метатезис" (metathesis pair), если вы можете превратить одно в другое перестановкой двух букв[26]. Например, "`converse`" и "`conserve`". Напишите программу, которая

[26] Это упражение навеяно примером с сайта `puzzlers.org`.

находит все пары-метатезисы в словаре. Подсказка: не проверяйте все пары слов и не проверяйте все возможные перестановки букв.
Вы можете скачать мое решение по адресу: `thinkpython.com/code/anagram_sets.py`.

Упражнение 12.5 Еще одна загадка из передачи *Car Talk*:[27]

Найдите самое длинное английское слово, которое остается правильным словом по мере того, как вы удаляете из него букву за буквой.

Вы можете удалять буквы с начала, с конца, с середины, но вы не можете перестанавливать оставшиеся буквы. Каждый раз, когда вы удаляете очередную букву, оставшиеся формируют другое английское слово. Если вы это сделаете, то в итоге у вас останется всего одна буква, которая также будет правильным английским словом, т.к. она находится в словаре. Найдите самое длинной такое слово. Сколько букв оно содержит?

Приведу пример не самого длинного слова: `sprite`. Вы начинаете со слова `'sprite'`, затем удаляете одну букву, `'r'`, и у вас остается слово `'spite'`. Далее вы удаляете `'e'` с конца, у вас остается `'spit'`. Затем мы удаляем `'s'` и получаем `'pit'`, затем `'it'` и, наконец, `'I'`.

Напишите программу, которая находит все слова, которые можно сократить подобным образом, а затем выводит самое длинное.

Поскольку данное упражнение несколько сложнее предыдущих, вот вам несколько советов:

1. Вы можете написать функцию, которая берет слово и возвращает список слов, которые можно получить, удаляя из исходного одну за одной буквы. Те слова будут как бы "потомками" исходного.
2. Рассуждая рекурсивно, слово можно сократить, если любого из его потомков можно сократить. В качестве базового значения рекурсии вы можете допустить пустую строку (которая тоже является сократимой).
3. В том списке слов, который я предоставил, `words.txt`, нет однобуквенных слов. Поэтому вы можете добавить туда буквы `"I"`, `"a"` и `""`.
4. Для улучшения производительности вашей программы, вы можете запоминать слова, являющиеся сократимыми.

Мое решение находится по адресу `thinkpython.com/code/reducible.py`.

[27] `www.cartalk.com/content/puzzler/transcripts/200651`

Глава 13

Углубленное изучение: выбор диапазона значений из структуры данных

13.1 Частотный словарь

Как обычно, вы должны, как минимум, хотя бы попытаться выполнить следующие упражнения перед тем, как заглядывать в мои решения.

Упражнение 13.1 Напишите программу, которая читает файл, разбивает каждую строку на слова, удаляет из слов все пробелы и знаки пунктуации и переводит их в нижний регистр.

Подсказка: В модуле `string` определена строковая константа, называемая `whitespace`, в которой содержится знаки пробела, табуляции, новой строки и т.п., а также `punctuation`, в которой содержатся знаки пунктуации. Давайте заставим Python попотеть:

```
>>> import string
>>> print(string.punctuation)
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Также, возможно, вы захотите использовать строковые методы `strip`, `replace` и `translate`.

Упражнение 13.2 Зайдите на сайт проекта Гуттенберг (`gutenberg.org`) и скачайте оттуда вашу любимую книгу, не защищенную авторским правом, в простом текстовом формате.

Модифицируйте вашу программу из предыдущего упражнения так, чтобы она читала вашу книгу, которую вы скачали, пропускала введение к книге в самом начале файла, и обрабатывала всю оставшуюся часть, как и в предыдущем упражнении.

Узнайте количество слов, используемых в данной книге. Сравните разные книги разных авторов, написанных в разные эпохи. Какой автор использует больше всего слов?

Упражнение 13.3 Модифицируйте вашу программу из предыдущего упражнения, чтобы она выводила бы 20 самых часто используемых слов в книге.

Упражнение 13.4 Модифицируйте вашу программу так, чтобы она читала список слов (см. раздел 9.1) и выводила все слова из книги, которые не входят в этот список. Сколько слов из них являются опечатками? Сколько из них распространенных слов, которые *должны* быть в списке слов, а сколько действительно малоупотребительных?

13.2 Случайные числа

Большинство компьютерных программ устроены так, что они выдают одни и те же результаты при одних и тех же исходных данных, поэтому говорят, что они **детерминированные** (deterministic). Как правило, детерминизм это хорошо, т.к. мы хотим, чтобы одни и те же вычисления производили бы один и тот же результат. Но существуют такие приложения, в которых мы бы хотели, чтобы компьютер вел себя непредсказуемым образом. Очевидным примером являются игры, но не только они.

На самом деле написать недетерминированную программу не так то и просто, но существуют способы заставить ее, как минимум, выглядеть недетерминированной. Одним из таких способов является использование алгоритма, выдающего **псевдослучайные** (pseudorandom) числа. Такие числа не являются действительно случайными, потому что они генерируются при помощи детерминированных вычислений, но глядя на эти числа, почти невозможно отличить их от случайных.

В модуле `random` содержатся функции для генерации псевдослучайных чисел (которые, начиная с этого места, я буду называть просто "случайные", для простоты).

Функция `random` возвращает дробное число от `0.0` до `1.0` (включая `0.0`, но не включая `1.0`). Каждый раз при вызове функции `random` вы получаете следующее число из их длинной последовательности. Чтобы посмотреть на пример использования, запустите следующий цикл:

```
import random
for i in range(10):
    x = random.random()
    print(x)
```

Функция `randint` берет параметры `low` (нижний предел) и `high` (верхний предел) и возвращает целое число в диапазоне между `low` и `high`, включая оба из них.

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Чтобы выбрать случайный элемент из последовательности, можно использовать метод `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

В модуле `random` также содержатся функции для генерации случайных величин из непрерывных распределений, включая гауссово, экспоненциальное, гамма и некоторые другие.

Упражнение 13.5 Напишите функцию `choise_from_hist`, которая берет в качестве параметра гистограмму, как она описана в разделе 11.1, и возвращает случайное значение из этой гистограммы, выбранное с вероятностью, пропорциональной его частоте. Например, для следующей гистограммы:

```
>>> t = ['a', 'a', 'b']
>>> h = histogram(t)
>>> print(h)
{'a': 2, 'b': 1}
```

ваша функция должна выдать `'a'` с вероятностью 2/3, и `'b'` с вероятностью 1/3.

13.3 Гистограммы слов

Вот программа, которая читает файл и создает гистограмму слов этого файла:

```
import string
def process_file(filename):
    h = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, h)
    return h

def process_line(line, h):
    line = line.replace('-', ' ')
    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        h[word] = h.get(word, 0) + 1

hist = process_file('emma.txt')
```

Программа читает файл `emma.txt`, в котором содержится текст *Эмма* автора Джейн Остин.

Функция `process_file` перебирает все строки файла, передавая их по очереди функции `process_line`. Гистограмма `h` используется в качестве аккумулятора.

Функция `process_line` использует метод строк `replace`, чтобы заменить все дефисы на пробелы перед использованием метода `split` для разбиения строки на список слов. Она перебирает список слов и использует методы `strip` и `lower` для удаления пунктуации и перевода слов в нижний регистр. (Проще сказать, что строки "конвертируются"; однако помните, что строки принадлежат к неизменяемому типу, поэтому методы, подобные `strip` и `low` возвращают новые строки.)

Наконец, `process_line` обновляет гистограмму либо создавая новый элемент, либо увеличивая соответствующий счетчик уже существующего.

Для подсчета общего количества слов в файле мы можем суммировать частоту их использования в гистограмме:

```
def total_words(h):
    return sum(h.values())
```

Количество разных слов равняется количеству элементов в словаре:

```
def different_words(h):
    return len(h)
```

Вот код для вывода результата:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

А вот и результат:

```
Total number of words: 161073
Number of different words: 7212
```

13.4 Самые распространенные слова

Чтобы найти самые распространенные слова, мы можем воспользоваться способом сортировки DSU; функция `most_common` принимает в качестве параметра гистограмму и возвращает список кортежей, содержащий слова и их частоты использования, отсортированный в обратном порядке по частоте:

```
def most_common(h):
    t = []
    for key, value in h.items():
        t.append((value, key))
    t.sort(reverse=True)
    return t
```

Вот цикл, который выводит десять наиболее частых слов:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[0:10]:
    print(word, '\t', freq)
```

А вот как выглядят результаты для *Эмма*:

```
The most common words are:
to    5242
the   5204
and   4897
of    4293
i     3191
a     3130
it    2529
her   2483
was   2400
she   2364
```

13.5 Опциональные параметры

Мы уже видели встроенные функции и методы, которые принимают переменное количество аргументов. Функции, написанные пользователем, также могут обладать такими свойствами. Например, вот функция, которая выводит самые часто употребляемые слова в гистограмме:

```
def print_most_common(hist, num=10)
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[0:num]:
        print(word, '\t', freq)
```

Первый параметр обязательный, а второй опциональный. **Значение по умолчанию** (default value) переменной `num = 10`.

Если вы передадите в функцию только один аргумент:

```
print_most_common(hist)
```

то переменная `num` примет значение по умолчанию. Если вы передадите два аргумента:

```
print_most_common(hist, 20)
```

то `num` получит значение этого второго аргумента (в данном случае это 20). Другими словами, опциональный аргумент **замещает** (override) значение по умолчанию.

Если в функции используются как обязательные, так и опциональные параметры, то сначала должны идти обязательные, а затем опциональные.

13.6 Вычитание словарей

Нахождение слов, используемых определенной книгой, которые, при этом, не находятся в списке слов файла `words.txt` является примером проблемы, которую можно определить как вычитание множеств данных; т.е. мы хотим найти все слова одного множества (слова в книге), которые не находятся в другом множестве (слова в списке).

Функция `subtract` принимает словари `d1` и `d2` в качестве аргументов и возвращает новый словарь, содержащий все ключи словаря `d1`, которые не содержатся в `d2`. Поскольку нас, в данном случае, не интересуют значения ключей, мы присваиваем им всем `None`.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

Чтобы найти слова в книге, которые не находятся в `words.txt`, мы можем воспользоваться функцией `process_file` для создания гистограммы `words.txt`, а затем использовать функцию `subtract`:

```
words = process_file('words.txt')
diff = subtract(hist, words)
print("The words in the book that aren't in the word list are:") for word in
diff.keys():
    print(word, end = ' ')
```

Вот некоторые результаты из книги *Эмма*:

```
The words in the book that aren't in the word list are:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Некоторые из этих слов являются именами и притяжательными конструкциями. Другие, подобные "*recontre*", уже не употребляются в настоящее время. Но есть и несколько распространенных слов, которые должны быть в списке!

13.7 Случайные слова

Для выбора случайного слова из гистограммы самым простым алгоритмом будет создание списка, в котором каждое слово дублируется количество раз, соответствующее частоте его использования, а затем выбор из этого списка:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)
    return random.choice(t)
```

Выражение `[word] * freq` создает список, в котором слово `word` встречается `freq` раз. Метод `extend` похож на метод `append`, за исключением того, что аргументом является последовательность.

Упражнение 13.7 Хотя этот алгоритм и работает, он не очень эффективен; каждый раз, когда вы выбираете случайное слово, он перестраивает весь список, который по размерам равен анализируемой книге. Очевидно, что для улучшения алгоритма можно создать список один раз, а затем использовать многократные выборки, но список все еще остается большим.

В качестве альтернативы можно:

1. Использовать `keys` для получения списка слов в книге.
2. Создать список, который будет содержать кумулятивную (накопительную) сумму частоты использования слов (см. упражнение 10.1). Последним элементом этого списка будет общее число слов в книге, *n*.
3. Выбрать случайное число от 1 до *n*. Использовать бисекционный поиск (см. упражнение 10.8) для нахождения индекса, где случайное число должно быть вставлено в кумулятивную сумму.
4. Использовать индекс для нахождения соответствующего слова в списке слов.

Напишите программу, которая использует этот алгоритм для выбора случайного слова из книги.

13.8 Анализ Маркова

Когда вы выбираете слова из книги случайным образом, то у вас получается набор слов, который больше похож на словарь, чем на предложение:

```
this the small regard harriet which knightley's it most things
```

Из последовательности случайных слов редко получается предложение, т.к. между последовательными словами нет никакой связи. Например, в английском языке после определенного артикля "*the*" как правило следует существительное или прилагательное, но не глагол или наречие.

Одним из способов определить такого рода связи является анализ Маркова[28], который для некоей последовательности слов оценивает вероятность появления следующего слова. Например, песня *Eric, Half of a Bee* начинается так:

```
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?
But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?
```

[28] Изучение этой темы основано на примере, взятом из книги Kernighan и Pike, *The Practice of Programming,* 1999.

В этом тексте после фразы "*the half*" всегда следует слово "*bee*", но после фразы "*the bee*" может быть либо "*has*", либо "*is*".

Результатом анализа Маркова является отображение каждого префикса (такого как, например, "*half the*" и "*the bee*") на все возможные суффиксы (такие как "*has*" и "*is*").

Если у вас имеется такое отображение, то вы можете сгенерировать случайный текст, который будет начинаться с любого префикса и продолжаться одним из суффиксов, выбранных случайным образом. Далее, вы можете составить конец этого префикса с новым суффиксом, чтобы получить новый префикс, и так далее.

Например, если вы начнете с префикса "*Half a*", то следующим словом должно быть обязательно "*bee*", потому что этот префикс встречается в тексте только один раз. Следующим префиксом будет "*a bee*", поэтому после него может идти один из суффиксов: "*philosophically*", "*be*" или "*due*".

В данном примере длина префикса всегда равна двум (т.е. префикс состоит из двух слов), но вы можете делать анализ Маркова с префиксами любой длины. Длина префикса называется "порядком" (order) анализа.

Упражнение 13.8 Анализ Маркова:

1. Напишите программу, которая читает текст из файла и производит анализ Маркова. В результате должен получиться словарь, который отображает префиксы на набор всех возможных суффиксов. Этот набор может быть списком, кортежем или словарем; вы можете выбрать сами подходящий тип данных. Вы можете испытать программу с длиной префикса равной двум, но в ней должна быть предусмотрена возможность легко менять порядок анализа (длину префикса).
2. В получившуюся программу добавьте функцию, которая будет генерировать случайный текст, основанный на анализе Маркова. Вот пример из текста *Emma* с длиной префикса равной 2:

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

   В данном примере я оставил пунктуацию, которая примыкала к словам. Результат выглядит почти правильным с точки зрения синтаксиса, хотя и не совсем. В смысловом отношении это предложение тоже почти несет какой-то смысл, но тоже не совсем.

   Что произойдет, если вы увеличите длину префикса? Будет ли получившийся случайный текст содержать больше смысла в таком случае?
3. Как только ваша программа заработает, вы можете попробовать провести наложение (mash-up): если вы анализируете текст из двух или более книг, то тот случайный текст, который у вас получится, может довольно интересным образом смешать слова и фразы из используемых источников.

13.9 Структуры данных

Применение анализа Маркова для генерации случайного текста приведено тут больше ради забавы, но, тем не менее, у этого упражнения есть и практическое применение: выбор диапазона значений из структуры данных. Выполняя предыдущее упражнение вы должны были решить:

- Как представить префиксы.
- Как представить набор всех возможных суффиксов.
- Как представить отображение от каждого префикса на коллекцию возможных суффиксов.

Ну хорошо, с последним пунктом все проще, т.к. единственной системой отображения, с которой мы знакомы, является словарь, поэтому его выбор будет естественным.

Для префиксов наиболее очевидным выбором будет использование строк, списков строк или кортежей строк. Для суффиксов же одним выбором будет список; другим выбором может быть гистограмма (словарь).

Каким образом вы должны осуществлять выбор? Прежде всего, вы должны подумать об операциях, которые вы будете осуществлять с каждой структурой данных. Для префиксов нам будет необходимо удалять слова из начала и добавлять их в конец. Например, если текущий префикс – "*Half a*", а следующее слово – "*bee*", то вам необходимо будет составить следующий префикс "*a bee*".

Возможно, что вашим первым выбором будет использование для этого списка, т.к. в нем легко удалять и добавлять элементы, но нам также необходимо будет использовать префиксы в качестве ключей словаря, поэтому данное обстоятельство исключает списки. С кортежами вы не можете использовать операции добавления и удаления элементов, но вы можете использовать оператор сложения для создания нового кортежа:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

Функция `shift` принимает в качестве параметров кортеж слов, `prefix`, а также строку, `word`, и возвращает новый кортеж, в котором содержатся все слова из `prefix` кроме первого, а также `word`, добавленное в конец.

Операции, которые нам необходимо совершать над набором суффиксов, включают в себя добавление новых суффиксов (или приращение счетчика частотности уже существующего суффикса), а также выбор случайного суффикса.

Добавление новых суффиксов одинаково легко осуществлять как при использовании списков, так и при использовании гистограмм. Выбирать случайный элемент из списка легко, а из гистограммы – сложнее (см. упражнение 13.7).

До сих пор мы говорили, главным образом, о вещах, которые сравнительно легко запрограммировать, но имеются и другие факторы, которые нужно будет принять во внимание при выборе структур данных. Одно из них это время. Иногда существуют теоретические причины, в силу которых одни структуры данных работают быстрее других. Например, я уже упоминал, что оператор `in` работает быстрее в словарях, чем в списках, по крайней мере, когда количество элементов велико.

Но часто вы не знаете заранее, какой способ будет работать быстрее. В подобном случае вы можете испробовать разные способы и сравнить их производительность. В таком случае вы можете провести **тест производительности** (benchmark). Практически можно поступить так: вы выбираете ту структуру данных, которую легче использовать, и смотрите, работает ли она достаточно быстро для данного приложения. Если да, то на этом можно и остановиться. Если нет, то есть средства, подобные модулю `profile`, которые могут определить те места в программе, которые занимают наибольшее время выполнения.

В качестве другого фактора необходимо рассмотреть объем занимаемого места на диске. Например, использование гистограмм для набора суффиксов может занять меньше места, т.к. каждое слово хранится только один раз вне зависимости от того, сколько раз оно появляется в тексте. В некоторых случаях экономия дискового пространства может увеличить скорость выполнения вашей программы, а в некоторых исключительных случаях ваша программа может не запуститься вообще, если вы израсходуете всю память. Но для многих приложений в первую очередь необходимо учитывать время выполнения, а уже потом – объем занимаемых данных.

И еще одна мысль в заключение: в этой дискуссии я подразумевал, что одна и та же структура данных будет использоваться и в анализе, и в генерации. Но, поскольку существуют разные фазы выполнения, вы можете использовать одну структуру для анализа, а затем конвертировать ее в другую структуру для генерации. Может получиться так, что время, потраченное на конвертацию, окупится тем, что ваша программа будет работать, в конечном итоге, быстрее.

13.10 Отладка

Когда вы отлаживаете программу, особенно, если вы работаете над трудноуловимой ошибкой, есть четыре вещи, которые вы можете попытаться сделать:

чтение: исследуйте ваш код, прочитайте его заново самому себе, и проверьте, что он означает именно то, что вы намеревались сделать.

выполнение: поэкспериментируйте, внося изменения и запуская различные версии. Часто проблема становится очевидной, стоит только вывести нужный результат в нужном месте программы, но иногда вам нужно потратить время на написание отладочного кода.

тщательное обдумывание: потратьте некоторое время на то, чтобы все тщательно обдумать! К какому типу относится данная ошибка: синтактическая, во время выполнения или смысловая? Какую полезную информацию вы можете извлечь из сообщений об ошибках или из вывода программы? Какой тип ошибки может вызвать тут проблему, с которой вы столкнулись. Что вы меняли в последний раз перед тем, как возникла проблема?

откат изменений: в некоторых случаях самое лучшее, что вы можете сделать это остановиться, отменить все недавние изменения, которые вы сделали, пока у вас не получится рабочая программа, которую вы понимаете. Потом можно снова начать модификацию.

Начинающие программисты часто зацикливаются на каком-то одном из вышеперечисленных методов устранения ошибок и забывают о других. Но каждый из методов может помочь лишь в определенном случае.

Например, чтение кода может помочь тогда, когда проблема кроется в типографической ошибке, но не тогда, когда она заключается в непонимании концепции. Если вы не понимаете то, что делает программа, вы можете перечитать ее 100 раз и, тем не менее, не увидеть ошибки, потому что ошибка – в вашей голове.

Выполнение экспериментов также способно помочь, особенно, если вы выполняете маленькие и простые тесты. Но если вы запускаете эксперименты не особо задумываясь, или не читаете ваш код, то это будет то, что я называю "случайное программирование", т.е. внесение случайных изменений до тех пор, пока программа не начнет выполняться правильно. Нет нужды говорить, что случайное программирование может занять значительное время.

Вам необходимо выделить время для обдумывания. Отладка похожа на экспериментальную науку. У вас должна быть, как минимум, одна гипотеза о том, в чем заключается проблема. Если есть две и более вероятностей, то попытайтесь придумать тест, который исключил бы одну из них.

В процессе раздумий помогает перерыв. Также и разговор. Если вы объясните проблему кому-нибудь еще (хотя бы самому себе), иногда вы можете найти ответ даже до того, как закончите задавать вопрос.

Но даже самые лучшие методики отладки не помогут, если имеется слишком много ошибок, или код, который вы пытаетесь исправить, слишком большой и сложный. Иногда лучшим выбором будет сделать откат изменений, упростив вашу программу до состояния, когда она работает и вы ее понимаете.

Начинающие программисты часто нехотя делают откаты, т.к. они не любят удалять даже одну строчку их кода (даже если она ошибочная). Если вам от этого будет проще, скопируйте вашу программу в другой файл перед тем, как вы начнете его сокращать. Потом вы можете вставлять обратно по кусочку за раз.

Нахождение трудного бага требует чтения, выполнения, тщательного обдумывания и, иногда, отката изменений. Если вы застопорились на чем-то одном, то попробуйте другое.

13.11 Словарь терминов

детерминистический (deterministic): термин относится к программе, которая делает то же самое всякий раз, когда ее запускают с одинаковыми входными данными.

псевдослучайный (pseudorandom): относится к последовательности чисел, которые выглядят случайными, но, тем не менее, сгенерированы детерминистической программой.

значение по умолчанию (default value): значение, присваиваемое опциональному параметру, если не был задан аргумент.

замещение (override): замена значения по умолчанию значением переданного аргумента.

тест производительности (benchmark): процесс выбора оптимальной структуры данных, применяя разный подход при кодировании, и затем сравнивая производительность на одинаковых входных данных.

13.12 Упражнения

Упражнение 13.9 Рангом (rank) слова называют его положение в списке слов, отсортированном по частоте их использования: самое частое слово имеет ранг 1, следующее – ранг 2 и т.д.

Закон Зипфа (Zipf's law) описывает отношения между рангами и частотностями слов естественного языка[29].

В частности, закон утверждает, что частотность f слова с рангом r подчиняется формуле:

$$f = cr^{-s}$$

где c и s – параметры, зависящие от языка и текста. Если вы вычислите логарифм обоих частей уравнения, то у вас получится следующее:

$$\log f = \log c - s \log r$$

Поэтому если вы построите и сравните графики $\log f$ и $\log r$, у вас должна получиться прямая линия с наклоном –s и отрезком $\log c$.

Напишите программу, которая читает текст из файла, подсчитывает частотность слов и выводит каждое слово на отдельной строке в нисходящем порядке с $\log f$ и $\log c$. Используйте любую программу для построения графиков для отображения результата и проверьте, действительно ли у вас получилась прямая линия. Можете ли вы подсчитать значение s?

[29] См. `wikipedia.org/wiki/Zipf's_law`.

Глава 14
Файлы

14.1 Персистентность (устойчивость)

Большинство программ, видимых нами до сих пор, были кратковременными. Другими словами, они выполнялись некоторое время и выводили некоторый результат, но когда их выполнение заканчивалось, их данные пропадали. Если вы снова запустите такую программу, то она начнет все сначала с чистого состояния.

Другие программы обладают свойством **персистентности** (persistence): они выполняются в течение некоторого времени (или постоянно); они хранят, как минимум, некоторые свои данные на долговременных устройствах хранения данных (например, на жестком диске); если их остановить и перезапустить, то они могут начать с того места, где они остановились.

Примером таких программ, обладающих персистентностью, являются операционные системы, которые работают, практически, все время, пока включен компьютер, или веб-серверы, которые работают постоянно, ожидая запросов по сети.

Самым простым способом сохранять и извлекать подобные данные является чтение и запись текстовых файлов. Мы уже видели программы, которые могут читать текстовые файлы. В этой главе мы познакомимся с программами, которые могут также записывать данные в файлы.

Альтернативным способом хранения состояния программы является использование баз данных. В этой главе я продемонстрирую простую базу данных и модуль `pickle`, который позволяет легко хранить данные программы.

14.2 Чтение и запись

Текстовый файл представляет из себя последовательность символов, которые могут храниться на жестком диске, флешке или CD-ROM. В разделе 9.1 мы видели, как можно открывать и читать такой файл.

Чтобы в файл можно было записывать, вы должны открыть его в режиме `'w'` в качестве дополнительного параметра:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Будьте внимательны! Если такой файл уже существует, при открытии его в режиме записи все его прежние данные, содержащиеся в нем, уничтожаются, и он готов принимать новые данные. Если же такой файл не существует, то создается новый файл с таким именем.

Метод `write` позволяет записывать данные в файл:

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Опять же, объект `file` ведет учет того, где производится запись, поэтому если вы снова вызовете метод `write`, то он добавит новые данные к концу файла:

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

Если вы закончили производить запись, то файл нужно закрыть:

```
>>> fout.close()
```

14.3 Оператор форматирования

Аргумент метода `write` должен быть строкового типа, поэтому для того, чтобы записать какие-либо данные в файл, их необходимо конвертировать в строку. Проще всего это сделать при помощи функции `str`:

```
>>> x = 52
>>> f.write(str(x))
```

В качестве альтернативы можно воспользоваться **оператором форматирования** (format operator), `%`. Когда этот оператор применяется к целым числам, то он вычисляет остаток от целочисленного деления. Но если первый операнд будет строкой, то `%` будет оператором форматирования.

При использовании его таким образом, первый операнд будет представлять из себя **форматируемую строку** (format string), содержащую одну или более **форматирующих последовательностей** (format sequence), которые определяют, как будет отформатирован второй операнд. В результате получится строка `string`.

Например, форматирующая последовательность '`%d`' означает, что второй операнд должен быть отформатирован как целое число (`d` здесь означает *decimal*):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Результат – '`42`' – строка, а не целое число 42.

Форматирующая последовательность может быть в любом месте строки, поэтому вы можете вставлять значения в предложение:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Если строка содержит больше одной форматирующей последовательности, то второй аргумент должен быть кортежем. Каждая форматирующая последовательность должна соответствовать элементу кортежа в своем порядке.

В следующем примере '`%d`' используется для форматирования целых чисел, '`%g`' для форматирования чисел с десятичной точкой (не спрашивайте меня, почему), и '`%s`' для форматирования строки:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

Количество элементов кортежа должно совпадать с количеством форматирующих последовательностей в строке. Подобным же образом должны совпадать и их типы:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

В первом примере в кортеже недостаточно элементов; во втором – элемент неправильного типа.

Это довольно мощный оператор, но он может быть несколько труден в использовании. Вы можете прочитать о нем больше в документации: `docs.python.org/lib/typesseq-strings.htm`

14.4 Имена файлов и пути

Файлы хранятся в **директориях** (directory), также называемых "папками". Каждая выполняющаяся программа имеет так называемую "текущую директорию" (current directory), которая является директорией по умолчанию для многих операций. Например, когда вы открываете файл для чтения, Python ищет его в текущей директории.

Модуль `os` предоставляет функции для работы с файлами и директориями (os означает "операционная система"). `os.getcwd` возвращает имя текущей директории:

```
>>> import os
>>> cwd = os.getcwd()
>>> print(cwd)
/home/dinsdale
```

Здесь `cwd` означает "current working directory" (текущая рабочая директория). В данном примере результатом является `/home/dinsdale`, т.е. домашняя директория пользователя с именем `dinsdale`.

Строка, подобная `cwd`, которая используется для идентификации файла, называется **путем** (path) к файлу. **Относительный путь** (relative path) начинается с текущей директории; **абсолютный путь** (absolute path) начинается с самой верхней директории файловой системы.

Пути, которые мы до сих пор видели, представляют из себя просто имена файлов, поэтому они относительны к текущей директории. Чтобы найти абсолютный путь к файлу, используйте `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path.exists` проверяет, существует ли данный файл или директория:

```
>>> os.path.exists('memo.txt')
True
```

`os.path.isdir` проверяет, является ли данный файл директорией:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Аналогично, `os.path.isfile` проверяет, является ли файл именно файлом, а не директорией.

`os.listdir` возвращает список файлов (и других директорий) в данной директории:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

Для демонстрации всех этих функций следующий пример "проходит" (walk) через директорию, выводя имена всех файлов и вызывая себя рекурсивно на все директории:

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` берет имя директории и файла и соединяет их в полный путь.

Упражнение 14.1 Модифицируйте функцию `walk` таким образом, чтобы вместо того, чтобы просто выводить имена файлов, она возвращала бы их в виде списка.

Упражнение 14.2 В модуле `os` имеется функция `walk`, которая аналогична нашей, но более универсальна. Прочитайте документацию к ней и используйте ее для вывода имен файлов в данной директории и поддиректориях.

14.5 Перехват ошибок

Много чего может пойти не так, когда вы читаете или пишете в файлы. Если вы попытаетесь открыть несуществующий файл, вы получите ошибку вида `IOError` (ошибка ввода/вывода):

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

Если у вас нет доступа к файлу:

```
>>> fout = open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

Если вы попытаетесь открыть директорию для чтения:

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

Для предотвращения подобных ошибок вы можете пользоваться функциями, такими как `os.path.exists` или `os.path.isfile`, но это отнимет много времени и кода для проверки всех возможных вероятностей (в примере выше "`Errno 21`" означает, что есть, как минимум, 21 возможных ошибок, которые могут произойти).

В подобных случаях гораздо лучше просто продолжать выполнение, а с ошибками разбираться, если они появятся. Именно это и делает команда `try`. Ее синтаксис аналогичен синтаксису `if`:

```
try:
    fin = open('bad_file')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Произошла какая-то ошибка.')
```

Python начинает выполнение выражений, идущих за `try`. Если все прошло хорошо, то он пропускает выполнение выражений, идущих за `except` и идет дальше. Если же происходит какая-либо ошибка, то он прекращает выполнение выражений `try` и переходит к выполнению выражений `except`.

Это называется **перехватом** (catching) ошибок. В данном примере выражение, следующее за `except`, выводит сообщение об ошибке, которое дает нам не слишком много информации о том, какая именно ошибка произошла. В общем случае, перехват ошибок дает вам возможность исправить проблему, попытаться заново или, как минимум, завершить программу безаварийно.

14.6 Базы данных

Базой данных (database) называется файл, организованный для хранения данных. Большинство баз данных организованно подобно словарям в том смысле, что они отображают имена ключей на их значения. Самым большим различием является то, что базы данных хранятся на диске (или другом долговременном хранилище), поэтому они существуют и после того, как программа завершит выполнение.

Модуль `dbm` предоставляет интерфейс для создания и обновления файлов баз данных.

Открытие базы данных подобно открытию любого другого файла:

```
>>> import dbm
>>> db = dbm.open('captions.db', 'c')
```

Режим '`c`' означает, что база данных должна быть создана (create) , если уже не существовала до этого. Результатом является объект `database`, который можно использовать (для большинства операций) подобно словарю. Если вы создадите новую запись, `dbm` обновляет файл базы данных:

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

Когда вы запрашиваете доступ к одному из элементов, `dbm` читает файл базы данных (обратите внимание, что `dbm` хранит ключи и их значения как байты, на что указывает буква '`b`'):

```
>>> print(db['cleese.png'])
b'Photo of John Cleese.'
```

Если вы присвоите новое значение существующему ключу, то `dbm` замещает им старое значение:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print(db['cleese.png'])
b'Photo of John Cleese doing a silly walk.'
```

Многие методы словарей, такие как `keys` и `items` работают также и с базами данных. Вот, например, пример перебора ключей в цикле `for`:

```
for key in db:
    print(key)
```

Как и с другими файлами, вы должны закрыть базу данных, когда закончите с ней работу:

```
>>> db.close()
```

14.7 Сериализация и десериализация

Ограничение `dbm` состоит в том, что ключи и значения должны быть байтами (строки автоматически конвертируются в байты). Если вы попытаетесь использовать любой другой тип, вы получите сообщение об ошибке.

Здесь на помощь приходит модуль `pickle`. Он переводит почти любые типы данных в байты, подходящие для хранения в базе данных, а затем переводит строки обратно в объекты. Данные операции называются **сериализацией** и **десериализацией** соответственно. Еще сериализацию называют **консервированием**.

`pickle.dumps` берет некоторый объект в качестве параметра и возвращает его же, но представленным в виде байтов (dumbs означает "*dump bytes*"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

Для человека последняя запись выглядит несколько странно, но она предназначена не для чтения человеком, а для чтения модулем `pickle`.

`pickle.loads` ("*load bytes*") реконструирует объект:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print(t2)
[1, 2, 3]
```

Хотя новый объект имеет такое же значение, что и старый, в общем случае это не тот же самый объект:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

Другими словами, такая операция кодирования, а затем декодирования имеет то же значение, что и копирование объектов.

Вы можете использовать `pickle` для хранения в базе данных объектов, не являющихся строками. На самом деле это используется настолько часто, что для этого был специально создан модуль `shelve`.

Упражнение 14.3 Если вы делали упражнение 12.4, измените ваш код таким образом, чтобы он создавал базу данных, которая отображала бы каждое слово в списке на список слов, в которых используется тот же самый набор букв.

Напишите другую программу, которая открывала бы файл этой базы данных и выводила ее содержимое в формате, удобном для восприятия человеком.

14.8 Конвейеры

Большинство операционных систем предоставляют интерфейс командной строки, который еще называют **оболочкой** (shell). Обычно оболочки предоставляют команды для навигации по файловой

системе и запуску приложений. Например, в Unix вы можете изменить директорию командой `cd`, отобразить содержимое директории командой `ls` и запустить, скажем, веб-браузер, скажем, Firefox, просто напечатав его имя: `firefox`.

Любую программу, которую вы можете запустить из оболочки, можно также запустить и из Python при помощи **конвейера** (pipe). Конвейер является объектом, который представляет из себя исполняющийся процесс.

Например, в Unix команда `ls -l`, как правило, выводит содержимое текущей директории (в длинном формате). Вы можете запустить `ls` с помощью `os.popen` следующим образом:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

Аргумент является строкой, представляющей из себя команду оболочки. Возвращаемым значением является объект, который ведет себя как открытый файл. Вы можете прочитать вывод процесса `ls` построчно при помощи `readline` или получить все целиком при помощи `read`:

```
>>> res = fp.read()
```

Когда вы закончили, вы должны закрыть конвейер так же, как и обычный файл:

```
>>> stat = fp.close()
>>> print(stat)
None
```

Возвращаемое значение здесь это окончательное состояние процесса `ls`; `None` означает, что он закончился нормально, т.е. без ошибок.

Часто конвейеры используются для постепенного чтения сжатых файлов, т.е. без того, чтобы предварительно их полностью разархивировать в определенное место. Следующая функция принимает имя сжатого файла в качестве параметра и возвращает конвейер, который использует `gunzip` для декомпрессии содержимого:

```
def open_gunzip(filename):
    cmd = 'gunzip -c ' + filename
    fp = os.popen(cmd)
    return fp
```

Если вы будете последовательно читать строки из объекта `fp`, вам никогда не нужно будет хранить полностью разархивированный файл в памяти или на диске.

14.9 Создание собственных модулей

Любой файл, который содержит код Python, можно импортировать в качестве модуля. Предположим, для примера, что у вас имеется файл с названием `wc.py`, содержащий следующий код:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

Если вы запустите эту программу, то она прочитает свой собственный текст и выведет количество строк в файле, равное, в данном случае, 7. Вы также можете импортировать ее следующим образом:

```
>>> import wc
7
```

Теперь у вас есть объект `module` под названием `wc`:

```
>>> print(wc)
<module 'wc' from 'wc.py'>
```

Он предоставляет функцию под названием `linecount`:

```
>>> wc.linecount('wc.py')
7
```

Вот так создаются модули в Python.

Единственной проблемой в данном модуле является то, что когда вы импортируете его, тестовый код в самом низу запускается на выполнение. Обычно, когда вы импортируете какой-либо модуль, то при этом лишь определяются все содержащиеся в нем функции, но они не запускаются на выполнение.

Поэтому в программах, которые предназначены для импорта в качестве модулей, часто используется следующая идиома:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

Здесь `__name__` это имя встроенной переменной, значение которой устанавливается как только программа запускается. Если программа выполняется в качестве скрипта, то значение переменной `__name__` будет `__main__`. В таком случае тестовый код будет выполнен. В противном случае, т.е. если модуль был импортирован, тестовый код не будет выполняться.

Упражнение 14.4 Поместите этот пример в файл с именем `wc.py` и запустите его как скрипт. Затем, запустите интерпретатор Python и импортируйте `wc`. Каково значение `__name__` , когда модуль был импортирован?

Предупреждение: если вы импортируете модуль, который уже был импортирован, то Python ничего не будет делать в этом случае. Он не будет перечитывать файл заново, даже если в него были внесены изменения.

Если вы хотите заново перезагрузить модуль, то вы можете воспользоваться встроенной функцией `reload`, однако, при этом могут возникнуть некоторые нюансы, поэтому проще и безопаснее просто перезапустить интерпретатор Python и затем импортировать модуль снова.

14.10 Отладка

Когда вы читаете или записываете в файлы, вы можете столкнуться с проблемами, вызванными непечатаемыми символами. Такие ошибки бывает трудно отловить, потому что пробелы, знаки табуляции и новой строки, как правило, невидимы:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2   3
 4
```

В таких случаях вам может помочь встроенная функция `repr`, которая берет любой объект в качестве аргумента и возвращает его строковое представление. В случае со строками она представляет непечатаемые символы последовательностью из обратной косой черты и символа:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Это может пригодиться при отладке.

Другая проблема может быть вызвана тем, что в разных операционных системах используются разные символы для обозначения конца строки. Некоторые системы используют для этого символ новой строки, обозначаемый `\n`. Другие используют символ возврата каретки, `\r`. Некоторые системы используют оба символа. Если вы обрабатываете файлы на разных операционных системах, то это может причинить вам проблемы.

Для большинства систем существуют приложения, конвертирующие один формат в другой. Вы можете узнать больше о них и о данной проблеме по адресу: `wikipedia.org/wiki/Newline`

Разумеется, вы можете написать и ваше собственное приложение.

14.11 Словарь терминов

персистентность (persistent): свойство программам, которые выполняются бесконечно долго и хранят, как минимум, некоторые свои данные в долговременной памяти (типа жесткого диска).

оператор форматирования (format operator): оператор `%`, который принимает форматируемую строку и кортеж, и генерирует строку, включающую элементы кортежа, отформатированные так, как это было определено в форматируемой строке.

форматируемая строка (format string): строка, используемая с оператором форматирования, содержащая форматирующие последовательности.

форматирующая последовательность (format sequence): последовательность символов в форматируемой строке, таких как `%d`, которая определяет то, как должна быть отформатирована строка.

текстовый файл (text file): последовательность символов, хранимых в долговременной памяти, такой как жесткий диск.

директория (directory): именованная коллекция файлов; также имеет другое название – *папка* (folder).

путь (path): строка, идентифицирующая файл.

относительный путь (relative path): путь, который начинается с текущей директории.

абсолютный путь (absolute path): путь, который начинается с самой верхней директории файловой системы.

перехват (catch): предотвращение аварийного завершения программы из-за ошибки путем использования выражений `try` и `except`.

база данных (database): файл, содержимое которого организовано в виде словаря с ключами и соответствующими им значениями.

сериализация (консервирование): конвертирование произвольного объекта в последовательность байтов для последующего его хранения на жестком диске.

десериализация: операция, обратная сериализации.

14.12 Упражнения

Упражнение 14.5 Модуль `urllib.request` предоставляет методы для манипулирования адресами URL и загрузки информации из интернета. Следующий пример загружает и выводит на экран секретное сообщение с сайта `thinkpython.com`:

```
import urllib.request
conn=urllib.request.urlopen('http://thinkpython.com/secret.html')
for line in conn.fp:
    print(line.strip())
```

Выполните этот код и следуйте инструкциям, которые вы увидите.

Упражнение 14.6 В большой коллекции MP3 файлов может встретиться более одной копии одной и той же песни, которая будет храниться в разных директориях под разными именами. Целью данного упражнения является поиск таких дубликатов.

1. Напишите программу, которая будет осуществлять рекурсивный поиск в директории и всех ее поддиректориях и возвращать список полных путей файлов с данным суффиксом (в нашем случае, .mp3). Подсказка: `os.path` предоставляет несколько полезных функций для манипуляции с файлами и их путями.
2. Для того, чтобы распознать дубликаты файлов, можно воспользоваться хэш-функцией, которая читает файл и генерирует краткую сумму его содержимого. Например, MD5 (Message-Digest algoritm 5) берет произвольной длины "сообщение" и возвращает его 128-битную "контрольную сумму". Вероятность того, что два файла с разным содержанием вернут одинаковую контрольную сумму, крайне мала.
   Вы можете прочитать об алгоритме MD5 на википедии `wikipedia.org/wiki/Md5`. В Linux вы

можете использовать программу `md5sum` (в Mac OS X – `md5`) и конвейер, чтобы проверять контрольную сумму из Python.

Упражнение 14.7 База данных фильмов в интернете (IMDb) представляет из себя онлайн-коллекцию информации о фильмах. Их база данных доступна в обычном текстовом формате, поэтому ее достаточно просто читать средствами Python. Для этого упражнения вам понадобятся файлы `actors.list.gz` и `actresses.list.gz`. Вы можете скачать их по адресу `www.imdb.com/interfaces#plain`.

Я написал программу, которая осуществляет парсинг этих файлов и разделяет их на имена актеров, названия фильмов и т.д. Вы можете скачать программу по адресу `thinkpython.com/code/imdb.py`

Если вы запустите `imdb.py` как скрипт, он прочитает `actors.list.gz` и напечатает по одной паре актер-фильм на строку. Или вы можете импортировать модуль `imdb`, после чего вы можете использовать функцию `process_file` для обработки файла. Аргументами являются имя файла, объект-функция и опциональное число строк для обработки. Вот пример использования:

```
import imdb
def print_info(actor, date, title, role):
    print(actor, date, title, role)

imdb.process_file('actors.list.gz', print_info)
```

Когда вы вызываете функцию `process_file`, она открывает файл, читает его содержимое и вызывает, в свою очередь, функцию `print_info` по одному разу на каждую строку файла. `print_file` принимает в качестве аргументов имя актера, дату, название фильма и роль, а затем выводит их.

1. Напишите программу, которая читает файлы `actors.list.gz` и `actresses.list.gz` и использует модуль `shelve` для построения базы данных, в котором каждый актер отображается на список фильмов, в которых он участвует.
2. Напишите программу, которая на основе базы данных, которую вы получили на предыдущем шаге, будет составлять другую базу данных, в которой каждый актер отображается на список других актеров, с которыми первый снимался хотя бы в одном фильме.
3. Напишите программу, которая будет исполнять "*Шесть градусов Кевина Бэкона*" (Six Degrees of Kevin Bacon), о чем вы можете прочитать на `wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`. Это непростая проблема, потому что она потребует от вас нахождения кратчайшего пути в графе. Об алгоритмах поиска кратчайшего пути в графе вы можете прочитать в `wikipedia.org/wiki/Shortest_path_problem`.

Глава 15
Классы и объекты

15.1 Типы данных, определяемые пользователем

Мы уже пользовались многими встроенными типами Python. Пришло время создать свой собственный тип. В качестве примера мы создадим тип под названием `Point`, который будет представлять из себя точку в двухмерном пространстве.

В математике для задания точки часто используют круглые скобки с координатами, разделенными запятой. Например, `(0,0)` обозначает начало координат, а `(x,y)` обозначает точку, которая лежит *x* единиц правее и *y* единиц вверх от начала координат.

Для представления точки в Python есть несколько способов:

- Можно хранить координаты отдельно в переменных *x* и *y*.
- Можно хранить координаты в качестве элементов списка или кортежа.
- Можно создать новый тип данных, представляющий точку в качестве объекта.

Создание нового типа несколько сложнее первых двух способов, но преимущества этого подхода вскоре станут очевидны.

Тип данных, определяемый пользователем, также называется **классом** (class). Определение класса выглядит примерно так:

```
class Point(object):
    """представляет точку в двухмерном пространстве."""
```

Заголовок указывает на то, что новый класс называется `Point`, который является одним из видов объектов (`object`), который, в свою очередь, является встроенным типом.

В теле класса находится строка документации, которая кратко объясняет, для чего предназначен данный класс. Внутри класса вы можете определить переменные и функции, но к этому мы вернемся позже.

Определение класса `Point` создает объект типа `class`:

```
>>> print(Point)
<class '__main__.Point'>
```

Т.к. класс `Point` был определен на самом верхнем уровне, его "полным именем" является `__main__.Point`.

Объект типа `class` похож на фабрику по созданию других объектов. Чтобы создать другой объект типа `Point`, вы должны вызвать `Point` так, как если бы он был функцией.

```
>>> blank = Point()
>>> print(blank)
<__main__.Point object at 0xb7e9d3ac>
```

Возвращаемое значение здесь это ссылка на объект `Point`, который мы присвоили переменной `blank`. Создание нового объекта называется **созданием экземпляра класса** (instantination), а созданный таким образом объект называется **экземпляром класса** (instance)**.**

Когда вы применяете функцию `print` к экземпляру класса, Python сообщает вам, к какому классу принадлежит данный объект и где он находится в памяти (префикс `0x` означает, что следующее за ним число записано в шестнадцатеричном формате).

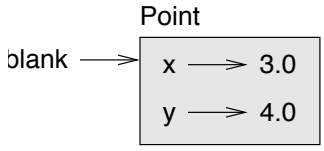
15.2 Атрибуты

Вы можете присваивать значения экземпляру класса при помощи точечной записи:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

Подобный синтаксис похож на тот, что используется для доступа к переменным, определенным в каком-либо модуле, например, `math.pi` или `string.whitespace`. Хотя в нашем случае мы присваиваем значения элементам объекта. Эти элементы называются **атрибутами** (attribute).

Следующая диаграмма показывает результат такого присваивания. Диаграмма состояния, которая изображает объект и его атрибуты, называется **диаграммой объекта** (object diagram):



Переменная `blank` ссылается на объект `Point`, который содержит два атрибута. Каждый атрибут ссылается на число с плавающей точкой.

Чтобы прочитать значение атрибута, можно использовать тот же синтаксис:

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

Выражение `blank.x` означает "иди к объекту, на который ссылается переменная `blank`, и получи значение `x`". В данном случае мы присваиваем значение переменной под названием `x`. Между переменной `x` и атрибутом `x` нет никакого конфликта.

Вы можете использовать точечную запись как часть любого выражения, например:

```
>>> print('(%g, %g)' % (blank.x, blank.y))
(3, 4)
>>>  distance = math.sqrt(blank.x**2 + blank.y**2)
>>>  print(distance)
5.0
```

Вы можете передавать экземпляр класса в качестве аргумента функции самым обычным образом, например:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

Функция `print_point` принимает объект типа `Point` в качестве аргумента и выводит его в математической записи. Например, вы можете передать в качестве аргумента `blank`:

```
>>> print_point(blank)
(3.0, 4.0)
```

Внутри этой функции `p` является синонимом `blank`, поэтому если функция изменяет `p`, то и `blank` меняется также.

Упражнение 15.1 Напишите функцию под названием `distance`, которая принимает два объекта типа `Point` и возвращает расстояние между ними.

15.3 Прямоугольники

В некоторых случаях совершенно очевидно, какими должны быть атрибуты объекта, но в других случаях нужно делать выбор из нескольких вариантов. Например, представьте, что вы разрабатываете

класс для представления прямоугольников. Какие атрибуты вы бы использовали для того, чтобы задать положение и размер прямоугольника? Для упрощения задачи давайте не будем принимать во внимание возможный угол поворота. Просто будем считать, что наши прямоугольники ориентированы либо горизонтально, либо вертикально.

Имеется, как минимум, две возможности:

- Вы можете задать один из углов прямоугольника (или центр), а также ширину и высоту.
- Вы можете задать два противоположных угла.

На данном этапе довольно сложно сказать, какой из этих двух способов лучше, поэтому мы остановимся на первом варианте, просто в качестве примера.

Вот как выглядит определение класса:

```
class Rectangle(object):
    """Представляет прямоугольник.
           атрибуты: width, height, corner.
    """
```

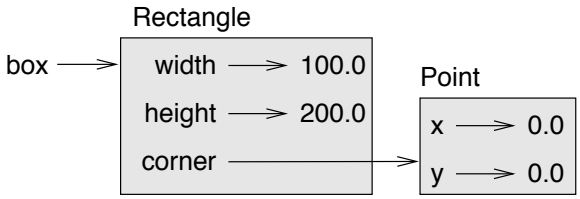
Строка документации перечисляет атрибуты: `width` (ширина) и `height` (высота) являются числами; `corner` (угол) является объектом типа `Point`, который задает нижний левый угол.

Чтобы создать прямоугольник, вам необходимо создать экземпляр класса `Rectangle` и присвоить значения его атрибутам:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Выражение `box.corner.x` означает "иди к объекту, на который ссылается переменная `box`, и выбери атрибут под названием `corner`; затем иди к этому объекту и выбери атрибут `x`."

Следующая диаграмма показывает состояние этого объекта:



Объект, который является атрибутом другого объекта, называется **вложенным** (embedded).

15.4 Экземпляры класса в качестве возвращаемых значений

Функции могут возвращать экземпляры классов. Например, следующая функция `find_center` принимает объект типа `Rectangle` в качестве аргумента и возвращает объект типа `Point`, который содержит координаты центра прямоугольника `Rectangle`:

```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

Вот пример, в котором `box` передается в качестве аргумента и присваивает возвращаемое значение типа `Point` переменной `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

15.5 Объекты принадлежат к изменяемым типам

Вы можете изменить состояние объекта, применив операцию присваивания к одному из его атрибутов. Например, чтобы изменить размер прямоугольника без изменения его положения, вы можете изменить значения `width` и `height`:

```
box.width = box.width + 50
box.height = box.width + 100
```

Также вы можете писать и функции, которые будут изменять объекты. Например, `grow_rectangle` принимает объект типа `Rectangle` и два числа, `dwidth` и `dheight` и прибавляет эти значения к ширине и высоте прямоугольника соответственно:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Вот пример, демонстрирующий ее работу:

```
>>> print(box.width) 100.0
>>> print(box.height) 200.0
>>> grow_rectangle(box, 50, 100)
>>> print(box.width)
150.0
>>> print(box.height)
300.0
```

Внутри функции `rect` является синонимом `box`, поэтому если функция модифицирует `rect`, то `box` тоже изменяется.

Упражнение 15.2 Напишите функцию `move_rectangle`, которая принимает объект типа `Rectangle` и два числа, `dx` и `dy`. Она должна изменять положение прямоугольника, добавляя `dx` к координате `x`, а `dy` к координате `y` угла `corner`.

15.6 Копирование

Использование синонимов может затруднить чтение программы, потому что изменения в одном месте могут вызывать непредсказуемые последствия в другом. Достаточно трудно отслеживать все переменные, которые могут ссылаться на данный объект.

В качестве альтернативы синонимам часто выступает копирование объектов. Модуль `copy` содержит функцию `copy`, которая создает дубликат любого объекта:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` и `p2` содержат одинаковые данные, но это разные объекты типа `Point`.

```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

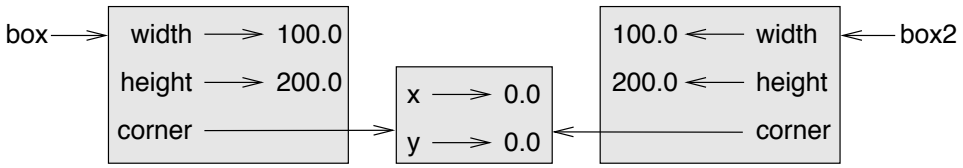
Оператор `is` демонстрирует нам здесь, что `p1` и `p2` являются разными объектами, что вполне соответствует нашим ожиданиям. Но вы могли также ожидать, что оператор == произведет значение `True`, т.к. в `p1` и `p2` содержатся одинаковые данные. Но в данном случае вы, вероятно, несколько удивитесь, узнав, что когда этот оператор применяется к экземплярам классов, его действие по

умолчанию не отличается от действия оператора `is`: он проверяет идентичность объектов, а не их равенство. Такое поведение можно изменить, и позднее мы увидим, как это можно сделать.

Если вы используете `copy.copy` для создания дубликата объекта типа `Rectangle`, то вы можете обнаружить, что при этом копируется сам объект типа `Rectangle`, но не встроенный объект типа `Point`.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Вот как выглядит диаграмма объектов:



Такая операция называется **мелким копированием** (shallow copy), потому что она копирует объект и все ссылки, которые он содержит, но не затрагивает внедренные объекты.

Для большинства приложений это не совсем то, что вам нужно. В этом примере вызов функции `grow_rectangle` с одним из объектов типа `Rectangle` в качестве параметра не затронет другой, но вызов `move_rectangle` с любым из них затронет их оба! Такое поведение приводит к путанице и чревато ошибками.

К счастью, модуль `copy` содержит метод, который называется `deepcopy`, который копирует не только сам объект, но и другие объекты, на который тот ссылается, а также другие объекты, на которые ссылаются *те* объекты, и т.д. Наверное, вы не слишком удивитесь, узнав, что такая операция называется **глубоким копированием** (deep copy).

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` и `box` – совершенно разные объекты.

Упражнение 15.3 Напишите такую версию `move_rectangle`, которая вместо того, чтобы модифицировать старый объект типа `Rectangle`, создает и возвращает новый.

15.7 Отладка

Когда вы начинаете работать с объектами, то у вас, по всей вероятности, будут возникать некоторые новые виды ошибок. Если вы попытаетесь получить доступ к несуществующему атрибуту, то вы получите сообщение об ошибке типа `AttributeError`:

```
>>> p = Point()
>>> print(p.z)
AttributeError: Point instance has no attribute 'z'
```

Если вы не уверены в том, к какому типу принадлежит некий объект, вы можете узнать это:

```
>>> type(p)
<class '__main__.Point'>
```

Если вы не уверены, имеется ли у некоторого объекта определенный атрибут, вы можете вызвать встроенную функцию `hasattr`:

```
>>> hasattr(p, 'x')
True
```

```
>>> hasattr(p, 'z')
False
```

Первым аргументом может быть любой объект; вторым аргументом должна быть *строка* (тип `string`), содержащая имя атрибута.

15.8 Словарь терминов

класс (class): тип данных, определяемый пользователем; определение класса создает новый объект типа `class`.

объект типа `class` (class object): объект, который содержит информацию о типе данных, определенных пользователем. Объект типа `class` можно использовать для создания экземпляров объектов этого типа.

экземпляр класса (instance): объект, который принадлежит к какому-либо классу.

атрибут (attribute): одно из именованных значений, ассоциируемых с объектом.

внедренный объект (embedded object): объект, который используется в качестве атрибута другого объекта.

мелкое копирование (shallow copy): копирование содержимого объекта, включая ссылки на внедренные объекты; осуществляется при помощи функции `copy` из модуля `copy`.

глубокое копирование (deep copy): копирование как содержимого объекта, так и любых внедренных объектов, а также любых объектов, внедренных в них и т.д.; осуществляется при помощи функции `deepcopy` модуля `copy`.

диаграмма объектов (object diargam): диаграмма, показывающая объекты, их атрибуты и значения атрибутов.

15.9 Упражнения

Упражнение 15.4 Файл `World.py`, являющийся частью `Swampy` (см. главу 4), содержит определение класса, определяемого пользователем, под названием `World`. Вы можете импортировать его следующим образом:

```
from World import World
```

Такой способ использования команды `import` импортирует класс `World` из модуля `World`. Следующий код создает объект типа `World` и вызывает метод `mainloop`, который ожидает действий пользователя:

```
world = World()
world.mainloop()
```

Должно появиться окно с заголовком и пустым квадратом. Мы воспользуемся этим окном для рисования точек `Point`, прямоугольников `Rectangle` и других фигур. Вставьте следующие строки перед вызовом `mainloop` и запустите программу снова:

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

Должен появиться зеленый прямоугольник с черным очертанием. Первая строка создает объект `Canvas`, который появляется в окне в виде белого квадрата. Объект `Canvas` предоставляет методы, типа `rectangle`, для рисования различных фигур.

`bbox` это список списков, который представляет из себя "ограничивающие линии" прямоугольника. Первая пара координат обозначает левый нижний угол; вторая пара обозначает правый верхний.

Вы можете нарисовать круг следующим образом:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

Первый параметр это координаты центра круга; второй параметр это его радиус; третий параметр указывает, должно ли у круга быть очертание окружности (значение по умолчанию – `None`); четвертый параметр указывает на цвет круга.

Если вы добавите эту строчку к программе, то в результате у вас должен нарисоваться национальный флаг государства Бангладеш (см. `wikipedia.org/wiki/Gallery_of_sovereign-state_flags`).

1. Напишите функцию `draw_rectangle`, которая принимает в качестве аргументов объекты типа `Canvas` и `Rectangle` и рисует прямоугольник `Rectangle` на канве `Canvas`.
2. Добавьте атрибут `color` (цвет) к объекту `Rectangle` и измените `draw_rectangle` таким образом, чтобы она использовала этот атрибут в качестве заполняющего цвета.
3. Напишите функцию `draw_point`, которая принимает объекты `Canvas` и `Rectangle` в качестве аргументов и рисует точку `Point` на канве `Canvas`.
4. Определите новый класс `Circle` с соответствующими атрибутами и создайте несколько объектов типа `Circle`. Напишите функцию `draw_circle`, которая рисует круги на канве.
5. Напишите программу, которая рисует национальный флаг Чешской Республики. Подсказка: вы можете нарисовать многоугольник следующим образом:
```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

Я написал маленькую программу, которая выводит все доступные цвета. Вы можете скачать ее отсюда: `thinkpython.com/code/color_list.py`

Глава 16
Классы и функции

16.1 Time

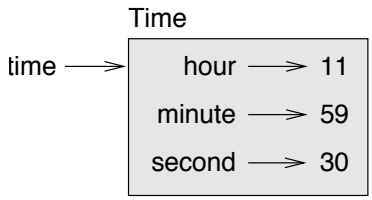
В качестве другого примера типа данных, определяемых пользователем, мы определим класс `Time`, который будет отображать время дня. Определение класса выглядит следующим образом:

```
class Time(object):
    """represents the time of day.
        attributes: hour, minute, second"""
```

Мы можем создать новый объект типа `Time` и присвоить ему атрибуты для часов, минут и секунд:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

Диаграмма состояния объекта `Time` выглядит следующим образом:



Упражнение 16.1 Напишите функцию `print_time`, которая принимает объект типа `Time` и выводит его данные в формате `hour:minute:second` (часы:минуты:секунды). Подсказка: форматирующая последовательность '`%.2d`' выводит целое число, используя, как минимум, две цифры, включая незначащий ноль вначале, если это необходимо.

Упражнение 16.2 Напишите булеву функцию `is_after`, которая берет два объекта типа `Time`, `t1` и `t2`, и возвращает `True`, если `t1` следует за `t2` хронологически, или `False` в противном случае. Сможете ли вы обойтись без выражения `if`?

16.2 Чистые функции

В нескольких следующих разделах мы напишем функции, которые будут прибавлять время. Они демонстрируют два вида функций: чистые функции и модификаторы. Они также демонстрируют образец разработки программ, который я называю **прототип и исправление** (prototype and patch), что является одним из способов решения сложной задачи: вы начинаете с простого прототипа и постепенно наращиваете функциональность.

Вот простой прототип функции `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second
    return sum
```

Функция создает новый объект типа `Time`, инициализирует его атрибуты и возвращает ссылку на новый объект. Это называется **чистой функцией** (pure function), потому что она не модифицирует ни один из объектов, переданных ей в качестве аргументов, а также потому, что она не производит

побочных действий, как, например, вывод результата вычислений, а только лишь *возвращает* некоторое значение.

Для проверки этой функции мы создадим два объекта типа `Time`: `start`, содержащий время начала фильма, такого как *Monty Python and the Holy Grail*, и `duration`, содержащий длительность фильма, которая равняется 1 час 35 минут. Функция `add_time` должна выяснить время, когда фильм заканчивается:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

Полученный результат `10:80:00` может быть не совсем то, что вы ожидали. Проблема заключается в том, что эта функция не обрабатывает корректно случаи, когда количество секунд или минут при сложении оказывается больше 60. Когда такое случается, мы должны "переносить" лишние секунды в минуты или лишние минуты в часы.

Вот исправленная версия:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Хотя эта функция и правильная, она становится уже слишком большой. Позднее мы познакомимся с ее альтернативной версией.

16.3 Модификаторы

Иногда от функции требуется модифицировать объекты, которые она получает в качестве параметров. В таком случае изменения становятся видимы и в том месте программы, где произошел вызов этой функции. Функции, работающие подобным образом, называются **модификаторами** (modifier).

Функцию `increment`, которая добавляет данное количество секунд к объекту типа `Time`, можно естественным образом написать в виде модификатора. Вот грубый набросок:

```
def increment(time, seconds):
    time.second += seconds
    if time.second >= 60:
        time.second -= 60
        time.minute += 1
    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Первая строчка совершает базовую операцию. Остальные имеют дело с особым случаем, как мы уже видели прежде.

Правильно ли работает эта функция? Что произойдет, если параметр `seconds` будет значительно превышать 60?

В подобном случае недостаточно будет перенести секунды один раз. Мы должны повторять это до тех пор, пока `time.second` не станет меньше 60. Одним из решений данной проблемы является замена `if` на `while`. При этом функция будет обрабатывать время правильно, но не слишком эффективно.

Упражнение 16.3 Напишите корректную версию функции `increment`, которая не содержит циклов.

Все, что можно сделать при помощи модификаторов, также можно сделать и при помощи чистых функций. В действительности существуют языки программирования, в которых допускается только использование чистых функций. Имеются некоторые свидетельства в пользу того, что программы, в которых используются чистые функции, требуют меньше времени на разработку и меньше подвержены ошибкам, чем программы, использующие модификаторы. Но использование модификаторов, порой, удобнее, а программы, использующие чистые функции, могут оказаться менее эффективными.

В общем случае я рекомендую использовать чистые функции там, где это возможно, а к модификаторам прибегать в случае очевидных преимуществ. Такой подход можно назвать **стилем функционального программирования** (functional programming style).

Упражнение 16.4 Напишите "чистую" версию функции `increment`, которая вместо модификации своего параметра создает и возвращает новый объект типа `Time`.

16.4 Что лучше: прототип или планирование?

Тот образец разработки, который я продемонстрировал, называется "прототип и исправление". Для каждой функции я написал ее прототип, который выполнял базовые вычисления, а когда я их тестировал, то исправлял ошибки по ходу их выявления.

Такой подход может быть достаточно эффективен, особенно если вы еще не поняли всю суть стоящей перед вами задачи. При таком последовательном исправлении ошибок может получится неоправданно усложненный код, т.к. он учитывает различные обстоятельства, а также не очень надежный, т.к. вы не можете быть уверены в том, что нашли и исправили все ошибки.

Альтернативой этому служит **разработка по плану** (planned development), при которой способность увидеть саму суть проблемы может значительно облегчить написание программы. В данном случае сутью является то, что объект типа `Time` представляет из себя трехзначное число по основанию 60 (см. `wikipedia.org/wiki/Sexagesimal`). Атрибут `seconds` представляет из себя "единицы", атрибут `minute` – "шестидесятки", а атрибут `hour` – "36-сотки".

Когда мы писали функции `add_time` и `increment`, мы, фактически, осуществляли сложение по основанию 60, именно по этому нам было необходимо переносить разряды из одной колонки в другую.

Данное наблюдение подталкивает нас к другому подходу для решения проблемы – мы можем конвертировать объект `Time` в целое число и воспользоваться тем преимуществом, что компьютер знает сам, как выполнять целочисленную арифметику.

Вот функция, которая конвертирует объект типа `Time` в целое число (т.е. количество секунд):

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

А вот функция, которая конвертирует целые числа в объект типа `Time` (напомним, что оператор `divmod` делит первый аргумент на второй и возвращает частное и остаток в виде кортежа):

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

Возможно, вам необходимо немного подумать, выполнить несколько тестов, чтобы убедиться, что эта функция работает правильно. Одним из способов убедиться в этом является проверка выражения `time_to_int(int_to_time(x)) == x` на многих значениях `x`. Это один из примеров проверки на последовательность.

Как только вы убедитесь в правильности, вы можете использовать эти функции для того, чтобы переписать заново `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Эта версия короче, чем первоначальная, и ее легче проверить.

Упражнение 16.5 Перепишите функцию `increment`, используя `time_to_int` и `int_to_time`.

В некотором роде конвертирование из основания 60 в основание 10 и обратно сложнее, чем просто обработка времени напрямую. Базовые конвертации более абстрактны; наша интуиция, которой мы пользуемся при подсчете времени, лучше.

Но если мы вникнем в суть проблемы использования чисел по основанию 60 и потратим некоторое время на разработку конвертирующих функций (`time_to_int` и `int_to_time`), у нас получится программа, которая будет короче, проще в чтении и отладке, а также более надежная.

Также позже будет проще добавлять новую функциональность в такую программу. Представьте, для примера, вычитание значений двух объектов типа `Time` для нахождения разницы длительности между двумя событиями. Было бы слишком наивно пытаться имитировать вычитание столбиком с заимствованием. Использование конвертирующих функций значительно облегчило бы эту задачу и сделало бы значительно менее вероятным наличие ошибок.

Ирония заключается в том, что в некоторых случаях усложнение проблемы (или, лучше сказать, представление ее в более общем виде), делает ее решение проще (потому что при этом бывает меньше различных частных случаев и, соответственно, меньше возможностей допустить ошибку).

16.5 Отладка

Объект типа `Time` хорошо согласован тогда, когда значения `minute` и `second` лежат между 0 и 60 (включая 0 и не включая 60), а также если значение `hour` больше нуля. `hour` и `minute` должны быть целыми числами, но `second` может принимать и дробное значение.

Подобные требования называются **инвариантами** (invariants), т.е. *неизменяемые*, потому что они всегда должны выполняться. Иными словами, если эти требования не выполняются, то что-то не так.

Написание кода для проверки инвариантов может помочь обнаружить ошибки и то, что их причинило. Например, у вас может быть функция типа `valid_time`, которая берет объект `Time` в качестве аргумента и возвращает `False`, если при этом нарушаются инварианты:

```
def valid_time(time):
    if time.hours < 0 or time.minutes < 0 or time.seconds < 0:
        return False
    if time.minutes >= 60 or time.seconds >= 60:
        return False
    return True
```

В начале каждой функции вы можете проверять ее аргументы, чтобы убедиться, что они корректны:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError, 'invalid Time object in add_time'
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Также вы можете использовать команду `assert`, которая проверяет данные инварианты и генерирует сообщение об ошибке, если они нарушаются:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Команда `assert` очень полезна, потому что она позволяет отделить нормальный код от кода, проверяющего ошибки.

16.6 Словарь терминов

прототип и исправление (prototype and patch): вид разработки программ, при котором вначале создается грубый набросок программы, затем она тестируется, а ошибки исправляются по мере их выявления.

разработка по плану (planned development): вид разработки программ, состоящий в том, что еще до разработки выявляется суть проблемы, затем заранее планируются шаги для ее разрешения.

чистая функция (pure function): функция, которая не модифицирует объекты, переданные ей в качестве аргументов; большинство чистых функций возвращают некоторый результат.

модификаторы (modifiers): функция, которая изменяет один и более из объектов, переданных ей в качестве аргументов; большинство модификаторов не возвращают никакого результата.

стиль функционального программирования (functional programming style): стиль разработки программ, при котором большинство функций являются чистыми.

инвариация (invariant): условие, которое всегда должно оставаться истинным во время работы программы.

16.7 Упражнения

Упражнение 16.6 Напишите функцию `mul_time`, которая принимает в качестве аргументов объект типа `Time`, а также некоторое число, и возвращает новый объект типа `Time`, являющийся произведением оригинального объекта и того числа.

Затем используйте `mul_time` для написания функции, которая принимает объект типа `Time`, который представляет из себя финишное время в гонке, и некоторое число, которое представляет из себя дистанцию. Функция должна возвращать объект типа `Time`, представляющий из себя среднюю скорость (время на один километр).

Упражнение 16.7 Создайте класс `Date` с атрибутами `day`, `month` и `year` (день, месяц и год, соответственно). Напишите функцию `increment_date`, которая принимает объект типа `Date` в качестве аргумента, `date` (дату) как целое число и `n`, а затем возвращает новый объект типа `Date`, который представляет из себя день, следующий через `n` дней после `date`. Трудность: обрабатывает ли ваша функция корректно високосные года? См. `wikipedia.org/wiki/Leap_ year`.

Упражнение 16.8 Модуль `datetime` предоставляет объекты `date` и `time`, похожие на объекты `Date` и `Time` из этой главы, но имеющие богатый набор методов и операторов. Прочитайте документацию `docs.python.org/lib/datetime-date.html`.

1. Используйте модуль `datetime` для написания программы, которая получает текущую дату и выводит день недели.
2. Напишите программу, которая принимает на входе любой день рождения и выводит возраст пользователя, а также число дней, часов, минут и секунд до его следующего дня рождения.

Глава 17
Классы и методы

17.1 Отличительные черты объектно-ориентированного программирования

Python является **объектно-ориентированным языком программирования** (object-oriented programming language), что означает, что в нем имеются средства, поддерживающие объектно-ориентированное программирование (ООП).

Нелегко дать четкое определение тому, что такое ООП, но мы уже встречались с некоторыми его характеристиками:

- Программы состоят из определений объектов и определений функций, а большинство вычислений выражается в виде операций над объектами.
- Каждое определение объекта соответствует некоторому объекту или концепции в реальном мире, а функции, которые осуществляют операции с этим объектом, соответствуют тому, как действуют объекты в реальном мире.

Например, класс `Time`, определенный в главе 16, соответствует тому, как люди записывают время дня, а функции, которые мы там определили, соответствуют тому, как люди обращаются с этим временем. Подобным же образом и классы `Point` и `Rectangle` соответствуют математическим концепциям точки и прямоугольника.

До сих пор мы не воспользовались теми преимуществами, которыми обладает Python для поддержки средств ООП. Строго говоря, эти средства не являются обязательными. Большинство из них лишь предоставляют альтернативный синтаксис для осуществления того, что мы уже делали и без них. Но во многих случаях эта альтернатива является более четким и кратким способом организовать структуру программы.

Например, в программе `Time` нет очевидной связи между определением класса и последующими определениями функций. При внимательном рассмотрении становится очевидным, что каждая функция принимает, как минимум, один объект типа `Time` в качестве аргумента.

Подобное наблюдение наводит нас на мысль о создании **методов** (method). Метод это функция, которая ассоциируется с определенным классом. Мы уже встречались с методами для строк, списков, словарей и кортежей. В этой главе мы создадим методы для типов, определяемых пользователем.

Семантически (т.е. по значению) методы не отличаются от функций, но есть две синтаксические разницы:

- Методы создаются внутри определения класса, для того, чтобы связь между классом и методом была явно выражена.
- Синтаксис при использовании метода отличается от синтаксиса при использовании функции.

В следующих нескольких разделах мы возьмем функции из предыдущих двух глав и преобразуем их в методы. Такое преобразование будет чисто механическим процессом. Вы можете осуществлять его, следуя простым инструкциям. Если вы не испытываете затруднений при конвертации одной формы в другую, то вы сможете выбирать ту форму, которая больше подходит для программы, которую вы пишите.

17.2 Печать объектов

В главе 16 мы определили класс `Time`, а в упражнении 16.1 вы написали функцию под названием `print_time`:

```
class Time(object):
    """represents the time of day.
       attributes: hour, minute, second
    """

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute,time.second))
```

Для вызова этой функции вам необходимо передать ей объект типа `Time` в качестве аргумента:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Для создания метода `print_time` все, что нужно сделать это поместить определение функции внутри определения класса. Обратите внимание на отступы:

```
class Time(object):
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Теперь у нас есть два способа вызова `print_time`. Первым (и менее распространенным) способом является использование синтаксиса вызова функции:

```
>>> Time.print_time(start)
09:45:00
```

При таком способе использования точечной записи `Time` является названием класса, а `print_time` – названием метода, которому `start` передается в качестве параметра.

Вторым (и более кратким) способом является способ вызова метода:

```
>>> start.print_time()
09:45:00
```

При таком использовании точечной записи `print_time` снова является названием метода, а `start` является объектом, на который вызывается этот метод, и называется **субъектом** (subject). Как и субъект (т.е. подлежащее) в обычном предложении говорит о том, о чем это предложение, так и субъект вызова метода говорит о назначении метода.

Внутри метода субъект присваивается первому параметру, так что в данном случае `start` присваивается параметру `time`.

По соглашению, первый параметр метода называется `self` (в английском языке это возвратное местоимение, означающее *я сам*), поэтому в более привычном виде `print_time` можно записать следующим образом:

```
class Time(object):
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

Причиной для подобного соглашения служит подразумеваемая метафора:

- Синтаксис для вызова функции `print_time(start)` подразумевает, что данная функция является активным агентом. Здесь как бы говорится: "Эй, `print_time`! Вот тебе объект для печати."
- В ООП объекты являются активными агентами. Вызов метода, подобного `start.print_time()` как бы говорит: "Эй, `start`! Ну-ка, распечатай сам себя."

Переход от вызова функций к вызовам методов может показаться довольно интересным делом, но польза подобного перехода не вполне очевидна. В тех примерах, которые мы видели до сих пор, польза не очевидна. Но иногда такой переход позволяет писать функции более широкого применения, а также облегчается поддержка и повторное использование кода.

Упражнение 17.1 Преобразуйте функцию `time_to_int` (из раздела 16.4) в метод. Возможно, что обратную функцию `int_to_time` не стоит преобразовывать в метод, т.к. не вполне очевидно, на какой объект он должен вызваться!

17.3 Другой пример

Вот версия `increment` (из раздела 16.3), преобразованная в метод:

```
# внутри класса Time:
    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

В данной версии подразумевается, что функция `time_to_int` была преобразована в метод, как в упражнении 17.1. Также обратите внимание на то, что это чистая функция, а не модификатор.

Вот как можно использовать метод `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

Субъект `start` присваивается первому параметру, т.е. `self`. Аргумент, равный `1337`, присваивается второму параметру, т.е. `seconds`.

Такой механизм присваивания параметров может показаться запутанным, особенно если вы допускаете ошибку. Например, если вы вызовете `increment` с двумя аргументами, вы получите следующее:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

(`Ошибка типизации: increment() принимает ровно 2 аргумента (было передано 3)`).

Такое сообщение может поначалу сбить с толку, т.к. в скобках было только два аргумента. Но субъект тоже засчитывается за аргумент, поэтому вместе их три.

17.4 Более сложный пример

Более сложным примером является функция `is_after` (из упражнения 16.2), потому что она принимает два объекта типа `Time` в качестве параметров. В данном случае имеется соглашение называть первый параметр `self` (я сам), а второй – `other` (другой):

```
# внутри класса Time:
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Для использования данного метода необходимо вызвать его на один объект, а другой передать в качестве аргумента:

```
>>> end.is_after(start)
True
```

Подобный синтаксис читается подобно тому, как если бы это было обычное предложение на английском языке: "end is after start?"

17.5 Метод `init`

Метод `init` (сокращение от initialisation – инициализация) является специальным методом, который вызывается тогда, когда объект создается в качестве экземпляра класса. Его полное имя `__init__` (два

символа подчеркивания, затем init и снова два символа подчеркивания). Метод init класса Time может выглядеть следующим образом:

```
# inside class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Очень часто параметры метода __init__ имеют те же имена, что и атрибуты. Выражение

```
self.hour = hour
```

записывает значение параметра hour в качестве атрибута self.

В данном случае параметры опциональны. Поэтому, если вы вызовете Time без аргументов, то вы получите их значения по умолчанию:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

Если вы предоставите один аргумент, то он заместит собой то значение, которое hour имеет по умолчанию:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

Если вы предоставите два аргумента, то они заместят собой hour и minute:

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

Наконец, три предоставленных аргумента заместят собой все три значения по умолчанию.

Упражнение 17.2 Напишите метод init для класса Point, который принимает x и y в качестве дополнительных параметров и присваивает им соответствующие атрибуты.

17.6 Метод str

Помимо __init__ в Python есть другой особый метод, называемый __str__, который предназначен для возвращения строки, являющейся представлением объекта.

Например, вот метод str объекта Time:

```
# внутри класса Time:
    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Если вы передадите этот объект функции print, то Python вызовет метод str этого объекта:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

Когда я пишу новый класс, то почти всегда я начинаю с написания методов __init__, что облегчает инициализацию объекта, и __str__, что помогает при отладке.

Упражнение 17.3 Напишите метод str для класса Point. Создайте любой объект типа Point и распечатайте его (примените к нему функцию print).

17.7 Перегрузка операторов

Определяя другие специальные методы вы можете установить поведение операторов для типов, определяемых пользователем. Например, если вы определяете метод `__add__` (что значит *добавить*) для класса `Time`, вы можете использовать оператор + для объектов типа `Time`.

Вот как может выглядеть подобное определение:

```
# inside class Time:
    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

А вот так вы можете его использовать:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

Когда вы применяете оператор + к объектам типа `Time`, Python вызывает метод `__add__`. Когда вы печатаете результат, то Python вызывает метод `__str__`. Так что много чего происходит за кадром!

Изменение поведения оператора таким образом, чтобы он работал с типами, определяемыми пользователем, называется **перегрузкой операторов** (operator overloading). Для каждого оператора в Python имеется соответствующий особый метод, подобный `__add__`. Вы можете посмотреть по ним документацию `docs.python.org/ref/specialnames.html`.

Упражнение 17.4 Напишите метод `add` для класса `Point`.

17.8 Диспетчеризация в зависимости от типа

В предыдущем разделе мы осуществляли сложение двух объектов типа `Time`, но, возможно, вы захотите складывать объект типа `Time` и целое число. Вот версия `__add__`, которая проверяет тип аргумента `other` и вызывает соответствующий ему метод: либо `add_time`, либо `increment`:

```
# внутри класса Time:
    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
            return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Встроенная функция `isinstance` принимает в качестве аргументов некоторое значение и объект типа `class`, и возвращает `True`, если это значение является экземпляром данного класса.

Если `other` является объектом типа `Time`, то `__add__` вызывает `add_time`. В противном случае считается, что параметр является обычным числом, и вызывается `increment`. Такая операция называется **диспетчеризация в зависимости от типа** (type based dispatch), потому что она вызывает разные методы в зависимости от типа аргументов.

В следующем примере оператор + работает с различными типами:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

К сожалению, сложение тут производится таким образом, что слагаемые нельзя переставлять. Если целое число будет первым операндом, то вы получите сообщение об ошибке:

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Проблема заключается в том, что вместо того, чтобы к объекту типа `Time` прибавить целое число, Python пытается целое число прибавить к объекту типа `Time`, и не знает, как это делать. Но эта проблема разрешается просто – путем добавления метода `__radd__`, что означает "right-side add" (сложение с правой стороны). Этот метод вызывается тогда, когда объект типа `Time` появляется с правой стороны оператора +. Вот его определение:

```
# внутри класса Time:
    def __radd__(self, other):
        return self.__add__(other)
```

А вот как он используется:

```
>>> print(1337 + start)
10:07:17
```

Упражнение 17.5 Напишите метод `add` для класса `Point`, который работает либо с объектом типа `Point`, либо с кортежем:

- Если второй операнд принадлежит к классу `Point`, то метод должен возвращать новый объект типа `Point`, у которого координата `x` является суммой координат `x` операндов; то же самое верно и в отношении координат `y`.
- Если второй операнд является кортежем, то метод должен прибавлять первый элемент кортежа к координате `x`, а второй элемент к координате `y`. В качестве результата должен возвращаться новый объект типа `Point`.

17.9 Полиморфизм

Диспетчеризация в зависимости от типа полезна тогда, когда она необходима, но, к счастью, она не всегда нужна. Часто вы можете избежать ее использования. Для этого нужно писать функции, которые корректно работают с аргументами разных типов.

Многие функции, которые мы написали для работы с функциями, будут также работать и с другими последовательностями. Например, в разделе 11.1 мы пользовались функцией `histogram` для подсчета того, сколько раз каждая буква встречается в слове.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
    else:
        d[c] = d[c]+1
    return d
```

Эта функция также работает со списками, кортежами и даже со словарями до тех пор, пока элементы последовательности `s` относятся к хешируемому типу и, следовательно, могут использоваться в качестве ключей словаря `d`.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Функции, которые могут работать с несколькими типами данных, называются **полиморфными** (polymorphic). Полиморфизм помогает повторно использовать код. Например, встроенная функция `sum`, которая производит сложение элементов в последовательности, будет работать до тех пор, пока элементы последовательности поддерживают операцию сложения.

Поскольку мы написали метод `add` для нашего класса `Time`, объекты типа `Time` будут работать с функцией `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

В общем случае, если все операции внутри функции могут работать с данным типом данным, то и сама функция может работать с этим типом.

Лучший вид полиморфизма это непреднамеренный полиморфизм, когда вы обнаруживаете, что функция, которую вы написали, может работать с типам данных, для которых вы ее не планировали.

17.10 Отладка

Допускается добавление атрибутов к объектам в любое время выполнения программы, но если вы приверженец теории типов, то иметь объекты одного типа с разными наборами атрибутов было бы сомнительной практикой. Гораздо лучше инициализировать все атрибуты объектов в методе `init`.

Если вы не уверены в том, имеется ли у данного объекта определенный атрибут, вы можете воспользоваться встроенной функцией `hasattr` (см. раздел 15.7).

Получить доступ к атрибутам объекта можно также и используя специальный атрибут `__dict__`, представляющий из себя словарь, который отображает имена атрибутов (как строки) на их значения:

```
>>> p = Point(3, 4)
>>> print(p.__dict__)
{'y': 4, 'x': 3}
```

В целях отладки всегда полезно держать эту функцию под рукой:

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print(attr, getattr(obj, attr))
```

`print_attributes` перебирает элементы в словаре объекта и выводит имя каждого атрибута и его соответствующее значение.

Встроенная функция `getattr` принимает объект и имя атрибута (как строку) в качестве параметров и возвращает значение атрибута.

17.11 Словарь терминов

объектно-ориентированный язык (object-oriented language): язык, в котором имеются такие средства, как определяемые пользователем классы и синтаксис использования методов, которые способствуют объектно-ориентированному программированию.

объектно-ориентированное программирование – ООП (object-oriented programming – OOP): стиль программирования, при котором данные и операции над ними организованы в классы и методы.

метод (method): функция, которая определена внутри класса и вызывается на экземпляр этого класса.

субъект (subject): объект, на который вызывается метод.

перегрузка операторов (operator overloading): изменение поведения оператора, такого как +, так, что он работает с типом данных, определяемых пользователем.

диспетчеризация в зависимости от типа (type-bases dispatch): шаблонный метод программирования, который проверяет тип операнда и вызывает разные функции для разных типов данных.

полиморфный (polymorphic): относится к функции, которая может работать с более чем одним типом данных.

17.12 Упражнения

Упражнение 17.6 Это упражнение является своего рода предупреждением об одной очень часто встречаемой и трудно обнаруживаемой ошибке при программировании в Python.

1. Создайте класс `Kangaroo` (*кенгуру*) со следующими методами:

    a. Метод `__init__`, который инициализирует атрибут под названием `pouch_contents` (*содержимое сумки*) пустым списком.

    b. Метод `put_in_pouch` (*положить в сумку*), который берет объект любого типа и добавляет его к `pouch_contents`.

    c. Метод `__str__`, который возвращает строковое представление объекта типа `Kangaroo` и содержимое сумки.

    Проверьте ваш код, создав два объекта типа `Kangaroo`, присвоив их переменным `kanga` и `roo`, а затем добавив `roo` к содержимому сумки `kanga`.

2. Скачайте файл `thinkpython.com/code/BadKangaroo.py`. Он содержит решение предыдущей задачи, но с одной ошибкой, трудноуловимой ошибкой. Найдите этот баг и исправьте его.

    Если вы не сможете справиться, то можете скачать файл `thinkpython.com/code/GoodKangaroo.py`, который содержит объяснение проблемы и демонстрацию решения.

Упражнение 17.7 В Python есть модуль `visual`, предназначенный для работы с 3-D графикой. Он не всегда входит в установку Python, поэтому вам может понадобиться установить его дополнительно из репозитариев или из сайта `vpython.org`.

В следующем примере создается 3-D пространство с размерами 256 единиц в ширину, длину и высоту, а также устанавливается "центр", представляющий из себя точку (128, 128, 128). Затем рисуется голубая сфера.

```
from visual import *
scene.range = (256, 256, 256)
scene.center = (128, 128, 128)
color = (0.1, 0.1, 0.9) # mostly blue
sphere(pos=scene.center, radius=128, color=color)
```

Здесь `color` представляет из себя RGB кортеж, т.е. его элементы это уровни красного (R), зеленого (G) и синего (B) цвета в диапазоне от 0.0 до 1.0 (см. `wikipedia.org/wiki/RGB_color_model`).

Если вы запустите этот код, то вы должны увидеть окно с черным фоном и голубую сферу. Если вы нажмете среднюю кнопку мыши и будет ею двигать, то это вызовет изменение масштаба. Вы также можете поворачивать сцену, двигая мышь при нажатой правой кнопке, но имея лишь одну сферу, трудно увидеть разницу.

Следующий код создает куб из сфер:

```
t = range(0, 256, 51)
    for x in t:
        for y in t:
            for z in t:
                pos = x, y, z
                sphere(pos=pos, radius=10, color=color)
```

1. Поместите этот код в скрипт и убедитесь, что он у вас работает.

2. Измените программу так, чтобы каждая сфера в кубе имела цвет, соответствующий ее положению в RGB пространстве. Обратите внимание на то, что координаты находятся в пределах 0-255, тогда как элементы RGB кортежа занимают диапазон 0.0 – 1.0.
3. Скачайте `thinkpython.com/code/color_list.py` и используйте функцию `read_colors` (читать цвета) для создания списка цветов, доступных в вашей системе, их имена и RGB значения. Для каждого цвета, который имеет имя, нарисуйте сферу в позиции, соответствующей ее RGB значению.

Вы можете посмотреть мое решение здесь: `thinkpython.com/code/color_space.py`.

Глава 18
Наследование

В этой главе мы разработаем классы, которые будут представлять собой игральные карты, колоды карт и игроков в покер. Если вы не умеете играть в покер, то вы можете прочитать о нем в википедии wikipedia.org/wiki/Poker, хотя это и не обязательно. Я скажу вам все, что вам необходимо будет знать, когда вы будете выполнять упражнения.

Если вы не знакомы с англо-американскими картами, то вы тоже можете прочитать о них в википедии: wikipedia.org/wiki/Playing_cards

18.1 Карта как объект

Колода состоит из 52 карт, каждая из которых принадлежит к одной из четырех мастей (suit) и одному их тринадцати рангов (rank). Вот карточные масти в порядке убывания при игре в бридж: пики (spades), черви (hearts), бубны (diamonds) и трефы (clubs). По рангу карты располагаются следующим образом: туз (ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, валет (jack), дама (queen) и король (king). В зависимости от вида игры, туз может быть рангом выше короля или ниже 2.

Если мы хотим определить новый объект, который будет представлять собой игральную карту, то вполне очевидно, что у него должны быть атрибуты suit и rank. Но не вполне очевидно, каков должен быть тип этих атрибутов. Можно, для примера, было бы использовать строки со словами типа "Spade" для мастей и "Queen" для рангов. Но при таком подходе будет проблемно сравнивать между собой карты на основании их рангов и мастей.

В качестве альтернативы можно использовать целые числа для того, чтобы **закодировать** (encode) ими ранги и масти. В данном контексте слово "закодировать" означает, что мы собираемся установить некоторое соответствие между числами и мастями или между числами и рангами. Такое кодирование не подразумевает никаких секретов (в противном случае это назвалось бы *шифрование*).

Например, следующая таблица показывает масти и соответствующие им числовые коды:

```
Spades      |→    3
Hearts      |→    2
Diamonds    |→    1
Clubs       |→    0
```

При такой кодировке карты легко сравнивать между собой, т.к. более старшая масть имеет больший числовой код.

Способ задания отображения рангов карт кажется вполне очевидным: картам с числами и тузу соответствуют из числовые значения, а для остальных карт (с людьми) можно ввести дополнительные коды:

```
Jack  |→    11
Queen |→    12
King  |→    13
```

Я использую символ |→ для того, чтобы подчеркнуть, что такое отображение не является частью программы Python. Это лишь часть программного дизайна, но это не отобразится явным образом в коде.

Определение класса Card будет выглядеть следующим образом:

```
class Card(object):
    """представляет стандартную игральную карту."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Как и обычно, метод init принимает опциональный параметр для каждого атрибута. Картой по умолчанию является двойка треф.

Для создания карты (т.е. объекта типа `Card`) вы должны вызвать `Card` с желаемыми мастью и рангом в качестве аргументов:

```
queen_of_diamonds = Card(1, 12)
```

18.2 Атрибуты класса

Для того, чтобы распечатать объект типа `Card` в удобном для человека формате, нам необходимо отображение числовых кодов на соответствующие им ранги и масти. Это довольно просто сделать при помощи списков, элементами которых являются строки. Мы присваиваем эти списки к **атрибутам класса** (class attribute):

```
# inside class Card:
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']
    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Переменные, подобные `suit_names` (*имена мастей*) и `rank_names` (*имена рангов*), которые определены внутри класса, но вне любого метода, называются атрибутами класса, т.к. они связаны с объектами из класса `Card`.

Этот термин делает различие между ними и переменными типа `suit` и `rank`, которые называются **атрибутами экземпляра класса** (instance attributes), т.к. они связаны с определенным экземпляром.

Доступ к обоим видам атрибутов осуществляется при помощи точечной записи. Например, в записи `__str__` `self` является объектом типа `Card`, а `self.rank` представляет его ранг. Подобным же образом, `Card` представляет собой объект типа `class`, а `Card_rank_names` это список строк, связанных с этим классом.

Каждая карта имеет свои собственные масть и ранг, но существует только одна копия `suit_names` и `rank_names`.

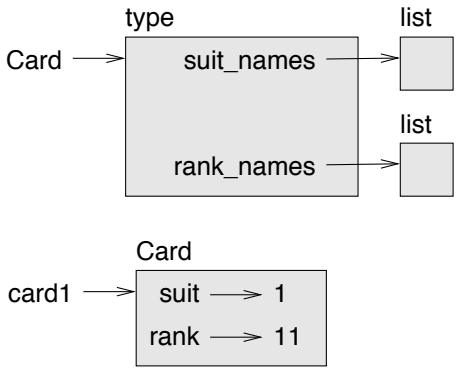
Объединяя все это вместе, выражение `Card.rank_names[self.rank]` означает: "используй атрибут `rank` объекта `self`, который является индексом списка `rank_names` из класса `Card` и выбери соответствующую ему строку."

Первым элементом списка `rank_names` является `None`, т.к. не существует карты нулевого ранга. Используя значение `None` в нашем списке в качестве "заглушки", мы добиваемся того, что наше отображение выглядит естественным образом, т.е. индекс 2 соответствует строке '`2`' и т.д. В противном случае мы вынуждены были бы использовать словарь вместо списка.

С помощью методов, которые мы создали до сих пор, мы уже можем создавать и распечатывать карты:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Вот диаграмма, показывающая объект класса `Card` и один экземпляр типа `Card`:



Card является объектом типа class, поэтому он имеет тип type. card1 имеет тип Card (для экономии места я не отобразил содержимое suit_names и rank_names).

18.3 Сравнение карт

Для встроенных типов данных существуют операторы сравнения (<, >, == и т.д.), которые сравнивают, которые из них больше, меньше, или они равны друг другу. Для типов, определяемых пользователей, мы можем перегрузить поведение встроенных операторов при помощи метода __cmp__ (от слова compare – *сравнивать*).

Метод __cmp__ принимает два параметра, self и other, и возвращает положительное число, если первый объект больше, отрицательное число, если первый объект меньше, и 0, если они равны друг другу.

В отношении карт не вполне очевиден правильный порядок сравнения. Например, что больше: 3 треф или 2 бубей? У одной карты выше ранг, а у другой – масть. Для того, чтобы сравнивать карты, вы должны определиться с тем, что старше: ранг или масть.

Ответ может зависеть от того, в какую игру вы собираетесь играть, но для простоты мы сделаем произвольный выбор в пользу масти, что она старше ранга. Поэтому все пики будут старше бубен и т.д.

Определившись с этим, мы можем написать метод __cmp__:

```
# внутри класса Card:
    def __cmp__(self, other):
        # сравнение мастей
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # масти совпадают ... сравнение рангов
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # ранги совпадают ... ничья
        return 0
```

Более кратко это можно написать, используя сравнение кортежей:

```
# внутри класса Card:
    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

У встроенной функции cmp такой же интерфейс, что и у метода __cmp__: она принимает два значения и возвращает положительное число, если первое значение больше; отрицательное число, если первое значение меньше; 0, если они равны.

Упражнение 18.1 Создаете метод __cmp__ для класса Time. Подсказка: вы можете использовать сравнение кортежей, но можете также рассмотреть и использование целочисленного вычитания.

18.4 Колоды

Теперь, когда у нас имеются карты Card, нам необходимо определить колоды Deck. Т.к. колода состоит из карт, это будет довольно естественно, если каждая колода Deck будет содержать список карт в качестве атрибута.

Ниже дано определение класса Deck. Метод __init__ создает атрибут cards и генерирует стандартный набор и 52 карт:

```
class Deck(object):
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

Самым простым способом заполнения колоды является использование вложенного цикла. Внешний цикл нумерует масти от 0 до 3. Внутренний цикл нумерует ранги от 1 до 13. Каждый проход создает новый объект типа `Card` соответствующей масти и ранга и добавляет ее к `self.cards`.

18.5 Распечатывание колоды

Вот метод `__str__` для `Deck`:

```
# внутри класса Deck:
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Этот пример демонстрирует довольно эффективный метод построения длинной строки: создание списка из отдельных строк и, затем, соединение их воедино с помощью метода `join`. Встроенная функция `str` вызывает метод `__str__` на каждую карту и возвращает его строковое представление.

Поскольку мы вызываем метод `join` на символ новой строки, каждая карта занимает одну строку. Вот как выглядит результат:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Хотя результат и занимает 52 строки, но это, тем не менее, лишь одна строка, содержащая символы новой строки `\n`.

18.6 Добавление, удаление, перемешивание, сортировка

Для игры в карты нам может понадобиться метод, который удаляет карту из колоды и возвращает ее значение. Для этого удобно воспользоваться методом списков `pop`:

```
# внутри класса Deck:
    def pop_card(self):
        return self.cards.pop()
```

Т.к. метод `pop` удаляет *последнюю* карту в списке, мы вынуждены применять его к самому низу колоды. В реальной игре такие веще не одобряются[30], но в данном контексте все нормально.

Для добавления карты мы можем воспользоваться методом `append`:

---

[30] См. `wikipedia.org/wiki/Bottom_dealing`

```
# внутри класса Deck:
    def add_card(self, card):
        self.cards.append(card)
```

Метод, подобный этому, который использует другую функцию, а сам не делает, практически, ничего, иногда называется **декоратором** (veneer). Метафора происходит из строительного дела, где часто тонкий слой шпона наклеивают поверх дешевой древесины.

В данном случае мы определяем "тонкий" метод, выражающийся в операции со списком, которая подходит для работы с колодой карт.

В качестве другого примера мы можем написать для класса `Deck` метод `shuffle`, который использует функцию `shuffle` из модуля `random`:

```
# внутри класса Deck:
    def shuffle(self):
        random.shuffle(self.cards)
```

Не забудьте импортировать модуль `random`.

Упражнение 18.2 Напишите для класса `Deck` метод `sort`, который будет использовать метод `sort` для списков для сортировки карт в колоде. `sort` использует метод `__cmp__`, который мы определили, чтобы установить порядок сортировки.

18.7 Наследование

Языковым средством, которое чаще всего ассоциируется с ООП, является **наследование** (inheritance). Наследование это способность определить новый класс, который является измененной версией уже существующего класса.

Слово "наследование" используется потому, что новый класс наследует методы существующего класса. Расширяя метафору, существующий класс называется **родительским** (parent), а новый класс – **дочерним** (child) или **производным**.

В качестве примера, давайте скажем, что мы хотим создать класс, представляющий собой игрока (hand), т.е. набор карт, которые держит один игрок. *Игрок* похож на *Колоду*: оба состоят из набора карт и оба требуют некоторых операций, типа добавления и удаления карт.

Игрок также отличается от *Колоды*: есть операции, которые бы мы хотели применить к *Игроку*, но которые не имели бы большого смысла применять в *Колоде*. Например, в покере мы могли бы сравнить между собой два *Игрока*, чтобы увидеть, который из них побеждает. В бридже мы могли бы подсчитать очки у *Игрока*, чтобы знать, какую ставку делать.

Такие отношения между классами – они похожи, но, тем не менее, различаются, – приводит нас к идее наследования.

Определение дочернего класса похоже на определение любого другого класса, но в скобках должно указываться имя родительского класса:

```
class Hand(Deck):
    """represents a hand of playing cards"""
```

Данное определение показывает нам, что класс `Hand` наследуется от класса `Desk`. Это означает, что в объектах, созданных из `Hand`, мы можем пользоваться методами, которые имеются в классе `Deck`, например, `pop_card` или `add_card`.

Класс `Hand` также наследует от класса `Deck` метод `__init__`, но на самом деле он не делает того, что нам нужно: вместо того, чтобы выдавать игроку, представленному классом `Hand`, 52 новые карты, нужно, чтобы в `Hand` метод `__init__` инициализировал `cards` пустым списком.

Если в классе `Hand` мы создадим метод `__init__`, то он заместит собой тот, что имеется в родительском классе `Deck`:

```
# внутри класса Hand:
    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

Поэтому, когда вы создаете новый объект класса `Hand`, Python вызывает *его* метод `init`:

```
>>> hand = Hand('new hand')
>>> print(hand.cards)
[]
>>> print(hand.label)
new hand
```

Но другие методы наследуются из класса `Deck`, поэтому мы можем использовать `pop_card` и `add_card` для работы с картами:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

Следующий шаг – инкапсуляция этого кода в метод `move_cards` (*сделать ход*) – напрашивается сам собой:

```
#inside class Deck:
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

Метод `move_cards` принимает два аргумента – объект типа `Hand` и количество карт, которые будут использоваться для совершения хода. Этот метод модифицирует как `self`, так и `hand`, и возвращает `None`.

В некоторых играх при ходе карта переходит от одного игрока к другому, или от игрока обратно в колоду. Вы можете использовать `move_cards` для любой из этих операций: `self` может быть как `Deck`, так `Hand`, а `hand`, несмотря на свое название, может быть `Deck`.

Упражнение 18.3 Напишите метод для класса `Deck` под названием `deal_hands`, который берет два параметра, – число игроков и количество карт на одного игрока, – и создает новые объекты типа `Hand`, учитывая количество карт на одни руки, и возвращает список объектов `Hand`.

Наследование является полезным свойством. Некоторые программы, которые без использования наследования повторяли бы много кода, при использовании наследования удается сделать значительно короче. Наследование облегчает повторное использование уже созданного кода в других приложениях, поскольку вы можете изменить поведение родительского класса не меняя его самого. В некоторых случаях структура, созданная при помощи наследования, отражает естественную сущность проблемы, что облегчает понимание работы программы.

С другой стороны, наследование может усложнить чтение программы. Когда вызывается какой-либо метод, то не всегда ясно, где искать его определение. Код, относящийся к какой-либо конкретной ситуации, может быть разбросан по нескольким модулям. Также следует учесть, что многие вещи, которые можно сделать при помощи наследования, можно сделать и без него, иногда даже лучше.

18.8 Диаграммы класса

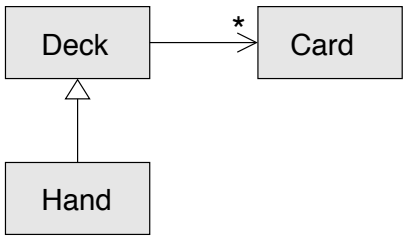
До сих пор мы видели лишь стековые диаграммы, показывающие состояние программы, и диаграммы объектов, показывающие атрибуты объектов и их значения. Эти диаграммы представляют из себя снимки состояния работающей программы, поэтому они изменяются в течение выполнения программы.

Также они отображают много деталей; в некоторых случаях даже слишком много. Диаграмма класса является более абстрактным представлением структуры программы. Вместо того, чтобы показывать индивидуальные объекты, она показывает классы и отношения между ними.

Есть несколько видов отношений между классами:

- Объекты в одном классе могут содержать ссылки на объекты в другом классе. Например, каждый прямоугольник `Rectangle` содержит ссылку на точку `Point`, а каждая колода карт `Deck` содержит ссылки на множество карт `Card`. Такой вид отношений называется **HAS-A** (СОДЕРЖИТ), как во фразе "a `Rectangle` has a `Point`" (класс `Rectangle` содержит класс `Point`).
- Один класс может наследоваться от другого. Такое отношение называется **IS-A** (ЯВЛЯЕТСЯ), как во фразе "a `Hand` is a kind of a `Deck`" (класс `Hand` является видом класса `Deck`).
- Один класс может зависеть от другого в том смысле, что изменения в одном классе потребуют изменений в другом.

Диаграмма класса (class diagram) является графическим представлением этих отношений.[31] Например, следующая диаграмма показывает отношения между классами `Card`, `Deck` и `Hand`:



Стрелка с полым треугольником представляет отношение IS-A; в данном случае оно показывает, что класс `Hand` наследуется от класса `Deck`.

Стандартная стрелка представляет отношение HAS-A; в данном случае оно показывает, что в классе `Deck` имеются ссылки на объекты `Card`.

Звездочка (`*`) возле стрелки обозначает **множественность** (multiplicity); она показывает, сколько объектов `Card` содержится в `Deck`. Множественность может быть простым числом, таким как 52; диапазоном, таким как 5...7; а также она может быть звездочкой, говорящей о том, что `Deck` может содержать любое количество `Card`.

Более детальная диаграмма может показать, что `Deck` на самом деле содержит *список* из объектов `Card`, но встроенные типы, такие как списки и словари, как правило, не включают в диаграммы классов.

Упражнение 18.4 Прочитайте `TurtleWorld.py`, `World.py` и `Gui.py` и нарисуйте диаграммы классов, показывающие отношения между классами, которые там используются.

18.9 Отладка

Если вы используете наследование, отладка программ может стать труднее, т.к. когда вы взываете метод на объект, вы можете не знать, какой именно метод будет вызван.

Предположим, вы пишите функцию, которая работает с объектами типа `Hand`, причем, хотите, чтобы она работала со всеми типами `Hand`, т.е. `PokerHand`, `BridgeHand` и т.п. Если вы, для примера, вызовете метод `shuffle` (*перемешать*), то может быть вызван тот, который определен в классе `Deck`. Но если любой из дочерних классов замещает этот метод, то он и будет вызван.

Каждый раз, когда вы не уверены, в каком порядке выполняются действия, проще всего добавить вызов функции `print` (или аналогичную команду) в начале метода, вызывающего сомнение. Если `Deck.shuffle` выводит сообщение, говорящее что-то вроде `Running Deck.shuffle`, то это означает, что запущенная программа отслеживает порядок выполнения действий.

В качестве альтернативы вы можете воспользоваться нижеследующей функцией, которая в качестве аргументов принимает любой объект и имя метода (как строку), а возвращает тот класс, в котором этот метод был определен:

---

[31] Диаграммы, которыя я здесь использую, похожи на UML (унифицированныя язык моделирования) с некоторыми упрощениями. См. `wikipedia.org/wiki/Unified_Modeling_Language`

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Вот пример ее работы:

```
>>> hand = Hand()
>>> print(find_defining_class(hand, 'shuffle'))
<class 'Card.Deck'>
```

Таким образом, метод `shuffle` данного объекта типа `Hand` определен в классе `Deck`.

В данной функции используется метод `mro`, который возвращает список объектов типа `class`. В этом списке уже ищется соответствующий метод. "MRO" означает "*method resolution order*".

Вот совет, которым вы можете воспользоваться при проектировании программ: всякий раз, когда вы замещаете метод, интерфейс нового метода должен быть таким же, как и у старого. Он должен принимать такие же параметры, возвращать такой же тип, у него должны быть такие же входные и выходные условия. Если вы будете соблюдать данное правило, то вы обнаружите, что любая функция, написанная для работы в родительском клас169се, таком как `Deck`, будет также работать и с дочерними классами, такими как `Hand` или `PokerHand`.

Если вы нарушите это правило, то ваша программа может рухнуть в одно прекрасное время, как карточный домик.

18.10 Словарь терминов

кодировать (encode): представлять один набор величин при помощи другого набора, устанавливая между ними соответствие.

атрибуты класса (class attribute): атрибуты, ассоциируемые с объектом типа `class`. Атрибуты класса определяются внутри класса, но вне любого из его методов.

атрибуты экземпляра класса (instance attribute): атрибуты, ассоциируемые с экземпляром класса.

декоратор (veneer): метод или функция, которые сами по себе, практически, ничего не делают, но предоставляют другой интерфейс для использовании другой функции.

наследование (inheritance): возможность определять новый класс, который является модифицированной версией ранее определенного.

родительский класс (parent class, superclass): класс, от которого происходит дочерний класс.

дочерний класс или **производный класс** (child class, subclass): класс, созданный на основе уже существующего класса путем наследования.

отношение ЯВЛЯЕТСЯ (IS-A relationship): отношение между дочерним и родительским классами.

отношение СОДЕРЖИТ (HAS-A relationship): отношение между двумя классами, при котором экземпляр одного класса содержит ссылки на экземпляр другого класса.

диаграмма класса (class diagram): диаграмма, показывающая классы в программе и их взаимоотношения.

множественность (multiplicity): запись в диаграмме класса, показывающая, сколько ссылок на экземпляры другого класса содержится в отношении HAS-A.

18.11 Упражнения

Упражнение 18.5 Ниже приведены возможные раскладки карт при игре в покер в возрастающем порядке их ценности (и в убывающем порядке возможности их выпадения):

пара (pair): две карты одного ранга

две пары (two pairs): две пары карт одного ранга

три одного ранга (three of a kind): три карты одного ранга

стрит (straight): пять карт одного ранга (не обязательно одной масти), идущих по порядку (туз может быть как самой младшей картой `Ace-2-3-4-5`, так и самой старшей `10-Jack-Queen-King-Ace`; но следующая комбинация не засчитывается: `Queen-King-Ace-2-3`)

флеш (flush): пять карт одной масти

фул-хаус (full house): три карты одного ранги и две другого

четыре одного ранга (four of a kind): четыре карты одинакового ранга

флеш-стрит (straight flush): комбинация флеш и стрит одновременно, т.е. пять карт одной масти, идущих по порядку

Целью следующих упражнений является подсчет вероятности выпадения указанных раскладов карт.

1. Скачайте следующие файлы с `thinkpython.com/code`:
   `Card.py` : полная версия классов `Card`, `Deck` и `Hand`, описанных в этой главе.
   `PokerHand.py` : не до конца разработанный класс, представляющий собой карты для игры в покер и некоторый код для его тестирования.
2. Если вы запустите файл `PokerHand.py`, он раздаст по 7 карт на 7 игроков и проверит, не выпал ли кому из них флеш. Изучите внимательно этот код перед тем, как идти дальше.
3. Добавьте к `PokerHand.py` методы `has_pair` (*имеется пара*), `has_twopair` (*имеются две пары*) и т.д., которые возвращают `True` или `False` в зависимости от того, имеется ли данный расклад в выпавших картах. Ваш код должен работать корректно для раздачи, содержащей любое количество карт (хотя чаще всего используются расклады на 5 и на 7 карт).
4. Напишите метод `classify` (классифицировать), который определяет расклад наивысшей ценности, выпавший игроку, и присваивает данное значение атрибуту `label`. Например, при раскладе на 7 карт одному игроку может выпасть пара и флеш. Флеш имеет большую ценность, чем пара, поэтому он и должен быть записан в атрибут `label`.
5. Когда вы убедитесь в том, что ваши методы классификации работают, следующим шагом будет подсчитать вероятность выпадения различных раскладов у игрока. Напишите для файла `PokerHand.py` функцию, которая будет перемешивать колоду карт, разделять ее на игроков, определять для каждого игрока наивысший расклад и подсчитывать, сколько раз встречается каждый расклад.
6. Напечатайте таблицу возможных раскладов и вероятности их выпадения. Запускайте программу со все большим и большим числом игроков, чтобы получать все более и более достоверные результаты вероятности. Сравните ваши результаты с `wikipedia.org/wiki/Hand_rankings`.

Упражнение 18.6 В этом упражнении используется `TurtleWorld` из главы 4. Вы напишете код, который заставит черепашек (объект типа `Turtle`) играть в салочки (другое название – *догонялки*; по-английски – tag). Если вы не знакомы с правилами этой игры, смотрите `wikipedia.org/wiki/Tag_(game)` или русскую версию на `http://ru.wikipedia.org/wiki/Салочки`.

1. Скачайте файл `thinkpython.com/code/Wobbler.py` и запустите его. Класс `Wobbler` наследуется от класса `Turtle`, что означает, что методы `lt`, `rt`, `fd` и `bk` будут работать и в `Wobbler`.
2. Изучите код и убедитесь в том, что вы понимаете, как он работает. Вы должны увидеть окно `TurtleWorld` и три черепашки `Turtle` в нем. Если вы нажмете кнопку `Run`, то черепашки начнут перемещаться в случайном порядке.

   Метод `step` (*шаг*) вызывается классом `TurtleWorld`. Он вызывает метод `steer` (*поворот*), который заставляет черепашку поворачивать в желаемом направлении, а также метод `wobbler`, который добавляет случайный поворот, пропорциональный неуправляемости объекта `Turtle`, и метод `move`, который перемещает ее на несколько пикселей вперед, в зависимости от скорости черепашки.
3. Создайте файл `Tagger.py`. Импортируйте в него все из класса `Wobbler`, затем определите класс

`Tagger`, который должен наследоваться от `Wobbler`. Создайте вызов функции `make_world`, передав ей объект `Tagger` в качестве аргумента.

4. Добавьте метод `steer` к классу `Tagger`, чтобы заместить тот, что наследовался от `Wobbler`. Для начала напишите такую версию, чтобы он всегда направлял `Turtle` в сторону начала координат. Подсказка: используйте математическую функцию `atan2` и атрибуты `Turtle` `x`, `y` и `heading`.
5. Измените метод `steer` таким образом, чтобы черепашки оставались в пределах границ. Для отладки вы можете воспользоваться кнопкой *Step*, которая вызывает метод `step` по одному разу для каждой черепашки.
6. Измените метод `step` таким образом, чтобы каждая черепашка направлялась на своего ближайшего соседа. Подсказка: объекты `Turtle` имеют атрибут `world`, который является ссылкой на `TurtleWorld`, в котором они живут, а `TurtleWorld`, в свою очередь, обладает атрибутом `animals`, т.е. списком всех черепашек в данном мире.
7. Измените `steer` таким образом, чтобы черепашки играли в салочки. Вы можете добавлять методы в класс `Taggler`, а также вы можете замещать `steer` и `__init__`, но вы не можете изменять или замещать `step`, `wobble` или `move`. Также методу `steer` разрешается изменять направление, но не положение черепашки.

   Вы можете немного изменить правила и метод `steer` таким образом, чтобы игра получилась более качественной. Например, у медленной черепашки должна быть возможность осалить (т.е. догнать) более быструю черепашку.

Вы можете посмотреть мою версию программы здесь: `thinkpython.com/code/Tagger.py`.

Глава 19
Углубленное изучение: Tkinter

19.1 GUI (графический интерфейс)

Большинство из тех программ, которые мы здесь видели, выполняются в текстовом режиме, но у многих программ имеется **графический интерфейс пользователя** (graphical user interface), также называемый сокращенно GUI.

В Python имеется возможность выбора между различными системами для построения программ с GUI, включая wxPython, Tkinter и Qt. У каждой из этих систем есть свои преимущества и недостатки, поэтому здесь нет общепринятого стандарта.

В этой главе мы познакомимся с Tkinter, т.к., на мой взгляд, эта система проще всего для начинающих. Большинство из того, что будет изучено в этой главе, можно применить и к другим модулям GUI.

Есть несколько книг и веб-сайтов, посвященных Tkinter. Одним из лучших Интернет-ресурсов является *An Introduction to Tkinter* (Введение в Tkinter), автор Fredrik Lundh.

Довольно часто модуль `Tkinter` идет по умолчанию с Python, но не всегда. Проверить, установлен ли этот модуль, можно импортировав его:

```
>>> import Tkinter
```

Если после этого вы увидите приглашение >>>, то все нормально, модуль установлен. Если появится сообщение об ошибке, говорящее, что данный модуль не найден, то вам нужно установить его дополнительно из репозиториев.

В некоторых системах название модуля пишется с маленькой буквы:

```
>>> import tkinter
```

Я написал модуль `Gui.py`, который содержится в `Swampy`. Он предоставляет упрощенный интерфейс к функциям и классам `Tkinter`. Примеры этой главы основаны на этом модуле.

Ниже дан простой пример, который создает и отображает GUI.

Для создания GUI вам необходимо будет импортировать `Gui` и создать экземпляр класса `Gui`:

```
from Gui import *
g = Gui()
g.title('Gui')
g.mainloop()
```

Если вы запустите этот код, то должно появиться пустое окно, состоящее из серого квадрата, с надписью `Gui` в заголовке. `mainloop` запускает **цикл с ожиданием события** (event loop), который ожидает действий пользователя и реагирует на них соответственно. Это бесконечный цикл. Он выполняется до тех пор, пока пользователь не закроет окно или не нажмет `Ctrl-C`, или не сделает что-нибудь, чтобы программа прекратила выполнение.

Этот GUI ничего особенного не делает, т.к. в нем нет **виджетов** (widget). Виджеты это элементы, из которых состоит графический интерфейс. Они включают в себя:

кнопка (button): виджет, который содержит текст или изображение, и совершает некоторое действие при нажатии на него.

канва (canvas): область, которая может отображать линии, прямоугольники, окружности и другие фигуры.

поле ввода (entry): область, где пользователь может набирать текст.

полоса прокрутки (scrollbar): виджет, который управляет видимой частью другого виджета.

фрейм (frame): контейнер, часто невидимый, который содержит другие виджеты.

Тот пустой серый квадрат, который вы видите при создании GUI и есть фрейм. Когда вы создаете новый виджет, он добавляется к этому фрейму.

19.2 Кнопки и обратные вызовы

Метод `bu` создает виджет `Button` (кнопка):

```
button = g.bu(text='Нажми меня.')
```

Значение, возвращаемое `bu`, это объект `Button`. Кнопка, которая появляется на фрейме, представляет из себя графическое отображение этого объекта. Вы можете управлять кнопкой, вызывая на нее методы.

bu принимает до 32 параметров, которые контролируют ее внешний вид и функциональность. Эти параметры называются **опциями** (option). Вместо того, чтобы задавать все 32 опции, вы можете использовать ключевые слова в качестве аргументов, типа `text='Нажми меня'`, чтобы задать необходимые опции, а остальные оставить со значениями по умолчанию.

Когда вы добавляете на фрейм виджет, то размер фрейма уменьшается до размера виджета. Если вы добавите больше виджетов, то фрейм увеличится в размере соответственно.

Метод `la` создает виджет `Label` (метка):

```
label = g.la(text='Нажми на кнопку.')
```

По умолчанию `Tkinter` помещает новые виджеты наверху и выравнивает их по центру. Вскоре мы увидим, как изменить такое поведение.

Если вы нажмете на кнопку, то обнаружите, что ничего особенного не происходит. Это потому, что вы еще не сказали ей, что она должна делать!

Опция, управляющая поведением кнопки, называется `command`. Значением `command` является функция, выполняемая при нажатии кнопки. Например, вот функция, создающая новый виджет `Label`:

```
def make_label():
    g.la(text='Спасибо.')
```

Теперь мы можем создать кнопку с этой функцией в качестве команды:

```
button2 = g.bu(text='Нет, нажми меня!', command=make_label)
```

Когда вы нажимаете на эту кнопку, то она должна выполнить функцию `make_label`, и новая метка должна появиться.

Значением опции `command` является объект типа `function`, который еще называют **обратный вызов** (callback), потому что после того, как вы вызываете `bu` для создания кнопки, порядок выполнения действий "вызывается обратно" к тому месту, где пользователь нажал кнопку.

Такой вид изменения потока вычислений характерен для **программам, управляемых событиями** (event driven programming). Действия пользователя, такие как нажатия на кнопки, клавиши, называют **событиями** (event). В программах, управляемых событиями, поток вычислений определяется действиями пользователя, а не программистом.

Сложность при написании программ, управляемых событиями, заключается в создании набора виджетов и обратных вызовов, которые будут работать корректно (или, как минимум, выдавать соответствующие сообщения об ошибках) при всех действиях пользователя.

Упражнение 19.1 Напишите программу, которая создает GUI с единственной кнопкой. При нажатии этой кнопки она должна создавать вторую кнопку. Нажатие второй кнопки должна создаваться надпись `"Отличная работа!"`

Что произойдет, если вы нажмете кнопку более одного раза? Вы можете посмотреть мой код здесь: `thinkpython.com/code/button_demo.py`

19.3 Виджет canvas

Одним из виджетов многоцелевого назначения является виджет Canvas, который создает область для рисования линий, окружностей и других фигур. Если вы выполняли упражнение 15.4, то вы уже должны быть знакомы с этим виджетом.

Метод ca создает новый виджет Canvas:

```
canvas = g.ca(width=500, height=500)
```

Параметры width и height задают размер канвы в пикселях (ширина и высота соответственно).

После того, как вы создали виджет, вы все еще можете изменить значения его опций при помощи метода config. Например, опция bg (от слова *background*) изменяет цвет фона:

```
canvas.config(bg='white')
```

Значением bg является текстовая строка с названием цвета. Список доступных цветов может отличаться в зависимости от используемой версии Python, но все версии должны предоставлять, как минимум, следующие цвета:

```
white       black
red         green       blue
cyan        yellow      magenta
```

Фигуры на канве называются **элементами** (items). Например, метод канвы circle рисует, как вы, вероятно, догадались, круг.

```
item = canvas.circle([0,0], 100, fill='red')
```

Первый аргумент является координатной парой, задающей центр круга. Второй аргумент задает его радиус.

Gui.py предоставляет стандартные декартовы координаты с началом отсчета в центре виджета Canvas, где положительным значениям Y соответствует направление вверх. Это отличается от некоторых других графических систем, где начало отсчета находится в верхнем левом углу, а ось Y направлена вниз.

Опция fill определяет цвет заливки круга (в приведенном примере – красный).

Возвращаемым значением метода circle является объект Item, который содержит методы, способные изменять элемент на канве. Например, вы можете использовать метод config для изменения любых опций круга:

```
item.config(fill='yellow', outline='orange', width=10)
```

Здесь параметр width задает толщину окружности, а outline – ее цвет.

Упражнение 19.2 Напишите программу, которая создает канву и кнопку. При нажатии кнопки на канве должна рисоваться окружность.

19.4 Последовательность из координат

Метод rectangle (прямоугольник) берет в качестве параметров последовательность из координат, которые задают противоположные углы прямоугольника. Следующий пример рисует зеленый прямоугольник с левым нижним углом, расположенным в начале координат, а верхним правым в точке с координатами (200, 100):

```
canvas.rectangle([[0, 0], [200, 100]],
                 fill='blue', outline='orange', width=10)
```

Такой способ задания углов называется **ограничивающий прямоугольник** (bounding box), потому что эти две противоположные точки определяют прямоугольник.

Метод oval берет в качестве параметра ограничивающий прямоугольник и рисует вписанный в него овал:

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

Метод `line` (линия) берет последовательность из координат и чертит линию, соединяющую две точки. Следующий пример чертит две стороны треугольника:

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

Метод `polygon` (многоугольник) берет те же аргументы, но чертит последнюю сторону многоугольника (если необходимо) и заливает его красным цветом:

```
canvas.polygon([[0, 100], [100, 200], [200, 100]],
               fill='red', outline='orange', width=10)
```

19.5 Больше виджетов

`Tkinter` предоставляет два виджета, позволяющие пользователю вводить текст: `Entry` – поле ввода в одну строку и `Text` – многострочное поле ввода.

Метод `en` создает новый объект `Entry`:

```
entry = g.en(text='Текст по умолчанию.')
```

Опция `text` позволяет вам поместить текст в это поле уже после его создания, а метод `get` возвращает содержимое объекта `Entry` (которое могло быть уже изменено пользователем):

```
>>> entry.get()
'Текст по умолчанию.'
```

Метод `te` создает виджет `Text`:

```
text = g.te(width=100, height=5)
```

Параметры `width` и `height` задают размеры виджета в символах и строках соответственно.

Метод `insert` помещает текст в виджет `Text`:

```
text.insert(END, 'Строка текста.')
```

`END` это специальный индекс, который означает последний символ в виджете `Text`.

Вы можете также задавать положение символов при помощи точечной записи, например, 1.1, где число до точки означает номер строки, а число после – номер колонки. Следующий пример добавляет буквы `'nother'` первого символа первой строки:

```
>>> text.insert(1.1, 'nother')
```

Метод `get` читает текст в виджете. В качестве аргументов он принимает начальный и конечный индексы. Следующий пример возвращает весь текст виджета, включая символ новой строки:

```
>>> text.get(0.0, END)
'Другая строка текста.\n'
```

Метод `delete` удаляет текст из виджета. Следующий пример удаляет весь текст, кроме первых двух символов:

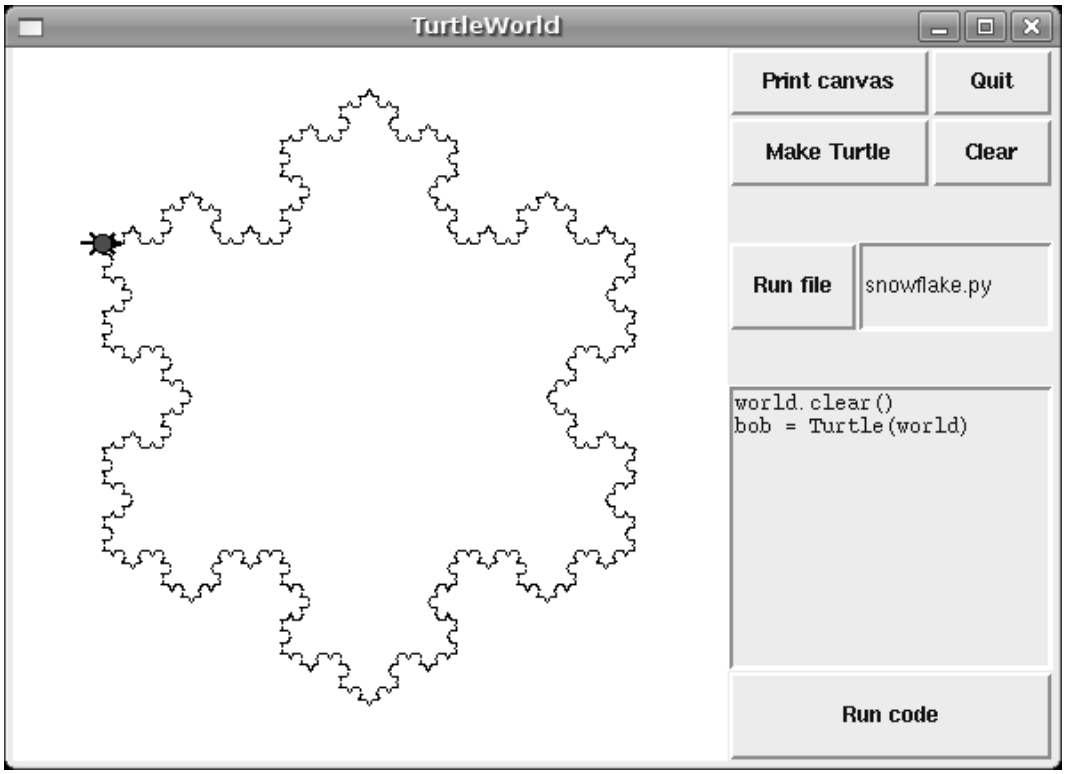
```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
'Др\n'
```

Упражнение 19.3 Измените ваше решение к упражнению 19.2, добавив виджет `Entry` и вторую кнопку. Когда пользователь нажимает на вторую кнопку, программа должна считывать название цвета, введенное в `Entry` и использовать его для изменения цвета круга. Используйте метод `config` для изменения существующего круга; не создавайте новый круг.

Ваша программа должна корректно справляться с ситуацией, когда пользователь пытается изменить цвет несуществующего круга, а так же когда задан неправильный цвет.

19.6 Упаковка виджетов

До сих пор мы помещали виджеты в одну колонку, но большинство программ имеют более сложный интерфейс. Например, вот слегка упрощенная версия `TurtleWorld` (см. главу 4):



В данном разделе содержится разбитый на части код, создающий этот интерфейс. Вы можете скачать законченный пример отсюда: `thinkpython.com/code/SimpleTurtleWorld.py`

В самой своей основе этого GUI лежат два виджета – `Canvas` и `Frame`, помещенные в один ряд. Поэтому первым шагом будет создание этого ряда:

```
class SimpleTurtleWorld(TurtleWorld):
    """Этот класс идентичен классу TurtleWorld,
       но код, который создает интерфейс, упрощен для лучшего понимания.
    """

    def setup(self):
        self.row()
        ...
```

Функция setup создает и располагает виджеты. Процесс размещения виджетов в GUI называется **упаковкой** (packing).

Метод `row` создает объект `Frame` и делает его текущим. До тех пор, пока этот `Frame` не будет закрыт, или не будет создан новый `Frame`, все последующие виджеты будут располагаться в ряд.

Следующий код создает `Canvas` и колонку `Frame`, которая содержит другие виджеты:

```
self.canvas = self.ca(width=400, height=400, bg='white') self.col()
```

Первым виджетом в колонке является `Frame` в виде сетки (grid), которая содержит четыре кнопки, расположенные два-на-два:

```
self.gr(cols=2)
self.bu(text='Print canvas', command=self.canvas.dump)
self.bu(text='Quit', command=self.quit)
self.bu(text='Make Turtle', command=self.make_turtle)
self.bu(text='Clear', command=self.clear)
self.endgr()
```

Метод `gr` создает сетку; аргументом является число колонок. Виджеты в сетке располагаются слева направо и сверху вниз.

Первая кнопка в качестве обратного вызова использует `self.canvas.dump`, вторая - `self.quit`. Это называется **связанными методами**, потому что они относятся к определенным объектам. Когда эти методы вызываются, они вызываются на свой объект.

Следующим виджетом в колонке является `Frame`, содержащий в ряд виджеты `Button` и `Entry`:

```
self.row([0,1], pady=30)
self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5)
self.endrow()
```

Первым аргументом метода `row` (ряд) служит список, определяющий, как лишнее место распределяется между виджетами. [0, 1] означает, что все лишнее место отводится второму виджету, `Entry`. Если вы запустите этот код и измените размер окна, вы увидите, что `Entry` увеличивается в размере, а `Button` нет.

Опция `pady` (прокладка) как бы "заполняет" ряд в направлении *y*, добавляя по 30 пикселей сверху и снизу.

`endrow` заканчивает этот ряд виджетов, так что последующие виджеты пакуются в колонке `Frame`. `Gui.py` содержит стек из объектов `Frame`:

- Когда вы используете методы `row`, `col` или `gr` для создания фрейма `Frame`, он помещается на верху стека и становится текущем фреймом.
- Когда вы используете `endrow`, `endcol` или `endgr` для закрытия `Frame`, этот фрейм удаляется из стека и предыдущий становится текущим.

Метод `run_file` (*запустить файл*) читает содержимое поля `Entry`, использует его в качестве имени файла, читает его содержимое и передает его методу `run_code` (*запустить код*). `self.inter` это объект `Interpreter` (*интерпретатор*), который знает, как взять строку и выполнить ее так, как код Python.

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

Два последних виджета это виджеты `Text` и `Button`:

```
self.te_code = self.te(width=25, height=10) self.te_code.insert(END,
'world.clear()\n') self.te_code.insert(END, 'bob = Turtle(world)\n')
self.bu(text='Run code', command=self.run_text)
```

`run_text` похож на `run_file`, за исключением того, что он берет код из виджета `Text`, а не из файла:

```
def run_text(self):
    source = self.te_code.get(1.0, END)
    self.inter.run_code(source, '<user-provided code>')
```

К сожалению, детали того, как располагаются виджеты, различаются в разных языках, а также в разных версиях Python. В самом `Tkinter` содержатся три механизма для расположения виджетов. Эти механизмы называются **менеджерами геометрии** (geometry manager). Тот, который я продемонстрировал в данном разделе, называется "сеточным" (grid) механизмом; остальные два называются "упаковочный" (pack) и "размещающий" (place).

К счастью, большинство из концепций данного раздела относится также и к другим модулям GUI, а также к другим языкам программирования.

19.7 Меню и объект `Callable`

Виджет Menubutton выглядит как кнопка, но отличается тем, что при ее нажатии выпадает меню. После того, как пользователь выберет любой пункт, меню исчезает. Вот код, который создает виджет Menubutton для выбора цвета (вы можете скачать его отсюда: thinkpython.com/code/menubutton_demo.py):

```
g = Gui()
g.la('Select a color:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

Метод mb создает Menubutton. Вначале текст на кнопке соответствует цвету по умолчанию. Следующий цикл создает по одному пункту меню для каждого цвета:

```
for color in colors:
    g.mi(mb, text=color, command=Callable(set_color, color))
```

Первым аргументом метода mi (menu item – *пункт меню*) является виджет Menubutton, для которого создаются эти пункты.

Опция command представляет из себя объект Callable (*то, что можно вызвать*), что является для вас нечто новым. До сих пор мы встречались с функциями и связанными методами, используемыми в качестве обратных вызовов (callback), что прекрасно работает до тех пор, пока вам не нужно передавать в функцию никаких аргументов. В противном случае вам будет необходимо создать объект Callable, который содержит функцию, такую как set_color (*установить цвет*), и ее аргументы, например, color.

Объект Callable содержит ссылки на функции и аргументы в качестве атрибутов.

Вот как может выглядеть функция set_color:

```
def set_color(color):
    mb.config(text=color)
    print(color)
```

Когда пользователь выбирает пункт меню, происходит вызов функции set_color. При этом Menubutton отображает вновь выбранный цвет. Название цвета также распечатывается. Если вы запустите этот пример, то сможете убедиться в том, что функция set_color вызывается тогда, когда вы выбираете пункт меню (и *не* вызывается тогда, когда вы создаете объект Callable).

19.8 Связь

Между виджетом (widget), событием (event) и обратным вызовом (callback) существует **связь** (binding): когда происходит какое-либо событие, например, нажатие, выполняется обратный вызов.

У многих виджетов существуют связи по умолчанию. Например, когда вы нажимаете кнопку, то связь, существующая по умолчанию, изменяет рельеф кнопки так, что она выглядит нажатой. Когда вы отпускаете кнопку, то существующая связь восстанавливает ее прежний облик и вызывает обратный вызов, определенный в опции command.

Вы можете использовать метод bind для замещения связей, существующих по умолчанию, на новые. Например, следующий код создает связь для канвы (код, используемый в этом разделе, вы можете скачать отсюда: thinkpython.com/code/draggable_demo.py).

```
ca.bind('<ButtonPress-1>', make_circle)
```

Первым аргументом является строка, представляющая собой имя события. Данное событие возникает, когда пользователь нажимает левую кнопку мыши. Другие события, связанные с мышью, включают в себя ButtonMotion, ButtonRelease и Double-Button.

В качестве второго аргумента выступает обработчик событий. Обработчик событий представляет из себя функцию или связанный (bound) метод, как, например, обратный вызов (callback), но важное различие заключается в том, что обработчик событий воспринимает объект Event (событие) в качестве параметра.

Вот простой пример:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

Объект `Event` содержит информацию о типе события и другие детали, как, например, координаты указателя мыши. В этом примере информация, которая нам нужна, это положение указателя, в котором происходит щелчок мыши. Эти значения выражаются в "координатах пикселей", что определено в графической системе ОС. Метод `canvas_coords` переводит эти координаты в "координаты на канве" для совместимости с методами объекта `Canvas`, такими как, например, `circle`.

Виджеты `Entry` часто связывают с событием `Return`, которое возникает, когда пользователь нажимает клавишу <Return> или <Enter>. Например, следующий код создает `Button` и `Entry`.

```
bu = g.bu('Make text item:', make_text)
en = g.en()
en.bind('<Return>', make_text)
```

Функция `make_text` вызывается тогда, когда нажата кнопка, или пользователь нажимает <Return> во время набора текста в поле `Entry`. Для этого нам требуется функция, которую можно вызывать как команду (без аргументов) или как обработчик событий (с событием `Event` в качестве аргумента):

```
def make_text(event=None):
    text = en.get()
    item = ca.text([0,0], text)
```

Функция `make_text` получает содержимое поля `Entry` и в качестве элемента `Text` на канве `Canvas`.

Возможно также создавать связи для элементов канвы. Ниже дано определение класса `Draggable`, являющимся дочерним классом класса `Item`. Это класс предоставляет связи, которые можно использовать для технологии перетаскивания объектов мышью (технология drag-and-drop).

```
class Draggable(Item):
    def __init__(self, item):
        self.canvas = item.canvas
        self.tag = item.tag
        self.bind('<Button-3>', self.select)
        self.bind('<B3-Motion>', self.drag)
        self.bind('<Release-3>', self.drop)
```

Метод `__init__` принимает объект `Item` в качестве параметра. Он копирует атрибуты объекта `Item` и затем создает связи для трех событий: нажатие кнопки, движение кнопки, отпускание кнопки.

Обработчик события `select` хранит координаты текущего события и оригинального цвета элемента, а затем изменяет этот цвет на желтый:

```
def select(self, event):
    self.dragx = event.x
    self.dragy = event.y
    self.fill = self.cget('fill')
    self.config(fill='yellow')
```

Метод `cget` (get configuration) означает "получить конфигурацию"; он берет имя опции в качестве строки и возвращает текущее значение этой опции.

Функция `drag` вычисляет расстояние, на которое переместился объект относительно своего начального положения, обновляет хранящиеся координаты и затем перемещает сам элемент.

```
def drag(self, event):
    dx = event.x - self.dragx
    dy = event.y - self.dragy
    self.dragx = event.x
    self.dragy = event.y
    self.move(dx, dy)
```

Вычисления производятся в координатах пикселей. Нет нужды переводить их в координаты канвы. Наконец, функция `drop` возвращает первоначальный цвет элемента:

```
def drop(self, event):
    self.config(fill=self.fill)
```

Вы можете использовать класс `Draggable` для того, чтобы иметь возможность перетаскивать существующие элементы. Например, вот измененная версия `make_circle`, которая использует метод `circle` для создания экземпляров `Item` и `Draggable`, чтобы их можно было перетаскивать:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
    item = Draggable(item)
```

Данный пример демонстрирует одно из преимуществ наследования: вы можете изменять характеристики родительского класса, не изменяя, при этом, его определения. В частности, это полезно, когда вы хотите внести изменения в поведение модуля, который писали не вы.

19.9 Отладка

Одной из трудностей при проектировании GUI заключается в том, чтобы отслеживать то, что происходит при создании GUI, и что происходит после в качестве ответа на действия пользователя.

Например, при написании обратного вызова (callback) часто допускают ошибку, состоящую в том, что вызывают саму функцию, вместо того, чтобы передать в качестве параметра ссылку на нее:

```
def the_callback():
    print('Called.')

g.bu(text='Неправильно!', command=the_callback())
```

Если вы запустите данный код, вы увидите, что он немедленно вызовет функцию `the_callback`, а уже *затем* создаст кнопку. Если вы нажмете на эту кнопку, то ничего не произойдет, потому что возвращаемым значением функции `the_callback` будет `None`. Как правило, вам не нужно вызывать callback при создании GUI; его нужно вызывать позже в качестве ответа на действия пользователя.

Другой трудностью при создании GUI является то, что вы не можете управлять порядком выполнения действий. То, какие программы выполняются и их порядок, определяется действиями пользователя. Это означает, что вам необходимо проектировать вашу программу таким образом, чтобы она работала корректно при всех возможных событиях.

Например, GUI из упражнения 19.3 содержит два виджета: один создает элемент `Circle`, а другой изменяет цвет этого элемента. Если пользователь создает круг и затем меняет его цвет, то не возникает никаких проблем. Но что произойдет, если пользователь будет изменять цвет круга, который еще не был создан? Или если он создаст более одного круга?

С увеличением числа виджетов становится чрезвычайно трудно представить себе все возможные последовательности событий. Один из способов справиться с этой трудностью заключается в том, чтобы инкапсулировать состояние системы в объект. Затем нужно рассмотреть:

- Каковы могут быть возможные состояния? В примере с виджетом `Circle` мы можем рассмотреть два состояния: до и после того, как пользователь создаст первый круг.
- Какие события могут произойти в каждом состоянии? В данном примере, пользователь может нажать любую из двух кнопок или выйти из программы.
- Каков желаемый результат для каждой пары состояние-событие? Т.к. есть два состояния и две кнопки, существует четыре пары состояние-событие для рассмотрения.
- Что может вызвать переход из одного состояния в другое? В данном случае, переход происходит тогда, когда пользователь создает первый круг.

Также вам может пригодиться определение и проверка инвариантов, которые должны соблюдаться всегда вне зависимости от последовательности событий.

Данный подход к программированию GUI может помочь вам в написании корректно работающего кода, и при этом вам не придется тратить время на то, чтобы испытать все возможные последовательности событий, которые может вызвать пользователь!

19.10 Словарь терминов

GUI (graphical user interface): графический интерфейс пользователя.

виджет (widget): один из элементов, составляющий графический интерфейс, что включает в себя кнопки, меню, текстовые поля и т.п.

опция (option): значение, которое определяет то, как выглядит или функционирует виджет.

ключевой аргумент (keyword argement): аргумент, показывающий имя параметра как часть вызова функции.

обратный вызов (callback): функция, связанная с виджетом, вызываемая, когда пользователь производит некоторые действия.

связанный метод (bound method): метод, связанный с определенным экземпляром класса.

программа, управляемая событиями (event-driven programming): стиль программирования, при котором порядок выполнения вычислений определяется действиями пользователя.

событие (event): действие пользователя, такое, как нажатие клавиши или кнопки мыши, которое вызывает ответ GUI.

цикл, ожидающий события (event loop): бесконечный цикл, который ожидает от пользователя действий и реагирует на них соответственно.

элемент (item): графический элемент, располагаемый на виджете `Canvas` (канва).

ограничивающий прямоугольник (bounding box): прямоугольник, как правило, невидимый, который включает в себя набор некоторых элементов; обычно задается двумя противоположными углами.

упаковка (pack): размещение и отображение элементов GUI.

менеджер геометрии (geometry manager): система упаковки виджетов.

связь (binding): связь, установленная между виджетом, событием и обработчиком событий. Обработчик событий вызывается тогда, когда данное событие происходит в виджете.

19.11 Упражнения

Упражнение 19.4 Для этого упражнения вы напишите программу для просмотра изображений. Вот простой пример:

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

Функция `PhotoImage` читает файл и возвращает объект `PhotoImage`, который `Tkinter` способен отобразить. `canvas.image` помещает это изображение на канве с центром, расположенным в данных координатах. Также вы можете помещать изображения на метки `Label`, кнопки `Button` и на некоторые другие виджеты:

```
g.la(image=photo)
g.bu(image=photo)
```

`PhotoImage` понимает лишь несколько форматов изображений, таких как GIF и PPM, но мы можем использовать библиотеку Python Imaging Library (PIL) для чтения и других форматов изображений.

Имя этого PIL модуля – `Image`, но в `Tkinter` уже имеется объект с таким же именем. Для предотвращения конфликта вам необходимо использовать выражение `import ... as`, что может выглядеть, например, так:

```
import Image as PIL
import ImageTk
```

Первая строка импортирует модуль `Image` и дает ему локальное имя `PIL`. Вторая строчка импортирует модуль `ImageTk`, который может преобразовывать изображения, понимаемые PIL в те, что понимает `Tkinter PhotoImage`. Вот пример:

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. Скачайте файлы `image_demo.py`, `danger.gif` и `allen.gif` с сайта `thinkpython.com/code`. Запустите файл `image_demo.py`. Вам может потребоваться установить PIL и ImageTk. Возможно, они находятся в репозиториях вашей операционной системы. Если нет, то вы можете найти их по адресу: `pythonware.com/products/pil/`.

2. В файле `image_demo.py` измените имя второго `PhotoImage` с `photo2` на `photo` и запустите программу снова. Вы должны увидеть второй `PhotoImage`, но не первый.
   Проблема заключается в том, что когда вы присваиваете новое значение `photo`, оно замещает ссылку на первый `PhotoImage`, который исчезает. То же самое происходит и тогда, когда вы присваиваете `PhotoImage` локальной переменной; он исчезает, когда функция заканчивает работу.
   Для решения этой проблемы вам необходимо хранить ссылки для каждого объекта `PhotoImage`, который вы хотите использовать. Вы можете использовать глобальную переменную, либо хранить `PhotoImage` в какой-либо структуре данных, либо в качестве атрибута какого-либо объекта.
   Такое поведение может вас запутать, и именно поэтому я предупреждаю вас об этом (и поэтому изображение говорит "Danger!" – *опасность*).

3. Взяв за основу этот пример, напишите программу, которая принимает имя директории и просматривает все файлы, содержащиеся в ней, отображая те из них, которые PIL распознает в качестве изображений. Вы можете использовать выражение `try`, чтобы отсеять файлы, которые PIL не понимает.
   Когда пользователь щелкает по изображению, программа должна показывать следующее.

4. PIL предоставляет множество методов для манипуляциями с изображениями. Вы можете прочитать о них по следующему адресу: `pythonware.com/library/pil/handbook`. В качестве дополнительного упражнения, выберете некоторые из этих методов и снабдите их графическими элементами для того, чтобы применять их к изображениям.

Вы можете посмотреть на простое решение здесь: `thinkpython.com/code/ImageBrowser.py`

Упражнение 19.5 Векторный графический редактор это программа, позволяющая пользователю рисовать и редактировать на экране фигуры, а также сохранять их в векторном формате, таком как Postscritp или SVG.[32]

Используя `Tkinter`, напишите простой графический редактор. Как минимум, он должен позволять пользователю рисовать пользователю линии, круги и прямоугольники, а также он должен использовать `Canvas.dump` для создания Postscript описаний содержимого канвы.

В качестве дополнительного упражнения вы можете создать возможность для пользователя выбирать и изменять элементы на канве.

Упражнение 19.6 Используйте Tkinter для написания простого веб-браузера. В нем должен быть виджет `Text`, в который вы можете вводить URL адрес, а также канва для отображения содержимого страницы.

Вы можете использовать модуль `urllib.request` для скачивания файлов (см. упражнение 14.5) и модуль `HTMLParser` для парсинга HTML тэгов (см. `docs.python.org/lib/moduleHTMLParser.html`).

Как минимум, ваш браузер должен отображать простой текст и гиперссылки. Дополнительно вы можете попытаться отображать цвет фона, форматированный текст и изображения.

[32] См. `wikipedia.org/wiki/Vector_graphics_editor`.

Приложение A
Отладка

В программах могут встречаться ошибки разного вида, поэтому полезно их различать для того, чтобы быстрее их находить и исправлять:

- Синтаксические ошибки (syntax errors) возникают во время того, как Python переводит код, написанный вами, в байтовый код. Эти ошибки указывают, как правило, на то, что что-то неверно с синтаксисом программы. Пример: если вы пропустите двоеточие в конце заголовка объявления функции с ключевым словом `def`, вы получите сообщение `SyntaxError: invalid syntax`.
- Ошибки во время выполнения программы (runtime errors) генерируются интерпретатором тогда, когда во время выполнения программы что-то идет не так. Большинство таких сообщений об ошибках включают в себя информацию о том, где произошла ошибка и какая функция при этом исполнялась. Пример: бесконечная рекурсия вызывает, в конце концов, ошибку во время выполнения с сообщением "`maximum recursion depth exceeded`" (*достигнута максимальная глубина рекурсии*).
- Семантические ошибки (semantic errors) представляют из себя проблемы в программе, которая при своей работе не выводит сообщений об ошибках, но делает не то, что от нее требуется. Пример: выражение может вычисляться не в том порядке, в каком вы ожидали; в результате вы получите неправильный результат.

Прежде всего при отладке ошибок необходимо выяснить, с каким видом ошибки вы имеете дело. Хотя следующий раздел разбит по типам ошибок, некоторые способы можно применять в различных ситуациях.

A.1 Синтаксические ошибки

Как правило, синтаксические ошибки легко исправить, как только вы выясните, где они произошли. К сожалению, сообщения об ошибках часто этому не слишком способствуют. Чаще всего выдаются сообщения `SyntaxError: invalid syntax` (Синтаксическая ошибка: неверный синтаксис) или `SyntaxError: invalid token` (Синтаксическая ошибка: неверный токен). Но они не информативны.

С другой стороны, эти сообщения говорят вам, где произошла ошибка. Вообще-то, они говорят о том, где Python обнаружил проблему, а это далеко не всегда то место, где находится сама ошибка. Иногда ошибка содержится в коде, который находится до того места, о котором говорит сообщение об ошибке, часто на предыдущей строке.

Если вы пишете программу маленькими порциями за раз, то вам будет сравнительно легко понять, где находится ошибка. Она будет в коде, который вы добавили последним.

Если вы перепечатываете код из книги, начните тщательно сравнивать ваш код с кодом из книги. Проверьте каждый символ. В то же самое время помните, что и книга может ошибаться. Поэтому если вы видите что-то похожее на синтаксическую ошибку, то оно вполне может этим быть.

Вот несколько способов, как можно предотвратить типичные синтаксические ошибки:

1. Убедитесь, что вы не используете одно из ключевых слов Python в качестве имени переменной.
2. Проверьте, что каждое групповое выражение имеет в конце заголовка двоеточие. Это включает в себя `for`, `while`, `if` и `def`.
3. Убедитесь, что кавычки, в которые заключены строки (тип `string`), имеют свои пары.
4. Если вы используете многолинейные строки с тройными кавычками (`"""` или `'''`), убедитесь, что строка закончена правильно. Неправильно законченная строка может вызвать ошибку `invalid token` в конце вашей программы, или Python будет рассматривать ваш последующий код в качестве продолжения строки, пока не встретится следующая строка. Во втором случае может даже вообще не появится сообщения об ошибке.

5. Незакрытые открывающие операторы, такие как `(`, `{` и `[` – заставляют Python считать следующую линию частью текущего выражения. Обычно ошибка происходит уже на следующей линии.
6. Не забывайте о разнице между = (присваивание) и == (сравнение).
7. Проверяйте отступы. Python может работать как с пробелами, так и со знаками табуляции, но если вы их смешиваете, это может вызвать ошибку. Лучший способ избежать подобных ошибок это использование текстового редактора, которому знаком синтаксис Python, и который последовательно делает за вас отступы.

Если ничего из этого не помогает, переходите к следующему разделу…

A.1.1 Я вношу изменения, но это не помогает

Если интерпретатор говорит, что есть ошибка, а вы ее не видите, это может быть вызвано тем, что вы и интерпретатор смотрите на разный код. Проверьте ваше программное окружение и убедитесь, что программа, которую вы редактируете, и программа, которую Python пытается выполнить – одно и то же.

Если вы не уверены, то попробуйте в самом начале программы поместить явную ошибку, и запустите программу снова. Если интерпретатор не обнаружит этой новой ошибки, значит он выполняет другой код.

Этому может быть несколько причин:

- Вы редактировали файл и забыли сохранить внесенные изменения перед тем, как запустить программу. Некоторые редакторы всегда делают это за вас автоматически, а другие нет.
- Вы изменили имя файла, но запускаете старый.
- Что-то в вашем программном окружении некорректно сконфигурировано.
- Если вы пишете свой модуль и затем используете команду `import`, убедитесь, что вы не даете вашему модулю имя одного из стандартных модулей, имеющихся в Python.
- Если вы используете `import` для чтения модуля, помните, что вам необходимо перезапустить интерпретатор или использовать команду `reload`, чтобы прочитать измененный файл. Если вы просто используете `import` еще раз, то ничего не произойдет.

Если вы застряли на одном месте и не можете понять, в чем дело, то вам может помочь следующий подход: начните с новой простой программы, типа "`Привет, мир!`", чтобы убедиться, что заведомо исправная программа работает корректно. Затем начинайте постепенно добавлять к ней кусочки оригинальной программы.

A.2 Ошибки во время выполнения

Когда ваша программа имеет правильный синтаксис, Python может ее скомпилировать и, как минимум, начать выполнять. Что при этом может пойти не так?

A.2.1 Моя программа абсолютно ничего не делает

Такая проблема часто встречается тогда, когда ваш файл состоит из функций и классов, но на самом деле не вызывает ни один из них. Такое поведение может быть намеренным, если вы планируете только импортировать этот модуль ради его функций и классов.

Если не намеревались использовать модуль таким образом, убедитесь, что вы вызываете какую-либо функцию для начала работы, или выполняете ее в интерактивном режиме. Также смотрите раздел "Поток вычислений" ниже.

A.2.2 Моя программа зависает

Если программа останавливается, и возникает ощущение, что ничего не происходит, то это "зависание". Часто это бывает вызвано бесконечным циклом или бесконечной рекурсией.

- Если есть какой-либо цикл, который вы подозреваете, то добавьте непосредственно перед ним функцию `print`, чтобы она выводила надпись "`вхождение в цикл`", и сразу же после цикла с надписью "`выход из цикла`".

Запустите программу. Если вы видите первую надпись, но не видите второй, значит вы попали в бесконечный цикл. Смотрите раздел "Бесконечный цикл" ниже.

- При бесконечной рекурсии чаще всего программа будет выполняться какое-то время, затем интерпретатор выдаст сообщение об ошибке `RuntimeError: Maximum recursion depth exceeded` (Ошибка при выполнении: достигнута максимальная глубина рекурсии). Если такое происходит, идите к разделу "Бесконечная рекурсия" ниже.

  Если вы не получаете такой ошибки, но, все же, подозреваете, что ошибка находится в рекурсивном методе или функции, то все равно советы из раздела "Бесконечная рекурсия" могут вам помочь.

- Если предыдущие шаги не помогли, то начинайте тестировать другие циклы и другие рекурсивные функции и методы.
- Если ничего из вышеперечисленного не помогло, то, вероятно, вы не понимаете, каков поток вычислений вашей программы. Идите к разделу "Поток вычислений" ниже.

Бесконечный цикл

Если вы считаете, что у вас есть бесконечный цикл, и если вы думаете, что знаете, в каком именно цикле произошла эта ошибка, то добавьте вызов функции `print` в конец цикла, которая будет выводить значения переменных, содержащихся в условии цикла, и значение этого условия.

Вот пример:

```
while x > 0 and y < 0:
    # сделать что-то с x
    # сделать что-то с y
    print("x: ", x)
    print("y: ", y)
    print("условие: ", (x > 0 and y < 0))
```

Теперь когда вы запустите программу, то после каждого прохождения цикла вы увидите три строки вывода. При последнем проходе цикла условие должно быть `False`. Если цикл все еще будет выполняться, то вы увидите значения `x` и `y` и можете подумать, почему они не обновляются корректно.

Бесконечная рекурсия

Чаще всего при бесконечной рекурсии программа выполняется некоторое время, затем выводится сообщение об ошибке "`Maximum recursion depth exceeded`".

Если вы подозреваете, что некоторая функция или метод вызывают бесконечную рекурсию, убедитесь, что они достигают базового значения. Другими словами, должны быть некоторые условия, при которых функция или метод возвратятся без нового рекурсивного вызова. Если это не так, то вы должны заново продумать алгоритм и определить базовое значение.

Если есть базовое значение, но программа его не достигает, добавьте вызов функции `print` в начало функции или метода, чтобы вывести на экран параметры. Теперь, когда вы запустите программу, то каждый раз при вызове функции или метода вы будете видеть несколько строк, выводящих параметры. Если параметры не стремятся к базовому значению, то вы должны подумать, почему.

Поток вычислений

Если вы не уверены в том, в каком порядке производятся вычисления в вашей программе, добавьте вызов функции `print` при начале каждой функции со словами "`начало функции foo`", где `foo` – имя функции.

Теперь, когда вы запустите программу, вы будете видеть, в каком порядке выполняются ваши функции.

A.2.3 При запуске программы у меня происходит исключение

Если во время выполнения программы что-то идет не так, Python выводит сообщение, включающее в себя название исключения, строку программы, в которой была обнаружена проблема, и отслеживание.

В отслеживании указывается функция, которая выполнялась в тот момент, затем функция, которая вызвала *эту* функцию, затем функция, которая вызвала *ту* функцию и т.д. Другими словами,

отслеживается последовательность вызванных функций, и вас приводят туда, где вы находитесь в данный момент. Также приводятся номера строк файлов, где произошел каждый вызов.

Прежде всего необходимо проверить то место в программе, в котором произошла ошибка, чтобы увидеть, можете ли вы понять, что произошло. Ниже приведен список ошибок во время выполнения, которые происходят чаще всего:

NameError (ошибка имени): вы пытаетесь использовать имя переменной, которая не существует в данном рабочем окружении. Помните, что локальные переменные остаются локальными переменными. Вы не можете ссылаться на них вне функции, где они были определены.

TypeError (ошибка типа): Есть несколько возможных причин:

- Вы неправильно используете значение. Например, использование нецелого числа в качестве индекса строки, списка или кортежа.
- Несовпадение типов элементов в форматируемой строке (при помощи оператора `%`) и элементов, используемых для ее форматирования. Эта ошибка возникает и тогда, когда число элементов не совпадает, и тогда, когда не совпадают их типы.
- Передача неправильного количества аргументов функции или методу. В случае с методами, вы должны посмотреть на определение метода и проверить, чтобы его первый параметр был `self`. Потом посмотрите на вызов метода. Убедитесь, что вы вызываете метод на объект подходящего типа и передаете подходящие аргументы.

KeyError (ошибка ключа): вы пытаетесь получить доступ к элементу словаря, используя ключ, который в словаре не содержится.

AttributeError (ошибка атрибутов): вы пытаетесь получить доступ к несуществующему атрибуту или методу. Проверьте правильность написания атрибута! Вы можете использовать функцию `dir()` для получения списков всех существующих в данный момент атрибутов.

Если `AttributeError` сообщает, что объект имеет тип `NoneType`, то это означает, что он имеет тип `None`. Довольно часто забывают вернуть значение из функции. Если вы дошли до конца функции и не вставили команду `return`, это значит, что она возвратит `None`. Другой частой ошибкой является использование возвращаемого значения метода списка, такого, как, например, `sort`, который возвращает `None`:

```
t = [3,1,2]
t = t.sort()
```

IndexError (ошибка индекса): индекс, который вы используете для получения доступа к элементу строки, списка или кортежа, превышает его длину минус один. Непосредственно перед тем местом, где происходит ошибка, добавьте вызов функции `print`, чтобы вывести значение индекса и длину массива. Имеет ли массив правильный размер? Имеет ли индекс правильное значение?

В Python имеется модуль `pdb` для отладки ошибок. Он позволяет вам исследовать состояние программы непосредственно перед ошибкой. Вы можете прочитать о нем здесь: `docs.python.org/lib/module-pdb.html`.

A.2.4 Я добавил так много вызовов функции `print`, что потерялся в них

Одна из проблем использования вызовов функции `print` для отладки ошибок заключается в том, что вы можете потеряться во всех ее сообщениях. Есть два выхода: вы можете упростить вывод или упростить программу.

Для упрощения вывода вы можете удалить или закомментировать вызовы `print`, если они вам не помогают, либо объединить их, либо отформатировать вывод так, чтобы его легче было понимать.

Для упрощения программы вы можете сделать несколько вещей. Во-первых, уменьшите размер вашей проблемы, с которой вы работаете. Например, если вы производите поиск в списке, ищите в *маленьком* списке. Если программа принимает данные от пользователя, введите самые простые данные, которые вызывают проблему.

Во-вторых, почистите программу. Удалите мертвый код. Реорганизуйте ее так, чтобы ее было проще читать. Например, если вы подозреваете, что проблема скрывается в одной из глубоко вложенных

частей программы, попытайтесь переписать эту часть, используя более простую структуру. Если ваше подозрение падает на большую функцию, попытайтесь разбить ее на маленькие функции и протестировать их по отдельности.

Часто процесс нахождения минимальных значений для тестирования приводит вас к обнаружению ошибки. Если вы обнаружите, что программа работает в одной ситуации и не работает в другой, это может дать вам ключ к пониманию того, что происходит.

Подобным же образом, если вы переписываете небольшую часть кода, это может помочь вам обнаружить трудноуловимую ошибку. Если вы внесли незначительные изменения, о которых вы думали, что они не должны были затронуть программу, а они затрагивают, то это может дать вам повод задуматься.

A.3 Семантические ошибки

Как правило, обнаруживать семантические ошибки труднее всего, т.к. интерпретатор не выдает вам никаких сообщений об ошибках. Вы должны понимать, как работает программа.

Прежде всего необходимо установить соответствие между кодом программы и ее поведением, которое вы видите. Вы должны строить предположения относительно того, что программа делает на самом деле. Это затрудняется тем, что компьютеры работают слишком быстро.

Часто вам будет хотеться, чтобы программа работала с человеческой скоростью, чего, впрочем, можно добиться при помощи некоторых отладчиков. Но довольно часто бывает так, что гораздо быстрее получается добиться результата при помощи вставки функции `print` в правильные места программы, чем настройка отладчика, вставка в программу точек остановок и затем пошаговое выполнение программы в том месте, где происходит ошибка.

A.3.1 Моя программа не работает

Вы должны задать сами себе следующие вопросы:

- Есть ли что-то такое, что программа должна выполнять, но не выполняет? Найдите тот раздел в коде, который выполняет эту функцию, и убедитесь, что тот код действительно выполняется так, как вы об этом думаете.
- Происходит ли что-то, что не должно происходить? Найдите код в программе, который выполняет ту функцию, и посмотрите, выполняется ли он.
- Имеется ли в вашем коде такой раздел, который оказывает эффект, который вы не ожидали увидеть? Убедитесь, что вы понимаете код, оказывающий данный эффект, особенно когда он касается вызовов функций или методов из других модулей. Прочтите документацию к используемой функции. Попробуйте ее в работе на небольшом примере и проверьте результаты.

Для того, чтобы создавать программы, вам необходимо иметь в голове модель того, как работает программа. Если вы пишите программу, которая не делает того, что вы от нее ожидаете, то часто проблема заключается не в самой программе, а в той модели, которая находится у вас в голове.

Лучшим способом скорректировать модель является разбить программу на отдельные компоненты (как правило, функции и методы) и протестировать их по отдельности. Как только вы обнаружите несоответствие между вашей моделью и реальностью, вы сможете решить проблему.

Естественно, создавать и отлаживать компоненты вы должны по мере того, как вы пишите программу. Если вы столкнетесь с проблемой, то у вас будет лишь небольшое количество нового кода, нуждающегося в исправлении.

A.3.2 Мое очень сложное выражение не делает то, что нужно

В создании сложных выражений нет никаких проблем до тех пор, пока вы можете их читать. Но они могут быть сложны в отладке. Часто бывает полезно разбить одно сложное выражение на несколько операций присваивания временным переменным.

Например:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

То же самое можно переписать следующим образом:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

Вторую версию легче читать, т.к. имена переменных предоставляют дополнительную информацию, а также она легче в отладке, т.к. вы можете проверить типы промежуточных переменных и отобразить их значения.

Другая проблема длинных выражений может заключаться в том, что порядок выполнения действий может не соответствовать ожидаемому. Например, если вы переводите выражение на язык Python, вы можете написать следующее:

```
y = x / 2 * math.pi
```

Но это неверно, т.к. у умножения и деления одинаковый приоритет, и они выполняются слева направо. Поэтому это выражение будет вычислять .

Если вы сомневаетесь, то вы всегда можете добавить скобки, чтобы явно установить порядок вычислений:

```
y = x / (2 * math.pi)
```

В таком случае не только программа будет работать корректно (в смысле, что она будет делать то, что от нее ожидается), но ее будет также проще читать тем людям, кто еще вы выучил наизусть все правила, регулирующие приоритеты выполнения вычислений.

A.3.3 Моя функция или метод не возвращают ожидаемое значение

Если у вас имеется команда `return`, после которой идет длинное выражение, то вы не сможете вывести его значение с помощью `print` до того, как функция вернет его. Опять же, вы можете использовать временные переменные. Например, вместо

```
return self.hands[i].removeMatches()
```

вы можете написать

```
count = self.hands[i].removeMatches()
return count
```

Теперь у вас будет возможность посмотреть на значение `count` до того, как функция вернет его.

A.3.4 Я действительно глубоко увяз и мне нужна помощь

Прежде всего отойдите от компьютера на несколько минут. Компьютеры излучают волны, которые воздействуют на мозг и вызывают следующие симптомы:

- Чувство подавленности и гнев
- Суеверные мысли ("компьютер меня ненавидит", "программа работает только тогда, когда я ношу кепку задом наперед")
- Программирование в случайном стиле (попытка написать все возможные программы, а затем выбрать ту, что работает правильно)

Если вы обнаруживаете в себе какие-либо из этих симптомов, то вам пора встать и прогуляться немного. А когда вы успокоитесь, то можно вернуться к вашей программе. Что она делает? Каковы могут быть причины ее поведения? Когда в последний раз у вас была рабочая программа, и что вы сделали потом?

Иногда для нахождения ошибки нужно лишь время. Я часто нахожу ошибки тогда, когда я не нахожусь за компьютером, а мой разум блуждает где-то. Самые лучшие места для обнаружения ошибок это поезда, душ и постель, как раз перед тем, как вы засыпаете.

A.3.5 Нет, все-таки мне нужна помощь

И такое бывает. Даже с лучшими программистами. Иногда вы работаете над программой так долго, что уже не можете видеть ошибку. Здесь может помочь свежий взгляд другого человека.

Перед тем, как вы позовете кого-либо на помощь, убедитесь, что у вас все для этого подготовлено. Ваша программа должна быть настолько простой, насколько это возможно, и вы должны работать с наименьшими данными, вызывающими ошибку. В соответствующих местах у вас должны стоять вызовы

функции `print`, которые должны выдавать исчерпывающую информацию. Вы сами должны достаточно хорошо понимать проблему, чтобы изложить ее кратко.

Когда вы пригласите кого-либо на помощь, убедитесь, что у вас имеется информация, которая ему может потребоваться:

- Если есть сообщение об ошибке, то что это за сообщение и на какую часть программы оно указывает?
- Что было самым последним, что вы делали, до того, как возникла ошибка? Какую строчку кода вы написали в последнюю очередь, или что это был за новый тест, что выявил ошибку?
- Что вы уже пробовали делать до сих пор и что вам удалось выяснить?

Когда вы обнаружите ошибку, подумайте над тем, что бы вы могли сделать, чтобы найти ее быстрее. Когда в следующий раз вы встретитесь с чем-то подобным, вы сможете быстрее отыскать ошибку.

Помните, что ваша цель это не просто заставить программу работать, но и *научиться* тому, как заставить вашу программу работать.