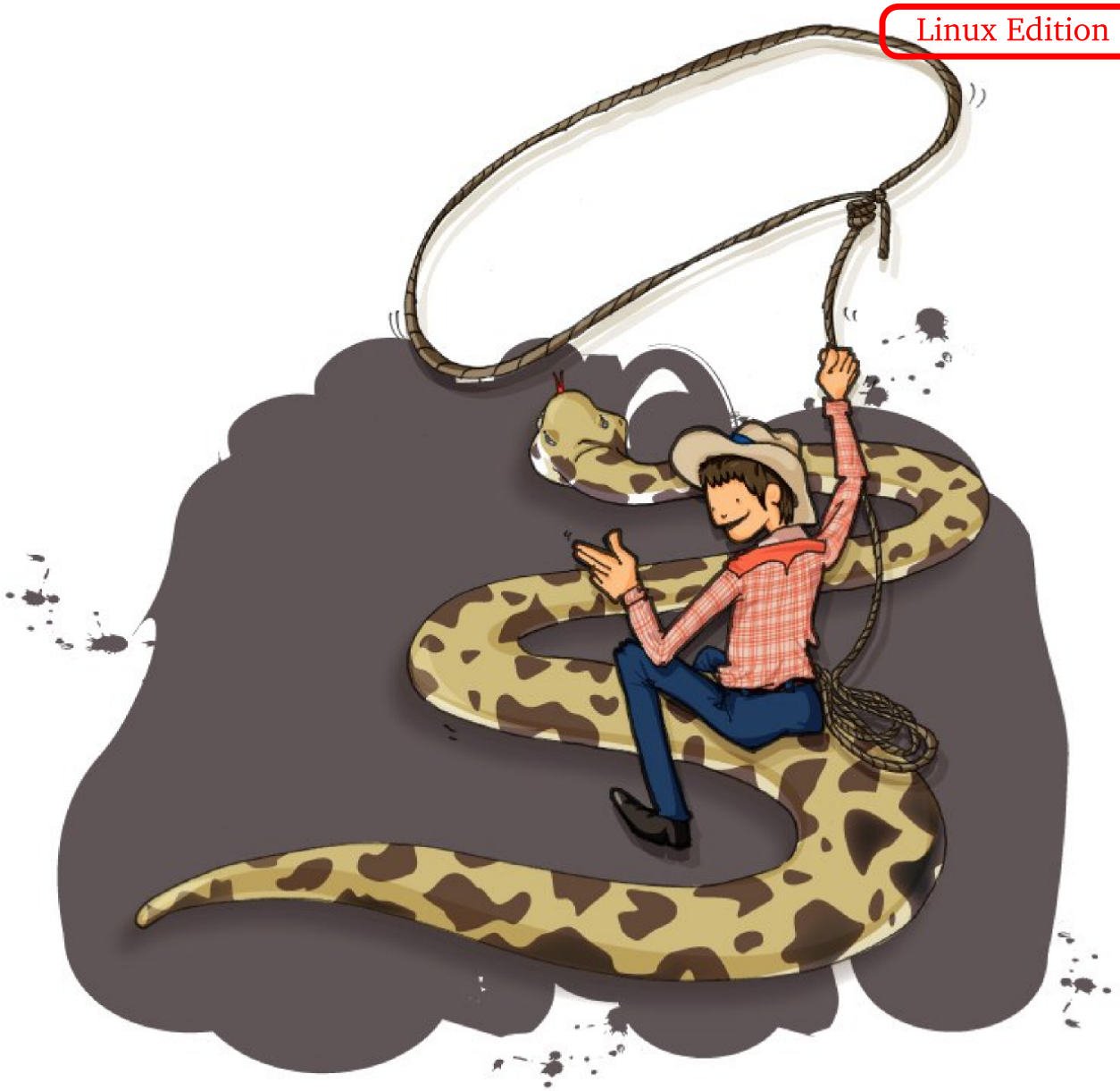
Snake Wrangling For Kids

Learning to Program with Python

Linux Edition



Written by Jason R. Briggs

Snake Wrangling for Kids, Learning to Program with Python
by Jason R. Briggs
перевод на русский язык:
Кочетов Е. М. <Egor.Kochetoff@gmail.com>

Version 0.7.7

Copyright ©2007, 2015

Cover art and illustrations by Nuthapitol C.

This book has been completely rewritten and updated, with new chapters (including developing graphical games), and new code examples. It also includes lots of fun programming puzzles to help cement the learning. Published by No Starch Press - available here: Python for Kids. Also find more info here.

Website:
http://www.briggs.net.nz/log/writing/snake-wrangling-for-kids
Github с переводом книги: `https://github.com/gluk47/swfk/tree/master/ru`

Thanks To:
Guido van Rossum (for benevolent dictatorship of the Python language), the members of the Edu-Sig mailing list (for helpful advice and commentary), author David Brin (the original instigator of this book), Michel Weinachter (for providing better quality versions of the illustrations), and various people for providing feedback and errata, including: Paulo J. S. Silva, Tom Pohl, Janet Lathan, Martin Schimmels, and Mike Cariaso (among others). Anyone left off this list, who shouldn't have been, is entirely due to premature senility on the part of the author.

License:

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 New Zealand License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/nz/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Ниже приведены основные положения лицензии.

Лицензия позволяет:

- **Делиться** — копировать, распространять и передавать эту работу,
- **Изменять** — адаптировать эту работу.

На следующих условиях:

Attribution. Обязательно явно указать авторство этой работы таким способом, как этого просит автор (и так, чтобы не казалось, что автор прямо поддерживает вас или вашу работу, основанную на его).

Noncommercial. Нельзя использовать эту работу и основанные на ней в коммерческих целях.

Share Alike. Если вы изменяете, перерабатываете, адаптируете и распространяете эту работу, обязательно распространять её на условиях этой же лицензии.

При любом использовании и распространении необходимо донести до получателей условия лицензии на эту работу.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.

Оглавление

Вступление

Пара слов для родителей...

Уважаемый родитель или иной управляющий компьютером!

Чтобы ваш ребёнок смог начать знакомиться с программированием, вам нужно установить Python на компьютер. Эта книга была недавно обновлена до версии Python 3.0, самой новой и несовместимой с предыдущими, так что если у вас установлена более старая версия Pyhton, вам стоит скачать и более старую версию этой книги.

Установка Python — достаточно простая задача, но есть несколько тонкостей — в зависимости от используемой операционной системы. Если вы только что купили сверкающий новый компьютер и не имеете никаких идей, что с ним делать, а предыдущее предложение начало вызывать у вас нервную дрожь или холодный пот, то, пожалуй, лучше вам найти кого-то, кто сделает это за вас. Установка Python может занять от 15 минут до пары часов в зависимости от скорости интернета и компьютера.

Прежде всего, скачайте и установите последнюю версию Python 3 для вашего дистрибутива. Дистрибутивов очень много, так что инструкции для всех тут привести не получится... да и скорее всего, если вы используете Linux, то уже знаете, как это сделать. Наверное, вы даже возмущены самой идеей того чтобы рассказывать вам, как что-либо устанавливать, так что тут я остановлюсь.

После установки...

...Вам, возможно, придётся в течение первых пары глав посидеть с ребёнком рядом, но после нескольких примеров ему будет только приятнее читать книгу самому (с компьютером вместе). Нужно рассказать ребёнку, как вводить команды в консоль, как пользоваться текстовым редактором (наподобие блокнота; Microsoft Word никак не подойдёт), открывать и сохранять файлы в этом редакторе. Всё остальное расскажет эта книга.

Спасибо за уделённое время; с наилучшими пожеланиями,
КНИГА.

Глава 1

Не все змеи будут шипеть на тебя

Возможно, тебе подарили эту книгу на день рожденья. А может, на рождество. Например, так: тётя Агата (у всех есть тётя Агата, но не все об этом знают) хотела подарить носки, хотя и не парные, но оба красивые, на два размера больше — на вырост (и всё равно бы эти носки не пригодились потом). А потом вместо этого услышала про эту книгу (которую можно взять и напечатать), вспомнила твои вопросы про всякие компьютерные штуки, и твои непонятные объяснения, как пользоваться компьютером, оборвавшиеся в момент, когда она начала разговаривать с компьютерной мышью, и решила подарить эту книгу. Во всяком случае эта книга уж точно лучше пары разных носков.

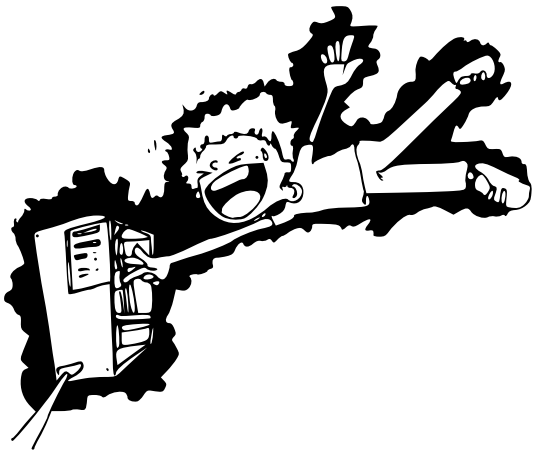
Надеюсь, я не слишком тебя разочаровываю тем, что я — возможно, напечатанная на какой-нибудь старой обёрточной бумаге (хотя если повезло, то и нет) — не слишком разговорчивая (прямо, скажем, совсем молчаливая) книга, с пугающим словом «изучение» в названии... Но представь на минутку и мои ощущения. Если бы вот ты был персонажем из какой-нибудь книги про волшебников, одна из которых наверняка есть у тебя в спальне на книжной полке, — у меня бы могли быть зубы... или даже глаза! А ещё какие-нибудь движущиеся картинки, таинственные звуки... ладно, чего я. В общем, я просто бумажная книжка, хотя могло бы быть и лучше.

Ах, много бы я дала за пару хороших острых челюстей...

Но вообще, быть конкретно такой книжкой тоже не слишком печально. Ну не могу я говорить... пальцы покусывать не могу; зато могу рассказать немного о том, что заставляет компьютеры работать. Не про разные аппаратные штуки — все эти провода, платы, чипы — они меня немного пугают. Электричеством, например, могут ударить (так что не стоит и пытаться ту-

да лезть, как по мне). Я могу рассказать о том, что удивительным образом скрыто внутри всех этих проводов, микросхем и что делает компьютер по-настоящему полезным.

Вообще, это здорово похоже на мысли, например, в твоей голове. Если бы мыслей у тебя не было — сидел бы ты, скажем, на полу в спальне и бессмысленно смотрел в пространство перед собой. Без *программ* компьютеры бы могли приносить пользу, пожалуй, разве что как стопор для двери. Да и то посредственный: вечно все бы об него спотыкались по ночам. А что может быть хуже, чем удариться ночью в темноте пальцем ноги с размаху о железный угол...



Итак, я всего лишь книга. И мне это хорошо известно.

У тебя в семье могут быть разные устройства вроде Playstation, Xbox, Wii — игровые консоли, — а ещё DVD-проигрыватель, может, даже современный холодильник и игрушечная машинка. В них во всех есть программы, которые делают их намного полезнее, чем если бы эти штуки были без программ. В DVD-проигрывателе есть программа для чтения и воспроизведения дисков. В холодильнике — какая-то простая программа для поддержания температуры при минимальных затратах электричества. В машинке — программа для приёма команд с пульта управления и для езды в ответ на эти команды. А в настоящих машинах программы показывают маршруты в объезд пробок и сигналят водителю, когда он паркуется, чтоб он никуда не въехал (в стену или соседнюю машину).

Зная, как писать программы, ты сможешь сделать множество самых разных полезных вещей. Можно свою игру написать. Можно писать страницы в интернете, которые что-нибудь делают, а не просто показывают текст и картинки. Можно упрощать себе выполнение домашней работы.

Так вот, пора приступить к чему-то чуть более интересному, чем эти рассуждения.

1.1. Пара слов про язык

Так же как и у людей, определённо как у китов, возможно, и у дельфинов и, возможно, у родителей (тут, конечно, спорно), у компьютеров есть свой собственный язык. Вообще, как и у людей, у компьютеров много языков. Какую букву английского алфавита ни возьми, она называет какой-нибудь язык. Вот, например, буквы A, B, C, D, E — не только буквы, но и названия языков программирования (что ещё раз доказывает, что у взрослых никакого воображения и хорошо бы им давать почитать хотя бы словарь перед тем, как названия придумывать).

Есть ещё языки программирования, названные в честь людей[1], есть языки-сокращения из заглавных букв (SQL, например), есть немножко названных в честь телевизионных шоу. А, да, ещё если дописать к этим буквам всяких значков типа плюсиков, решёточек (+, #), то тоже получатся названия разных языков программирования. И ещё впридачу некоторые языки очень похожи и отличаются только в каких-то мелочах.

Что я говорила? Никакого воображения!

К счастью, многие из языков уже почти не используются или совсем исчезли; но однако же список способов «говорить» с компьютером всё ещё пугающе велик. Я буду обсуждать только один из них, потому что иначе всё так и закончится их перечислением, не успеем мы приступить к чему-то действительно интересному.

1.2. Орден Неядовитых Удушающих Змей...

... или просто питонов.

Вообще, питон — не только змея[2], но и язык программирования. Многие называют его «пайтон», как принято за рубежом, где его и придумали (и пишут его название как Python). Язык, правда, был назван не в честь змеи — это один из немногих языков программирования, названных в честь телевизионного шоу. Монти Пайтон (Monty Python) — британское телешоу, популярное с 1970х годов. Требуется достичь некоторого возраста и иметь определённый склад ума, чтобы счесть его забавным, но многим нравится. Хотя лет до 12 смотреть вообще смысла нет, будет скучно и непонятно.

Есть несколько особенностей Питона (языка программирования, а не змеи), делающих его очень полезным, чтобы учиться программировать. Для нас сейчас важнее всего то, что используя его, можно быстро сесть и начать писать

[1] например, язык Ада — в честь Ады Лавлейс. А есть ещё язык РАЯ.

[2] которая может не есть полтора года кряду!

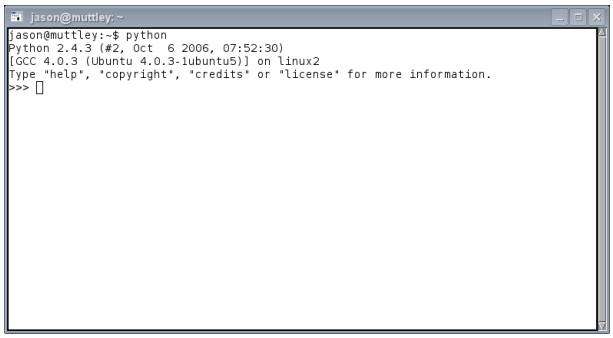


Рис. 1.1: Консоль Питона в Линуксе.

какие-то программки, пусть и очень простые, без долгих разговоров и объяснений.

Это тот момент, когда надо убедиться, что твоя мама, папа или кто там управляет компьютером прочли часть «Заметка для мам и пап». Есть хороший способ проверить это. Спроси у них, как называется программа-*эмулятор терминала* — это может быть `yakuake`, `konsole`, `rxvt`, `xterm` или ещё какая-нибудь, их очень много бывает — именно поэтому придётся спросить. Запусти эту программу, напиши *в командной строке* «python3» (без кавычек) и нажми на клавиатуре клавишу Enter. На экране должно появиться что-то похожее на рисунок 1.1.

Если ты обнаружишь, что они не прочитали инструкции в начале книги...

... и из-за этого у тебя не получилось что-то сделать, то перелистни эту книгу на начало, подсунь им под нос введение, пока они читают утреннюю газету, и умоляюще посмотри на них. Иногда помогает говорить «пожалуйста-пожалуйста-пожалуйста» до тех пор, пока они не встанут и не сделают всё, что надо. Ну и конечно, можно попробовать сделать всё самостоятельно, это может оказаться даже проще.

1.3. Первая программа на Питоне

Так или иначе, если ты добрался досюда, у тебя уже открыта *консоль*, или командная строка Питона — это один из способов запускать команды и целые программы на Питоне. После запуска консоли или ввода любой команды ты увидишь *приглашение командной строки*, которое в Питоне выглядит вот так:

```
>>>
```

Если записать несколько команд на Питоне одну за другой, получится программа, которую можно запускать и не через консоль, но пока на минутку остановимся на простых командах, которые можно вводить прямиком в командную строку (после «приглашения»). Например, можно ввести туда следующую команду:

```
print(''Всем привет!'')
```

Чтобы всё получилось, нужно ввести и скобки и кавычки (вот эти: "") так, как написано выше. Тогда на экране должно появиться что-то вроде такого:

```
>>> print(''Всем привет!'')
Всем привет!
```

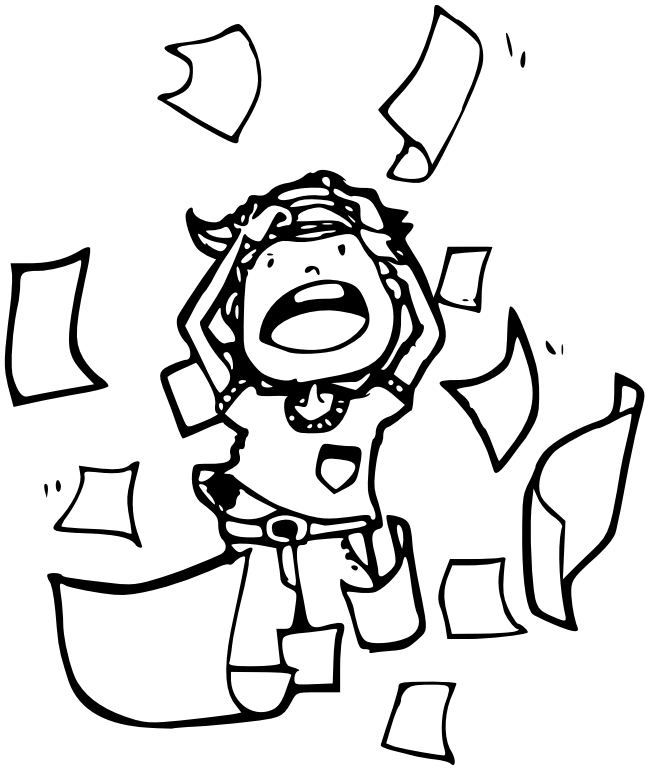
После этого приглашение командной строки появится снова, чтобы показать, что Питон готов принимать новые команды. Поздравляю! Ты только что создал и запустил свою первую программу на Питоне — пусть пока и всего из одной команды: print — функции, которая просто печатает всё, что написано в скобках. Потом мы много будет использовать эту команду.

1.4. Вторая программа на питоне... опять то же самое?

Программы на Питоне были бы не слишком полезными, если бы их приходилось каждый раз вводить заново в командную строку или если бы ты написал программу для кого-то, а ему бы пришлось её перепечатывать, чтобы запустить.

Программа для редактирования текстов (Microsoft Word, Libreoffice Writer или другая подобная), которую ты, вероятно, используешь для выполнения каких-нибудь домашних заданий, получена из исходного кода размером примерно от 10 до 100 миллионов строк. Если печатать это на бумаге с двух сторон не очень крупно, это может занять, например, 400 000 страниц. Это стопка бумаги высотой 40 метров, с десятиэтажный дом. Такое количество бумаги нести из магазина в дом, чтобы перепечатать, пришлось бы долго... очень долго...

...а если бы ещё и ветер подул в подходящий момент... за бумагой пришлось бы долго бегать. Так вот, хорошая новость: всем этим заниматься не обязательно.



Открой текстовый редактор (можешь опять спросить у родителей, как он называется: например, kate, gedit, kdevelop, но никак не Microsoft Word, он не подойдёт) и напиши туда точно ту же самую команду, что ты до этого вводил в консоль:

```
print(''Всем привет!'')
```

Теперь сохрани этот файл в своей домашней папке. Наверху в программе должен быть значок сохранения, а когда тебя спросят, куда сохранять, нажми на какую-нибудь кнопку типа домика. В качестве имени файла введи «hello.py». Теперь опять открой терминал и напиши:

```
python hello.py
```

В консоли должно появиться приветствие от программы, точно так же, как в прошлый раз (примерно как на рисунке 1.2).

Вот. Теперь ты видишь, что мудрые люди, создавшие Питон, спасли тебя от ввода одних и тех же программ много-много-много раз для выполнения одних и тех же действий. Как они делали в 1980х. Я серьёзно, им приходилось вводить каждый раз кучу команд для выполнения одной и той же программы. Можешь спросить у папы — вдруг у него был ZX81 в молодости — так там приходилось так делать. Теперь можно просто написать имя программы, и она целиком исполнится от начала до конца.

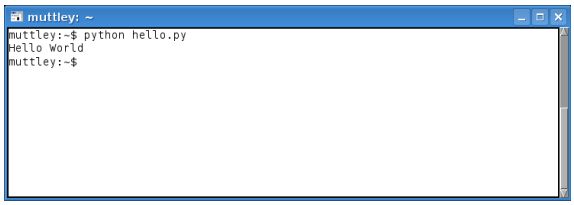


Рис. 1.2: Запуск программы на Питоне, сохранённой в текстовый файл

Конец начала

Добро пожаловать в удивительный мир программирования! Мы начали с простой программы, которая печатает «Всем привет» («Hello world») — все с этого начинают, когда учатся программировать. В следующей главе мы займёмся чуть более полезными вещами в консоли Питона, а потом изучим, как написать программу посложнее.

Глава 2

8 умножить на 3.57 равняется...

Чему равно 8 умножить на 3.57? Пришлось бы использовать калькулятор, чтобы посчитать? Ладно, этот пример можно вычислить и в уме, но не в том дело. То же самое можно сделать в консоли Питона. Запусти её опять (как описано в предыдущей главе), если она ещё не запущена, и введи туда такую команду: `8*3.57`. Потом нажми Enter.

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type ''help'', ''copyright'', ''credits'' or ''license'' for more information.
>>> 8 * 3.57
28.56
```

Звёздочка (*) (обычно это shift + 8) используется для умножения вместо привычного символа × (или ·), потому что не везде их можно просто так ввести с клавиатуры, а буква X путалась бы с собственно буквой, если бы её использовали как знак умножения. Ладно, как насчёт чего-нибудь более полезного?

Представь, что тебе приходится заниматься делами по хозяйству раз в неделю, за что каждый раз ты получаешь по 5 рублей и ещё у тебя есть курьерская подработка по доставке газет за 30 рублей в неделю. Сколько денег такими темпами накопится за год?

Если бы мы решали эту задачу на бумаге, то написали бы что-то вроде:

```
(5 + 30) × 52
```

Что значит: (5 руб + 30 руб), умноженное на 52 недели в году. Можно, конечно, и сразу сократить эту запись до такой:

```
35 × 52
```

И это уже совсем просто посчитать что на калькуляторе, что в столбик (в уме сложнее). А можно всё то же самое сделать в консоли:

Питон сломался!?!?

Если ты возьмёшь калькулятор и введёшь туда `8×3.57`, ответ на экране будет такой:

28.56

В Питоне ответ может быть такой же, а может быть такой:

28.55999999

Чем Питон отличается? Уж не поломан ли он??
Да нет, вообще-то. Просто так дробные числа (числа с десятичной запятой, или *числа с плавающей запятой*) представляются в компьютере: приближённо. Результаты всегда почти точные, но далеко справа после точки могут набегать небольшие ошибки вычисления после выполнения последовательности действий. Дробные числа представляются в компьютере не очень просто, мы не будем сейчас на этом останавливаться. Я хочу сказать, не удивляйся, что *иногда* результаты вычислений не в точности равны тому, что ты ожидаешь; это верно для умножения, деления, сложения и вычитания.
Целые числа в компьютере представляются точно и всегда вычисляются без ошибок в Питоне (в других языках программирования есть свои хитрости).

```
>>> (5 + 30) * 52
1820
>>> 35 * 52
1820
```

Как быть, если ты тратишь 10 рублей в неделю? Что теперь останется к концу года? Можно было бы разными способами записать это на бумаге, но давай опять обратимся к консоли:

```
>>> (5 + 30 - 10) * 52
1300
```

Это значит: 5 рублей и ещё 30 рублей минус 10 потраченных рублей, и всё умножается на 52 недели в году. В конце года будет 1300 накопленных рублей. Ладно, я понимаю, что выглядит это всё не слишком полезно, и без Питона

тут было бы легко обойтись. Мы потом опять вспомним про этот пример, и я покажу, как сделать из него куда более полезный.

В питоньей консоли можно умножать, складывать, вычитать и делить. Можно выполнять и другие арифметические действия, но мы пока их пропустим. Вот так выглядят символы математических операций:

$+$	Сложение
$-$	Вычитание
$*$	Умножение
$/$	Деление

Для деления используется косая черта («прямой слэш» или просто «слэш», как его ещё называют), потому что рисовать дроби с клавиатуры не так-то просто, а символа деления ($\div$) обычно тоже нет.

Если бы тебя попросили сказать, сколько коробок, в которые влезает по 20 яиц, нужно для 100 яиц, ты мог бы написать что-то такое:

$$\frac{100}{20}$$

или такое:

$$\begin{array}{r|l} 100 & 20 \\ \hline 100 & 5 \\ \hline 0 & \end{array}$$

ну или, может, такое (хотя так чаще за рубежом пишут)

$$100 \div 20$$

Так вот, в Питоне надо написать вот так: `100 / 20`.

Что, на мой взгляд, гораздо проще. Впрочем, что я в этом понимаю, я всего лишь книга.

2.1. Использование скобок и «приоритет операций»

В языках программирования (да и просто в математике) мы используем скобки, чтобы управлять тем, что называется «приоритет операций». Операции не ограничиваются четырьмя арифметическими, бывают и другие, но смысл одинаковый. Из этих четырёх операций умножение и деление, как обычно, выполняются перед сложением и вычитанием, то есть у них выше приоритет. В примере ниже у всех операций одинаковый приоритет, поэтому они все выполняются слева направо:

```
>>> print(5 + 30 + 20)
55
```

В примере ниже тоже у всех одинаковый приоритет, и тоже все операции выполняются по порядку слева направо:

```
>>> print(5 + 30 - 20)
15
```

А вот в следующем примере есть умножение. Тут сначала умножаются числа 20 и 30, а потом к ним прибавляется 5: у умножения выше приоритет.

```
>>> print(5 + 30 * 20)
605
```

Что будет, если добавить скобок? Вот что:

```
>>> print((5 + 30) * 20)
700
```

Почему поменялся результат? Потому что скобки меняют порядок операций. Сначала вычисляется то, что в скобках, а потом всё, что снаружи. Тут сначала к 5 прибавляется 30, а результат всего Питон умножает на 20.

Можно использовать скобки более сложным образом. Можно внутри скобок писать ещё скобки:

```
>>> print(((5 + 30) * 20) / 10)
70
```

Тут Питон сначала вычисляет то, что внутри самых вложенных скобок, потом то, что во внешних скобках, и потом выполняет операцию деления, которая вообще не в скобках (самые внешние скобки нужны слову print, а не чтобы указывать порядок операций). Это выражение можно прочитать так: «прибавь к 5 30, потом это всё умножь на 20 и результат подели на 10». Без скобок всё было бы иначе:

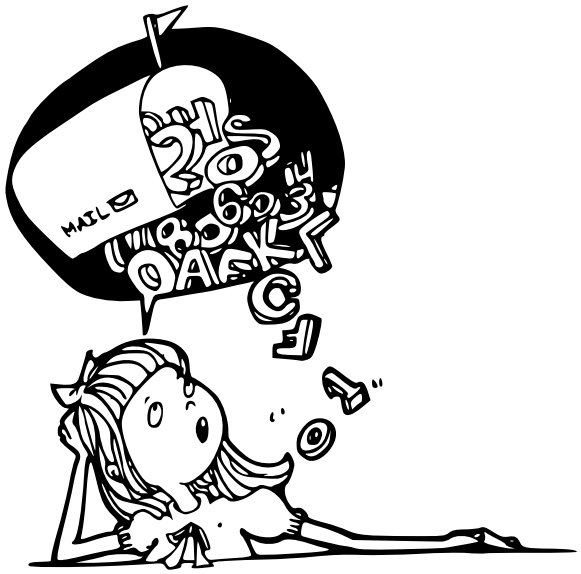
```
>>> 5 + 30 * 20 / 10
65
```

Тут сначала 30 умножается на 20, потом делится на 10 и к этому всему прибавляется 5.

Помни, что умножение и деление всегда выполняются перед сложением и вычитанием, если только скобки не указывают делать иначе.

2.2. Нет ничего столь же непостоянного, как переменная

«Переменная» в программировании — название места для хранения разных штук. «Разные штуки» могут быть числами, текстом, списками чисел или строк и множеством других вещей, о которых обо всех сейчас пока рассказать сложно. Давай пока считать, что переменная — это что-то вроде почтового ящика.



В почтовый ящик можно положить что-нибудь: письмо, газету — и в переменную можно точно так же положить что-нибудь: число, строку, список чисел. Именно таким образом работают переменные в большинстве языков программирования: в них что-нибудь кладут. Правда, не во всех языках так.

В Питоне переменные используются немножко иначе. Они не хранят что-то внутри себя, а, скорее, работают наклейкой на почтовом ящике. Эту наклейку можно снять и переклеить на другой ящик, например. А можно прилепить сразу на несколько каких-нибудь коробок (обмотав их предварительно скотчем). В питоне эта операция прилепливания выглядит как знак равенства. Чтобы создать переменную, нужно написать её имя слева от этого знака, а справа — то, на что эта переменная указывает. Например, вот так:

```
>>> Фёдор = 100
```

Мы сейчас создали переменную, называющуюся `Фёдор`, и сказали, что она указывает на число 100. Тут мы ещё говорим Питону запомнить это число, потому что собираемся его потом использовать. Чтобы узнать, на что переменная указывает, можно вывести её значение функцией `print`, как-то так:

```
>>> Фёдор = 100
>>> print(Фёдор)
100
```

Можно сказать Питону, что `Фёдор` теперь должен указывать на другое число:

```
>>> Фёдор = 200
>>> print(Фёдор)
200
```

Теперь `Фёдор` указывает на число 200 (а число 100, которое больше не нужно, Питон потом забудет, когда оно будет мешать). И `print` подтвердил нам, что Фёдор теперь указывает на 200. Можно наклеить на это же самое 200 и ещё одну наклейку, то есть создать ещё одну переменную, указывающую на него:

```
>>> Фёдор = 200
>>> Василий = Фёдор
>>> print(Василий)
200
```

В этом кусочке кода мы создаём переменную `Василий`, которая указывает туда же, куда и `Фёдор`. Если Фёдор поменяет своё значение, Василия это не коснётся.

Вообще, конечно, «Фёдор» — не самое подходящее имя для переменной. Обычно имена переменных записывают английскими буквами — так программистам со всего мира гораздо проще читать и дополнять программы друг друга. Ну и кроме того, имя переменной должно бы говорить, зачем она нужна. Вот с почтовым ящиком всё понятно, он нужен, чтобы разную почту хранить. А переменная может хранить совершенно разные вещи, совсем по-разному устроенные, и переменных в программе обычно очень много. Поэтому называть переменную нужно так, чтобы сразу было понятно, для чего она используется[1].

Представь себе, что ты открыл консоль Питона, напечатал `Фёдор = 200`, а потом отошёл от компьютера. На 10 лет. Чтобы залезть на Эверест, потом пересечь пустыню Сахару, попрыгать с моста на резинке в Новой Зеландии и спуститься на лодках по реке Амазонке. И потом обратно вернулся к консоли. Самое сложное будет понять: почему Фёдор и почему 200?!

Обычно, чтобы всё забыть, хватает гораздо меньшего времени, чем 10 лет. А часто программу пишут сразу несколько людей, каждый из которых не знает мысли остальных. Так что лучше с самого начала называть все переменные понятно и не рассчитывать на свою память.

[1] Некоторые даже считают, что придумывать хорошие имена переменным — самое сложное в работе программиста.

Так-то! Смотри, например, мы можем назвать переменную так: `number_of_students`.

```
>>> number_of_students = 200
```

Так сделать получится, потому что имена переменных можно составлять из букв, цифр и знаков подчёркивания (главное, чтобы имя не начиналось с цифры). Если ты вернёшься к своему коду через 10 лет, то «number_of_students» всё ещё будет иметь смысл (по крайней мере, если ты знаешь английский хоть немного, что в любом случае пригодится). Можно будет напечатать вот так:

```
>>> print(number_of_students)
200
```

И сразу будет понятно, что количество студентов (где-то) — 200. Не всегда есть смысл придумывать длинные осмысленные имена переменных: иногда переменная используется всего в паре строк, иногда можно использовать вообще одну букву в качестве имени, и есть такие случаи, когда всем это будет привычно и понятно. В основном, это зависит от того, захочется ли потом понять по имени переменной, зачем она вообще нужна, или это будет почему-нибудь понятно и без этого.

```
это_тоже_допустимое_имя_переменной_но_наверное_не_слишком_полезное
```

2.3. Используем переменные

Ладно, как создавать переменные мы разобрались. Но что с ними делать?

А вот что. Помнишь тот пример, который мы считали выше? Сколько денег заработается за год? Вот этот:

```
>>> print((5 + 30 - 10) * 52)
1300
```

Что если нам сюда добавить три переменных? Вот так:

```
>>> chores = 5
>>> paper_round = 30
>>> spending = 10
```

В переменных записано, сколько денег получается за неделю: 5 руб. за домашние дела и 30 — за курьерство — и сколько денег тратится в неделю: 10 рублей. Можно теперь перепечатать тот же пример вот так:

```
>>> print((chores + paper_round - spending) * 52)
1300
```

Ответ такой же. А что если теперь за домашние дела ты будешь получать семь рублей вместо пяти? Надо записать в переменную `chores` значение 7, а потом нажать пару раз кнопку вверх на клавиатуре, чтобы появился опять пример на экране, и Enter, вот так:

```
>>> chores = 7
>>> print((chores + paper_round - spending) * 52)
1404
```

Так пришлось гораздо меньше печатать, чтобы получить новый ответ, чем без использования переменных. Можно попробовать поменять другие переменные и опять нажимать клавишу вверх, чтобы получать новые ответы:

```
Что будет если тратить вдвое больше:
>>> spending = 20
>>> print((chores + paper_round - spending) * 52)
884
```

Если тратить 20 рублей в неделю, то к концу года получится накопить только 884 рубля. Пока переменные не сильно много пользы нам принесли, но уже понятно, что с ними проще и что они могут что-нибудь хранить.

Вспоминай почтовый ящик с наклейкой на нём!

2.4. Кусочек строки

Если ты уделяешь внимание моим словам, а не просто бегло проглядываешь книжку в поисках интересных картинок, то можешь вспомнить, что я говорила, что в переменных можно хранить разного сорта вещи, а не только числа. Например, можно хранить текст, или, как его обычно называют в программировании, строки (по-английски: *string*). Такое название может поначалу показаться странным, но, если подумать, текст — это просто буквы, составленные в слова, составленные в строки, и ничего больше. Может быть, так понятнее.

А может, и не понятнее...

В таком случае, всё, что тебе необходимо знать, — это что строка — не более чем составленные вместе буквы, цифры и любые другие символы. Ну и обычно они что-то значат все вместе. Всё содержимое этой книги можно записать в строку. Твоё имя можно записать строкой. Твой домашний адрес

тоже. Мы, кстати, уже использовали строку в первой программе, вот эту: ”Всем привет!”.

В Питоне строка создаётся просто: нужно поставить кавычки (такие: ” или такие: ’) вокруг текста. Можно вспомнить про нашу бесполезную переменную Фёдора и записать в неё строку, вот так:

```
>>> Фёдор = ’’я маленькая строка’’
```

И как всегда, можно узнать, что хранит Фёдор:

```
>>> print(Фёдор)
я маленькая строка
```

Или вот пример с одинарными кавычками:

```
>>> Фёдор = ’и я строка’
>>> print(Фёдор)
и я строка
```

Но если ты попытаешься сохранить в переменную сразу несколько строк текста[1], используя просто кавычки, то ничего не выйдет. Например, попробуй напечатать команду, написанную ниже, и получишь пугающе непонятное сообщение об ошибке, что-то вроде такого:

```
>>> Фёдор = ’’первая строка
  File ’’<stdin>’’, line 1
    Фёдор = ’’первая строка
                           ^
SyntaxError: EOL while scanning string literal
```

Про эту ошибку мы поговорим позже, а пока можешь просто запомнить, что чтобы сохранить в переменную сразу несколько строк текста, нужно использовать три одинарных кавычки[2]:

```
>>> Фёдор = ’’’вот строка
... с переносом строки внутри’’’
```

Напечатай эту переменную, чтобы проверить, что всё получилось:

[1] По-английски это называется *multilne string*, то есть «многострочная строка». Строка с переносами строк внутри, иначе говоря.

[2] Обычно одинарную кавычку можно ввести только в английской раскладке и она находится на той же клавише, где русская «Э», справа→ на клавиатуре.

```
>>> print (Фёдор)
вот строка
с переносом строки внутри
```

Пока ты печатал эту строчку с переносом, то мог обратить внимание, что Питон вместо приглашения на второй строке напечатал три точки (...). Он это делает в тех случаях, когда что-то, начатое на первой строке, продолжается на следующей — вот как строка. Пока кавычки не закроются, Питон так и будет печатать «...», показывая, что ждёт завершения строки. Потом мы и в других случаях увидим эти три точки.

2.5. Развлечения со строками

Вот есть интересный вопрос: чему равно 10 * 5 (10 умножить на 5)? Ответ, конечно: 50.

Ладно, это был неинтересный вопрос.

А чему равно 10 * 'a' (10 умножить на букву а)? Вопрос может показаться бессмысленным, но вот ответ от Питона:

```
>>> print(10 * 'a')
aaaaaaaaaa
```

Это работает не только с отдельными символами, но и со строками подлиннее:

```
>>> print(20 * 'abcd')
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

Есть ещё один трюк со строками. В них можно вставлять значения. Для этого в строку нужно добавить «%s» как маркер, куда вставлять значение. Проще всего это показать на примере:

```
>>> mytext = 'Мне %s лет'
>>> print(mytext % 12)
Мне 12 лет
```

В первой строке создаётся переменная `mytext`, содержащая некоторые слова и шаблон подстановки (в английских инструкциях он называется *placeholder*) `%s`. Этот шаблон — это такая инструкция Питону: «замени меня чем-нибудь». На следующей строке, когда мы печатаем значение переменной `mytext`, мы используем символ `%`, чтобы указать Питону, на что именно заменять шаблон подстановки: на число 12. Можно использовать одну и ту же строку и подставлять внутрь неё разные значения:

```
>>> mytext = 'Привет, %s, как твои дела?'
>>> name1 = 'Иннокентий'
>>> name2 = 'Саша'
>>> print(mytext % name1)
Привет, Иннокентий, как твои дела?
>>> print(mytext % name2)
Привет, Саша, как твои дела?
```

В примере выше создаются три переменных (`mytext`, `name1` и `name2`), первая из которых содержит маркер «`%s`». Потом мы печатаем значение этой строки, подставляя внуть неё значения строк `name1` и `name2`, используя операцию «`%`».

Можно вставить в строку больше, чем один шаблон для подстановки, вот так:

```
>>> mytext = 'Привет, %s и %s, как обстоят ваши дела?'
>>> print(mytext % (name1, name2))
Привет, Иннокентий и Саша, как обстоят ваши дела?
```

Когда используется несколько маркеров, нужно подставляемые переменные окружать скобками, как в примере выше. Несколько переменных, окружённых скобками, называются *кортеж* (по английски — *tuple*). Это слово потом ещё будет использоваться.

2.6. Не совсем список для покупок

Яйца, молоко, мыр, сельдерей, масло и сода. Не вполне полный список для покупок, но сойдёт для наших целей. Если бы тебе захотелось сохранить это в переменной, можно было бы это сделать так, в строку:

```
>>> shopping_list = 'яйца, молоко, сыр, сельдерей, масло, сода'
>>> print(shopping_list)
яйца, молоко, сыр, сельдерей, масло, сода
```

Другой способ — создать «список», особый сорт объектов в Питоне:

```
>>> shopping_list = [ 'яйца', 'молоко', 'сыр', 'сельдерей', 'масло', 'сода' ]
>>> print(shopping_list)
[ 'яйца', 'молоко', 'сыр', 'сельдерей', 'масло', 'сода' ]
```

Чтобы сделать список, приходится больше печатать (все эти кавычки), но результат сильно полезнее строки. Например, мы можем просто напечатать третий элемент списка, указав его *индекс в списке* (позицию) в квадратных скобках:

```
>>> print(shopping_list[2])
сыр
```

Да, здесь нет опечатки. Мы напечатали третий элемент, хотя в квадратных скобках стоит цифра 2. Всё потому, что номера элементов в списке считаются с нуля. Это достаточно непривычно и кажется бессмысленным большинству людей — но не программистам[1]. Скоро ты тоже привыкнешь считать с нуля вместо одного, чем тоже будешь удивлять многих людей.

Можно ещё напечатать все элементы с третьего по пятый, используя двоеточие внутри квадратных скобок:

```
>>> print(shopping_list[2:5])
['сыр', 'сельдерей', 'масло']
```

`[2:5]` — значит: нужны все элементы с индексами от 2 до 5 (не включая 5). Поскольку индексы считаются с нуля, то получатся как раз третий, четвёртый и пятый элементы, а шестой (который с индексом 5) не будет включён в результат.

В списках можно хранить (как и в переменных) что угодно. Можно хранить числа:

```
>>> mylist = [ 1, 2, 5, 10, 20 ]
```

...и строки:

```
>>> mylist = [ 'a', 'bbb', 'ccccccc', 'ddddddddd' ]
```

...и всё вместе:

```
>>> mylist = [1, 2, 'a', 'bbb']
>>> print(mylist)
[1, 2, 'a', 'bbb']
```

...и другие списки:

```
>>> list1 = [ 'a', 'b', 'c' ]
>>> list2 = [ 1, 2, 3 ]
>>> mylist = [ list1, list2 ]
>>> print(mylist)
[['a', 'b', 'c'], [1, 2, 3]]
```

В примере выше создаётся перменная `list1`, содержащая список из трёх букв, и `list2`, содержащая список из трёх цифр; а потом создаётся переменная `mylist`, которая содержит оба эти списка. Дальше можно сходить с ума,

[1] Индекс элемента в списке просто показывает, сколько элементов надо пропустить от начала списка, чтобы добраться до нужного. Индекс 0 значит, что пропускать не надо ни одного, индекс 2 — что надо пропустить 2 элемента, и получить как раз третий.

создавая списки списков списков списков... но обычно никому не нужно делать такие сложные вещи в Питоне. С другой стороны, полезно знать, что так вообще можно и что список может хранить всё что угодно.

А не только какие продукты надо купить.

Изменяем элементы

Можно изменить элемент списка, просто присвоив ему новое значение, как будто это обычная переменная. Можно, например, заменить сельдерей салатом:

```
>>> shopping_list[3] = 'салат'
>>> print(shopping_list)
['яйца', 'молоко', 'сыр', 'салат', 'масло', 'сода']
```

Добавляем ещё элементы...

Можно добавить к списку новых элементов, вызвав функцию `append`. Подробно про функции (и как делать свои) мы поговорим позже, но пока можно воспользоваться готовой вот так:

```
>>> shopping_list.append('шоколадка')
>>> print(shopping_list)
['яйца', 'молоко', 'сыр', 'салат', 'масло', 'сода', 'шоколадка']
```

И этот список, несомненно, гораздо лучше оригинальноого.

...и удаляем элементы

Можно удалить из списка элементы, используя функцию `remove`. Например, чтобы удалить соду из списка, надо написать вот так:

```
>>> shopping_list.remove ('сода')
>>> print shopping_list
['яйца', 'молоко', 'сыр', 'салат', 'масло', 'шоколадка']
```

Ещё можно удалить элемент списка по ему индексу (как, кстати, и любую другую переменную). Для этого есть команда `del` (сокращение от слова `delete`, удалить), которая используется вот как:

```
>>> del shopping_list[0]
>>> print(shopping_list)
['молоко', 'сыр', 'салат', 'масло', 'шоколадка']
```

Теперь из списка исчез первый элемент — тот, который с индексом 0.

Два списка лучше одного!

Можно объединить два списка, сложив их, как будто мы складываем числа:

```
>>> list1 = [ 1, 2, 3 ]
>>> list2 = [ 4, 5, 6 ]
>>> print(list1 + list2)
[1, 2, 3, 4, 5, 6]
```

Можно записать результат сложения списков в новую переменную:

```
>>> list1 = [ 1, 2, 3 ]
>>> list2 = [ 4, 5, 6 ]
>>> list3 = list1 + list2
>>> print(list3)
[1, 2, 3, 4, 5, 6]
```

Кстати, этот метод можно использовать, чтобы добавлять в список новые значения вместо использования функции `append`. Вот, например, так:

```
>>> list1 = [ 1, 2, 3 ]
>>> list1 = list1 + [ 4 ]
>>> print (list1)
[1, 2, 3, 4]
>>> list1 = list1 + [5, 6, 7]
>>> print (list1)
[1, 2, 3, 4, 5, 6, 7]
```

Ещё список можно умножать примерно так же, как мы до этого умножали строку:

```
>>> list1 = [ 1, 2 ]
>>> print(list1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

В примере выше умножение списка на 5 — это инструкция Питону «повтори список 5 раз.

Деление и вычитание для списка не имеют смысла, поэтому если попытаться так сделать, получится только ошибка:

```
>>> list1 / 20
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

или вот:

```
>>> list1 - 20
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
TypeError: unsupported operand type(s) for -: 'type' and 'int'
```

И попробуй угадай, что Питон хочет сказать этими сообщениями об ошибке...

2.7. Кортежи и списки

Кортеж (про который мы говорили раньше) вроде как список, но чтобы сделать его, нужно использовать круглые скобки вместо квадратных. Пользоваться кортежами можно примерно так же, как списками:

```
>>> t = (1, 2, 3)
>>> print(t[1])
2
```

Основное отличие от кортежей списков — это что кортежи нельзя изменять. Если попытаться так сделать, то просто получится ещё одно сообщение об ошибке:

```
>>> t[0] = 4
Traceback (most recent call last):
  File ''<stdin>'', line 1, in ?
TypeError: 'tuple' object does not support item assignment
```

Это, конечно, не значит, что если переменная хранила кортеж, то в неё нельзя записать ничего больше. Нельзя именно поменять тот кортеж, на который она указывает, но можно вместо него записать новый.

Например, вот этот код будет работать без проблем:

```
>>> myvar = (1, 2, 3)
>>> myvar = [ 'список', 'строк' ]
```

Тут мы вначале создаём переменную `myvar`, указывающую на кортеж из трёх чисел. Потом говорим Питону, что теперь эта переменная указывает на список строк. Это может показаться немного странным — зачем вообще нужны кортежи и как именно они отличаются от списков. Тут можно представить шкафчики для вещей в бассейне, только с прозрачной дверцей. Можно положить вещи в шкафчик и выбросить ключ. Тогда нельзя будет поменять, что именно лежит в шкафчике (хотя посмотреть на них всё-таки получится), и

шкафчик будет кортежем. А можно не выкидывать ключ, тогда шкафчик будет как список: всегда можно будет что-то туда положить или вынуть. На все шкафчики можно приклеивать и отлеплять метки — это наши переменные.

Кортежи иногда бывают необходимы, когда списки не подходят именно из-за того, что списки можно менять. Мы потом столкнёмся с такими ситуациями.

2.8. Как ещё развлечься

В этой главе мы увидели, как вычислять простые математические примеры в консоли Питона. Потом мы ещё обсудили, как использовать скобки для изменения результатов вычисления, чтобы менять приоритеты операций. Ещё мы научились говорить Питону запоминать значения для дальнейшего использования — при помощи переменных — и узнали, что Питон использует «строки» для хранения текста, а ещё списки и кортежи, в которых можно хранить несколько разных значений сразу.

Упражнение 1

Создай список своих любимых игрушек `toys` и список своей любимой еды `foods`. Объедини списки и сохрани результат в переменную `favourites`, а затем напечатай на экран её значение.

Упражнение 2

Если у тебя есть 3 коробки по 25 шоколадок и ещё 10 коробок по 32 конфеты, сколько у тебя всего сладостей в штуках? Как можно добавить сюда переменных и быстро узнать, сколько будет сладостей, если в каждую из коробок с конфетами положить по 35 конфет вместо 32?

Упражнение 3

Запиши в переменные свои имя и фамилию. Потом создай строку с шаблонами подстановки, подставь в неё имя и фамилию (операцией %) и выведи результат на экран.

Глава 3

Черепахи и другие медленные создания

Есть нечто общее между черепахами из реального мира и черепахой в Питоне. В реальном мире черепаха — это (иногда) зелёная рептилия, которая движется очень медленно и носит на себе свой дом. В мире Питона всё почти так же: черепаха — это маленькая чёрная стрелочка, которая очень медленно ползает по экрану. Правда, с домиком у этой стрелочки не сложилось.

Вообще, черепашка в Питоне оставляет за собой след, так что она похожа больше на улитку, чем на черепаху. Но «черепаха» звучит более гордо, так что модуль в Питоне называется именно так. Можно представлять себе черепаху, зажавшую в зубах маркер и рисующую, пока ползёт.

Давным-давно, в тёмные старые времена люди придумали язык программирования Лого (Logo). Это был язык для управления роботом-черепашкой Ирвингом. Со временем черепашка превратилась из робота, ползающего по полу, в маленькую стрелочку, перемещающуюся по экрану.

Что, кстати, показывает нам, что с развитием технологии не всегда всё улучшается. Маленькая робот-черепашка на полу была бы куда как забавнее.

В Питоне есть модуль[2] `turtle` (черепаха), и он в целом похож на язык Лого. Только Лого ничего, кроме черепашки, и не умеет, а Питон умеет ещё много всего другого. Модуль `turtle` полезен, чтобы понять, как компьютер рисует изображение на экране.

Ладно, давай теперь просто посмотрим, как же этот модуль работает. Во-первых, его надо «импортировать», то есть сказать Питону, что он нам нужен:

```
>>> import turtle
```

Потом нам надо создать «холст» для рисования — смысл тот же, что и в

[2]Про модули подробно поговорим мы чуть позже, а пока просто стоит иметь в виду, что модуль — это что-то, что можно использовать в своей программе, набор разных функций.

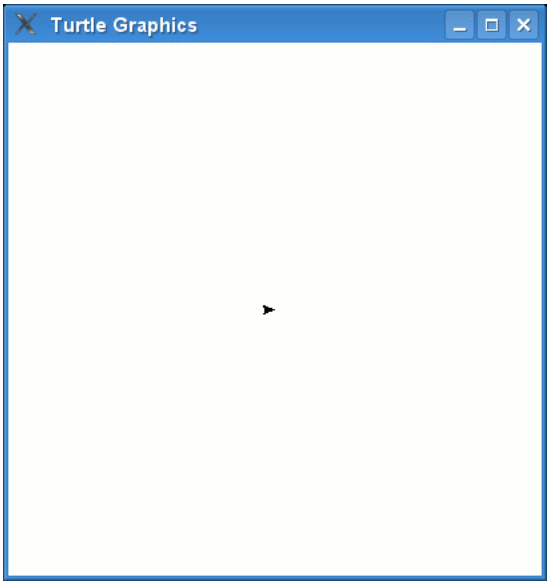


Рис. 3.1: Стрелочка, изображающая черепаху

реальном холсте, которым пользуются художники. Холст будет нужен, чтобы на нём рисовать. Мы создадим пустой:

```
>>> t = turtle.Pen()
```

Тут мы вызываем функцию `Pen` модуля `turtle`, и она автоматически создаёт холст (*canvas* по-английски). Вообще, функция — это что-то вроде маленькой программы, то есть кусочек кода, который можно использовать много раз (подробно функции мы обсудим потом). В данном случае функция `Pen` *возвращает* нам черепашку, то есть результат этой функции — черепашка, к которой мы приклеиваем переменную `t`. Результатом этого кода должна быть картинка наподобие 3.1.

Да, эта маленькая стрелочке посреди экрана — действительно черепаха. И да, на черепаху она внешне не похожа примерно ничем.

Этой черепахе можно отправлять инструкции, используя функции созданного объекта `t`. Вот, например, можно попросить черепаху продвинуться вперёд, туда, куда показывает стрелочка. Для этого есть функция `forward`, в скобках которой надо указать, на сколько точек на экране продвинуться. Например, чтобы подвинуть черепаху вперёд на 50 точек (и нарисовать линию длиной 50 точек), нужно выполнить такую команду:

```
>>> t.forward(50)
```

И в результате должно получиться как-то так: 3.2.

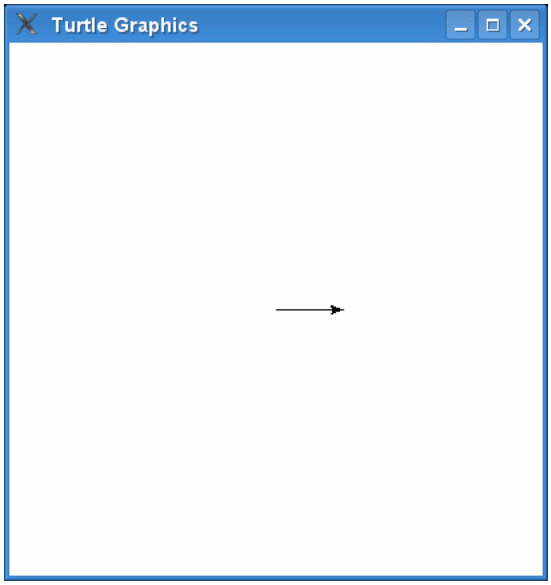


Рис. 3.2: Черепашка нарисовала линию.

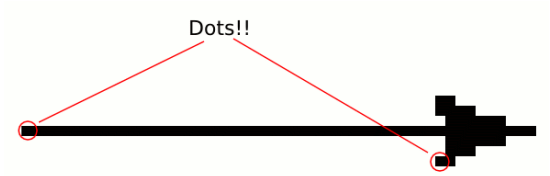


Рис. 3.3: Сильно увеличенные линия и стрелочка.

С точки зрения черепахи, она прошла вперёд 50 шагов. А мы бы сказали, что она прошла 50 точек по экрану.

И что это за точки такие?

Всё на экране компьютера состоит из отдельных маленьких точек, каждая из которых окрашена в свой цвет. Обычно их называют пикселями, чтобы не путать ни с какими другими точками, и так я и буду дальше делать. Все программы на компьютере, все игры заставляют точки на экране окрашиваться в разные нужные цвета. Эти отдельные точки можно увидеть, вооружившись лупой. Если сильно увеличить линию, нарисованную черепахой, то можно увидеть, что это много квадратных точек, оказавшихся рядом, как на картинке 3.3.

В следующих главах мы ещё вспомним про пиксели, они нам пригодятся.

Вернёмся к черепашке. Её ещё можно попросить повернуть направо и налево.

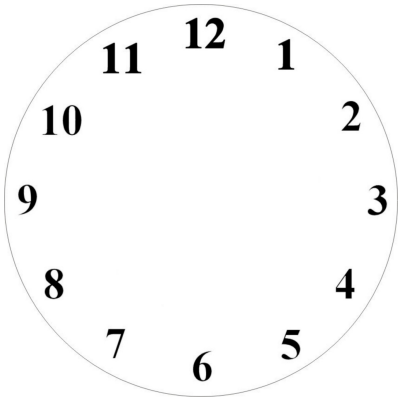


Рис. 3.4: Циферблат с отметками часов

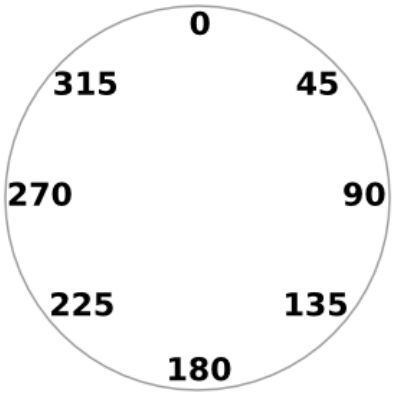


Рис. 3.5: Градусы.

```
>>> t.left(90)
```

Эта команда говорит черепашке повернуть налево на 90 градусов (то есть против часовой стрелки). Если ты ещё не знаешь про градусы и как ими меряют углы, то это можно представить себе вот как. На рисунке 3.4 есть циферблат от часов.

На циферблате по кругу написаны числа от 1 до 12 (или до 60, если там написаны минуты). Так вот градусы пишутся так же по кругу, только всё умножается на 30. Вместо цифры 3 — 90° (90 градусов), вместо шести — 180°, как на рисунке 3.5. А черепашка как будто стоит в центре циферблата и смотрит в сторону нуля, наверх.

И что происходит, когда мы отдаём команду `left(90)`?

Если встать, поднять руку ровно в сторону и показать туда, то чтобы повернуться лицом в ту сторону, в которую ты показываешь, нужно повернуться как раз на 90 градусов. Если показывать правой рукой, то на 90° вправо, левой рукой — на 90° влево. Так же и черепашка в питоне поворачивается туда, где

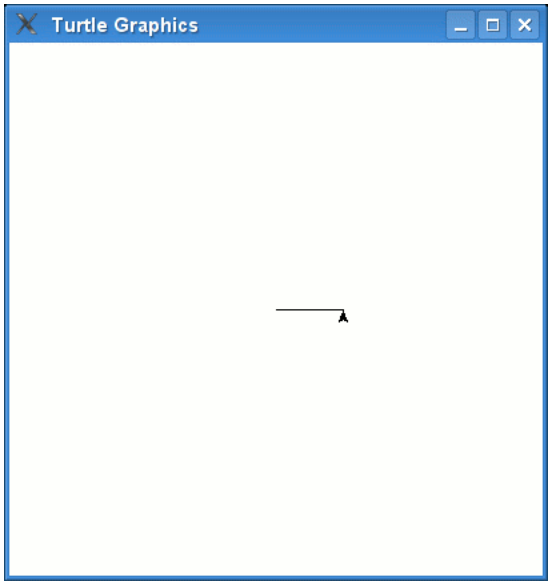


Рис. 3.6: Черепашка, повернувшая налево.

её правый бок или левый. При этом голова черепашки остаётся на месте (рисует она как раз маркером, зажатым в зубах), так что функция `t.left(90)` приводит к тому, что есть на рисунке 3.6. Черепашка ползла вправо и повернула вверх.

Давай теперь посмотрим, как все эти команды работают вместе:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

На экране черепашка нарисовала квадрат и остановилась, глядя в ту же сторону, что и в начале пути, как на рисунке 3.7.

Можно взять и очистить весь холст, воспользовавшись функцией `clear` (что как раз и переводится как «очистить»):

```
>>> t.clear()
```

Есть и другие полезные функции, применимые к черепашке. Например, `reset`: тоже очищает экран и ещё перемещает черепашку в начальное положение. Ещё есть функция `backward`, которая говорит черепашке двигаться назад (при этом направление её взгляда не меняется, она просто пятится). Функция `right` говорит черепашке повернуть направо; функция `up` говорит ей оторвать маркер от холста, то есть перестать рисовать при движении: не всё

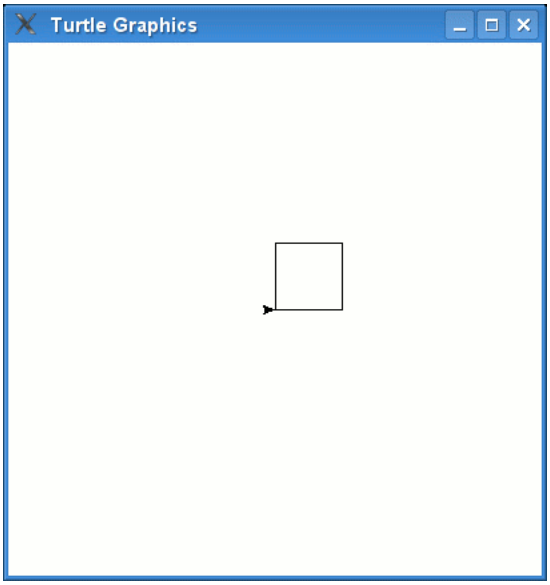


Рис. 3.7: Квадрат нарисовался.

можно нарисовать, если рисовать при каждом движении, иногда нужно просто переместиться. Есть и функция `down`, которая говорит ей обратно опустить маркер на холст и снова рисовать, пока она перемещается. Все эти функции вызываются таким же образом, как в примерах выше:

```
>>> t.reset()
>>> t.backward(100)
>>> t.right(90)
>>> t.up()
>>> t.down()
```

В следующих главах мы ещё воспользуемся услугами черепашки.

3.1. Как ещё развлечься

В этой главе мы познакомились с маленькой черепашкой, которая нарисовала нам немного линий, поворачиваясь направо и налево. Ещё мы обсудили градусы, которые здорово похожи на числа на циферблате часов.

Упражнение 1

Создай холст, используя функцию `Pen`, и нарисуй там прямоугольник.

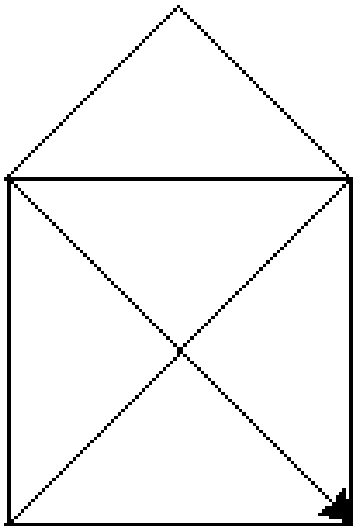


Рис. 3.8: Домик, который нарисовали, не отрывая карандаш от бумаги.

Упражнение 2

Создай холст, используя функцию `Pen`, и нарисуй там треугольник.

Упражнение 3*

Создай холст, используя функцию `Pen`, и нарисуй там домик, как на рисунке 3.8, не отрывая маркер от холста (не пользуясь функцией `up`). Это упражнение сложнее предыдущих и может не получиться без посторонней помощи, ничего страшного[1].

[1] Чтобы всё получилось, нужно иметь в виду, что может понадобиться повернуть на 45 или 135 градусов и что если стороны домика длиной 100 точек, то косые линии будут длиной примерно 71 и 141 точка.

Глава 4

Как задать вопрос

С точки зрения программистов, вопрос задаётся тогда, когда в зависимости от ответа нужно выполнить одни команды или другие. Во многих языках программирования такой вопрос записывается с использованием слова `if`, что на русский переводится как «если»[2]. Такое выражение ещё называется **условным оператором**.

Сколько тебе лет? Если тебе больше двадцати, ты супер стар!

Утверждение выше на Питоне может быть записано вот так:

```
if возраст > 20:
    print('ты супер стар!')
```

Условный оператор состоит из слова `if`, после которого записывается условие, завершаемое двоеточием (`:`). Все следующие строки, которые выполняются в зависимости от этого условия, должны начинаться с одинакового количества пробелов. Большинство людей использует тут 4 пробела, потому что так уже легко видеть *блок кода*, и при этом он не слишком далеко уезжает направо. Для вставки такого отступа в начало строки обычно используют клавишу `Tab`, она на клавиатуре слева (←) под цифрами, слева от буквы Й.

Если ответ на вопрос, который написан после `if`, — да, или `True`, как это записывается в Питоне, то блок кода, начинающийся с отступов, выполняется.

Условие, которое надо писать после `if`, — это выражение, на которое можно ответить «да» (`True`, «истина») или «нет» (`False`, «ложь»). Чтобы записывать условия, есть специальные значки, вот они:

[2] В некоторых языках прямо по-русски и пишут «если», но всё же обычно так не принято делать. В Питоне надо писать `if`.

==	равно
!=	не равно
>	больше чем
<	меньше чем
>=	больше чем или равно
<=	меньше чем или равно

Например, если тебе 10 лет, то условие `твой_возраст == 10` истинно (равно `True`). Если же тебе не 10 лет, то оно ложно (равно `False`). Тут есть хитрость: чтобы сравнить два числа, надо написать два знака равенства подряд. Если написать только один, будет ошибка. А один знак нужно писать, чтобы в переменную занести какое-нибудь значение, то есть никак не после `if`.

Теперь допустим, что тебе больше 10 лет и в переменной `age` хранится твой возраст. Тогда вот такое условие...

```
age > 10
```

...будет равно `True`. А если тебе меньше 10 лет, то это условие будет равно `False`. И если тебе 10 лет, условие тоже будет ложно, зато будет истинно условие `age>=10`.

Давай попробуем теперь ввести примеры в консоль:

```
>>> age = 10
>>> if age > 10:
...     print('я тут!')
```

Если ввести это в консоль, что произойдёт?..

Да ничего.

Переменная `age` не больше 10, так что `print` не выполнится. А как насчёт такого:

```
>>> age = 10
>>> if age >= 10:
...     print('тут я!')
```

Вот если этот пример запустить, то Питон выведет сообщение в консоль. И следующий пример тоже сработает:

```
>>> age = 10
>>> if age == 10:
...     print('вот я где!')
вот я где!
```

4.1. Сделай вот это... ИЛИ ВОТ ЭТО!

Можно расширить условный оператор и сказать Питону, что делать, когда условие ложно. Можно, например, напечатать в консоль «Привет», если тебе 12 лет или «Пока» в ином случае. Для этого пригодится слово `else` (в переводе — «иначе»).

```
>>> age = 12
>>> if age == 12:
...     print('Привет!')
... else:
...     print('Пока.')
Привет!
```

Если ты напечатаешь в консоль этот пример, то увидишь в ответ «Привет!». Стоит изменить значение переменной `age` на что-нибудь другое, как сообщение от Питона поменяется:

```
>>> age = 8
>>> if age == 12:
...     print('Привет!')
... else:
...     print('Пока.')
Пока.
```

4.2. Сделай вот это... или ещё вот это... ИЛИ ВОТ ЭТО!

Можно ещё дальше расширить условный оператор, используя слово `elif` (сокращение от «else if»). Например, можно вот так печатать, сколько тебе лет (да, не слишком полезно, но позволяет ухватить суть этого условного оператора):

```
1. >>> age = 12
2. >>> if age == 10:
3. ...     print('похоже, тебе 10 лет')
4. ... elif age == 11:
5. ...     print('я знаю, тебе 11 лет')
6. ... elif age == 12:
7. ...     print('ух ты, а тебе 12 лет')
8. ... elif age == 13:
9. ...     print('тебе целых 13 лет!')
10. ... else:
11. ...     print('Столько люди не живут.')
12. ...
13. ух ты, а тебе 12 лет
```

В примере кода выше строка 2 проверяет, равно ли значение возраста 10.

Если нет, то сразу после этого выполняется строчка 4, которая проверяет, равно ли значение возраста 11. Если нет — то проверяется условие в строке 6. Оно оказывается истинным, поэтому выполняется строка 7 и больше никаких проверок не производится.

4.3. Комбинируем условия

Можно проверять внутри одного условия сразу несколько выражений. Для этого используются английские слова «и»: `and` и «или»: `or`. Так например, пример выше можно было бы записать следующим образом, объединив проверки в одно большое условие:

```
1. >>> if age == 10 or age == 11 or age == 12 or age == 13:
2. ...     print('Я знаю, тебе %s лет' % age)
3. ... else:
4. ...     print('Столько люди не живут.')
```

Если любое из условий в строке 1 истинно, то все следующие и не проверяются и выполняется блок кода, следующий за `if`, то есть строка 2 в этом примере. Если же все условия ложны, то выполнится блок кода под `else`, то есть строчка 4. Этот пример можно ещё сократить, воспользовавшись операциями сравнения <= и >=:

```
1. >>> if age >= 10 and age <= 13:
2. ...     print('Тебе %s лет' % age)
3. ... else:
4. ...     print('А сколько же?')
```

Тут если твой возраст не меньше 10 лет и не больше 13, то Питон напечатает, сколько тебе лет, а иначе — удивится.

4.4. Пустота

Есть ещё специальное значение, которое можно присвоить любой переменной, и о котором мы раньше не говорили: **ничего**.

Точно так же, как переменной можно присвоить числа, строки и списки, переменной можно присвоить и «ничего». В Питоне это записывается словом `None` и значит, что в переменной ничего нет (в других языках используют слова типа `nil`, `null`, `nullptr`). При этом значение этой переменной можно напечатать, как и значение любой другой, и это не вызовет ошибки, как было бы, если бы переменная вообще не была объявлена.

```
>>> myval = None
>>> print(myval)
None
```

Присвоить переменной None может быть нужно, чтобы указать, что переменная чему-то будет равна потом, но сейчас её значение неизвестно[1].

Вот пример: допустим, мы хотим сходить в кино втроём, и для этого нам надо скинуться деньгами, кому сколько не жалко. Когда все решат, сколько им не жалко, и положат сумму, например, в конверт, можно на эти деньги выбрать фильм и купить билеты (а чтобы показать все фильмы из ближайших кинотеатров, на которые хватит этих денег, вполне можно написать программу на Питоне). Так вот, для этого мы заведём три переменных для каждого из зрителей и запишем в них None, что будет значить, что человек ещё вообще не сдавал деньги (если бы мы записали туда 0, это бы значило, что человек не хочет и не будет сдавать денег):

```
>>> зритель1 = None
>>> зритель2 = None
>>> зритель3 = None
```

Теперь можно проверить, все ли сдали деньги, пользуясь if'ом:

```
>>> if зритель1 is None or зритель2 is None or зритель3 is None:
...     print('Надо подождать ещё, не все сдали деньги')
... else:
...     print('Мы собрали %s руб.' % (зритель1 + зритель2 + зритель3))
```

if проверяет, записано ли в какую-то переменную значение None и, если это так, сообщает об этом. Если же в каждую переменную записано, кто сколько сдал денег, то Питон напечатает нам общую собранную сумму.

Вот что будет, если только два человека решились:

```
>>> зритель1 = 100
>>> зритель2 = None
>>> зритель3 = 300
>>> if зритель1 is None or зритель2 is None or зритель3 is None:
...     print('Надо подождать ещё, не все сдали деньги')
... else:
...     print('Мы собрали %s руб.' % (зритель1 + зритель2 + зритель3))
Надо подождать ещё, не все сдали деньги
```

А вот, если все трое:

[1] Если же хочется вообще удалить переменную, то надо не присвоить ей None, а написать вот так (для переменной myval): del myval.

```
>>> зритель1 = 100
>>> зритель2 = 500
>>> зритель3 = 300
>>> if зритель1 is None or зритель2 is None or зритель3 is None:
...     print('Надо подождать ещё, не все сдали деньги')
... else:
...     print('Мы собрали %s руб.' % (зритель1 + зритель2 + зритель3))
Мы собрали 900 руб.
```

4.5. В чём разница...?

Какая разница между `10` и `'10'`?

Так вообще, кажется, что, кроме пары кавычек, её и нет. Хотя вот в предыдущих главах ты узнал, что `10` — это число, а `'10'` — строка. И это различие гораздо существеннее, чем можно подумать.

Недавно мы проверяли, чему равен возраст, вот так:

```
>>> if age == 10:
...     print('помни: тебе 10 лет')
```

И если в переменную `age` записать значение 10, то всё, что надо, на экране напечатается:

```
>>> age = 10
>>> if age == 10:
...     print('помни: тебе 10 лет')
...
помни: тебе 10 лет
```

Но если в ту же переменную записать `'10'` (с кавычками), то ничего печататься не будет:

```
>>> age = '10'
>>> if age == 10:
...     print('помни: тебе 10 лет')
...
```

Как же так? Почему теперь ничего не работает? Ну потому что строка — это не число, хотя и выглядят они одинаково:

```
>>> age1 = 10
>>> age2 = '10'
>>> print(age1)
10
>>> print(age2)
10
```

Вот, видишь! Выглядят совсем одинаково, если напечатать. Но число никогда не будет равно строке.

Это вроде как странно, но смысл какой-то такой, что если сравнивать 10 книг и 10 кирпичей, они никогда не будут равны — они просто разные. То есть можно сравнить 10 штук книг и 10 штук кирпичей — количество (число) одинаковое, но сказать, что 10 кирпичей — это одно и то же («равно»), что и 10 книг, вряд ли получится. Вот так и тут.

Но это не страшно, Питон умеет прочитать строку и понять, какое число там записано, и наоборот, записать цифрами число в строку. Вот так можно превратить строку `'10'` в число `10`:

```
>>> age = '10'
>>> converted_age = int(age)
```

Теперь переменная `converted_age` хранит число 10 (не строку). Функция `int`, которая используется для такого преобразования, названа как сокращение от английского слова «integer», что значит «целое число», число без дробной части, без запятой.

Чтобы обратно перевести число в строку, есть функция `str` (сокращение от «string», «строка»):

```
>>> age = 10
>>> converted_age = str(age)
```

Теперь в переменной `converted_age` лежит строка `'10'`. Самое время вернуться к тому сравнению, которое у нас не работало:

```
>>> age = '10'
>>> if age == 10:
...     print('Тебе %s лет' % age)
...
```

Если мы преобразуем переменную перед проверкой, тогда мы получим другой результат:

```
>>> age = '10'
>>> converted_age = int(age)
>>> if converted_age == 10:
...     print('Тебе %s лет' % age)
...
Тебе 10 лет
```

Или даже прямо так, короче и без дополнительной переменной:

```
>>> age = '10'
>>> if int(age) == 10:
...     print('Тебе %s лет' % age)
...
Тебе 10 лет
```

Глава 5

Снова и снова

Нет ничего хуже, чем делать одно и то же много раз подряд, одинаково и монотонно. Примерно поэтому родители могли когда-нибудь советовать тебе считать воображаемых овец, пока ты не уснёшь (а вовсе не потому, что эти шерстяные животные обладают магическими способностями усыплять). Всё дело в том, что повторять без конца одно и то же — скучно, и твоё сознание скоро сдастся и уступит место сну.

Программисты тоже не большие любители повторять одни и те же действия. От этого их тоже клонит в сон. Так что почти во всех языках программирования есть такая штука как *цикл*[2]. В частности, *цикл for*. И вот зачем. Допустим, тебе захочется 5 раз напечатать «Привет». Можно было бы сделать это так:

```
>>> print('Привет')
Привет
>>> print('Привет')
Привет
>>> print('Привет')
Привет
>>> print('Привет')
Привет
>>> print('Привет')
Привет
```

Но так писать очень утомительно.

Вместо того, чтобы повторять кучу раз одно и то же, можно использовать цикл `for` (не забудь про 4 пробела во второй строчке перед `print`):

```
>>> for x in range(0, 5):
...     print('Привет')
```

[2]Некоторые языки обходятся без циклов, а некоторые — и без переменных. Но, как я и говорила, о других языках тут мы поговорить просто не успеем.

```
...
Привет
Привет
Привет
Привет
Привет
```

Функция `range` — быстрый способ создать список чисел от начального значения (включительно) до конечного (не включительно). Вот пример:

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

И в нашем цикле `for` на самом деле строчка `for x in range(0, 5)` говорит Питону создать список чисел `[0, 1, 2, 3, 4]`, сохранить по очереди каждое из этих чисел в переменной `x` и выполнить для каждого значения этой переменной блок кода, написанный под `for`. Переменную `x` тоже можно использовать для печати:

```
>>> for x in range(0, 5):
...     print('[%s] Привет' % x)
...
[0] Привет
[1] Привет
[2] Привет
[3] Привет
[4] Привет
```

Если явно записать то, что происходит в цикле `for`, то получится что-то такое:

```
x = 0
print('[%s] Привет' % x)
x = 1
print('[%s] Привет' % x)
x = 2
print('[%s] Привет' % x)
x = 3
print('[%s] Привет' % x)
x = 4
print('[%s] Привет' % x)
```

Так что цикл нас тут избавил от лишних восьми строк кода. И это невероятно полезно, потому что средний программист чуть более ленив, чем бегемот жарким днём, когда дело доходит до печатания кода. Хорошие программисты ненавидят делать одно и то же больше одного раза (и на то есть серьёзные

причины[1]), так что цикл — одна из самых полезных инструкций в языке программирования.

Аккуратней с пробелами!

Возможно, ты пытался запустить примеры из этой главы и в качестве результата получил только сообщение об ошибке вроде такого:

```
IndentationError: expected an indented block
```

Такая ошибка значит, что в начале строки не хватает пробелов, хотя бы одного. Пробелы в начале строки в Питоне очень важны. Скоро мы о них поговорим подробнее...

Можно не ограничиваться функцией `range` для цикла `for`, можно использовать и другие списки (которые мы раньше создавали). Вот, например, список покупок:

```
>>> shopping_list = [ 'яиц', 'молока', 'сыра', 'сельдерея', 'масла', 'соды' ]
>>> for i in shopping_list:
...     print('Как бы не забыть купить %s' % i)
Как бы не забыть купить яиц
Как бы не забыть купить молока
Как бы не забыть купить сыра
Как бы не забыть купить сельдерея
Как бы не забыть купить масла
Как бы не забыть купить соды
```

Этот код значит «сохрани по очереди каждый из элементов списка в переменной `i` и выполни `print` для каждого из значений».

Опять же, если бы мы решили делать это без списка, то пришлось бы печатать что-нибудь такое:

[1] Например, если найдётся ошибка в каком-то коде, то исправить её нужно будет только один раз. Если один и тот же ошибочный код использовать несколько раз, то где-то ошибку точно забудут исправить.

```
>>> shopping_list = [ 'яиц', 'молока', 'сыра', 'сельдерея', 'масла', 'соды' ]
>>> print('Как бы не забыть купить %s' % shopping_list[0])
Как бы не забыть купить яиц
>>> print('Как бы не забыть купить %s' % shopping_list[1])
Как бы не забыть купить молока
>>> print('Как бы не забыть купить %s' % shopping_list[2])
Как бы не забыть купить сыра
>>> print('Как бы не забыть купить %s' % shopping_list[3])
Как бы не забыть купить сельдерея
>>> print('Как бы не забыть купить %s' % shopping_list[4])
Как бы не забыть купить масла
>>> print('Как бы не забыть купить %s' % shopping_list[5])
Как бы не забыть купить соды
```

Так что `for` снова спас нас от кууучи лишнего кода.

5.1. Зачем нужны блоки программистам?

Чтобы делать их из кода.

Что же такое «блок кода»?

Блок кода — это несколько программных инструкций, объединённых вместе. Например, в примере с циклом `for` может захотеться не только печатать, что надо купить, но и как-нибудь покупать. Предположим, что есть функция `buy` для этого (на самом деле её пока нет). Тогда можно записать этот пример вот так:

```
>>> for i in shopping_list:
...     buy(i)
...     print(i)
```

Печатать это в консоль не стоит и пытаться, потому что функции `buy` нету, и результатом будет только сообщение об ошибке. Зато тут есть настоящий блок кода, вот этот:

```
buy(i)
print(i)
```

В Питоне пробельные символы, а именно символ табуляции (который получается, если нажать клавишу Tab[1]) и символ пробела, очень важны. Код с одинаковым отступом от начала строки автоматически группируется в блоки.

[1] В консоли Питона этот способ не сработает, только в текстовом редакторе так получится.

```
это блок 1
это блок 1
это блок 1
    это блок 2
    это блок 2
    это блок 2
это всё ещё блок 1
это всё ещё блок 1
    это блок 3
    это блок 3
        это блок 4
        это блок 4
        это блок 4
```

Единственное правило тут — что нужно использовать одно и то же количество пробелов в начале строк внутри всего блока. Вот так неправильно:

```
>>> for i in shopping_list:
...     buy(i)
...       print(i)
```

Отступ второй строки здесь 4 пробела, а третьей строки — 6 пробелов, её начало сдвинуто ещё на 2 символа вправо. Такой код вызовет ошибку. Если уж блок начался с отступом в 4 пробела, то так он и должен продолжаться. А если хочется внутри блока поместить другой, то нужно будет использовать 8 пробелов (два раза по четыре) в начале строк этого внутреннего блока. Вот так правильно:

```
    вот первый блок кода
    вот первый блок кода
```

И вот блок кода, в который вложен ещё один. Все строчки в нём начинаются с 8 пробелов:

```
    вот первый блок кода
    вот первый блок кода
        вот второй блок кода
        вот второй блок кода
```

Зачем может захотеться вложить один блок кода в другой? Обычно это случается, когда второй блок зависит от первого. Например, если второй блок — тело цикла `for`. Строчка со словом `for` находится в первом блоке кода, а всё, что должно выполняться в цикле, нужно поместить во второй блок.

Если ты начинаешь блок в консоли Питона, то Питон продолжает этот блок до тех пор, пока ты не нажмёшь Enter в пустой строке. Каждую строчку блока при этом Питон начинает с трёх точек.

Давай попробуем ввести в консоль пример. Напечатай туда код, написанный ниже, не забывая про 4 пробела в начале строк в блоке кода после `for`[1].

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)
...     print(i * 3)
...
a
aaa
b
bbb
c
ccc
```

После второй строчки с `print` нужно нажать Enter на пустой строке — так Питон поймёт, что ты хочешь завершить блок. И напечатает всё, что просили.

Следующий пример вызовет ошибку:

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)
...       print(i)
...
File "<stdin>", line 3
  print(i)
  ^
IndentationError: unexpected indent
```

В начале второй строчки с командой `print` вместо четырёх пробелов шесть, что Питону крайне не нравится: отступы должны быть одинаковыми, какие бы они ни были.

Запомни

Если ты используешь четыре пробела, чтобы сделать отступ для одного из блоков кода, то все следующие блоки нужно тоже начинать с четырёх пробелов. Если же для отступов используются два пробела, то два и нужно использовать во всей программе. Большинство людей используют четыре пробела, чего я и тебе советую.

Вот пример чуть посложнее с двумя блоками кода:

[1] Вообще, в консоли хватит и одного пробела. Для блока, вложенного в первый, — двух пробелов и так далее. 4 пробела используют в программах, которые сохраняют в файлы и которые потом люди будут читать ещё раз.

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)
...     for j in mylist:
...         print(j)
...
```

Где в этом коде блоки и что он делает?..

Блоков два. Первый — часть первого цикла `for`:

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)                #
...     for j in mylist:        #<-- все эти строки - это первый блок кода
...         print(j)            #<-- эта строка - тоже часть первого блока
...
```

Второй блок состоит из одной строки, выполняющейся во вложенном цикле `for`:

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)
...     for j in mylist:
...         print(j)              #<-- вот эта строка - второй блок кода
...
```

Можешь ли ты понять, глядя на этот код, что он делает?

Он напечатает каждую из трёх букв из списка, но сколько раз? Если мы внимательно посмотрим на код, то, скорее всего, сможем ответить на этот вопрос. Мы знаем, что первый цикл `for` выполнится для каждой буквы один раз. Он напечатает эту букву, а потом запустит вложенный цикл. А вложенный цикл опять выполнится по разу для каждой буквы и просто её напечатает. Получается, что сначала на экране появится буква 'a', затем — по очереди буквы 'a', 'b', 'c'. Потом — буква 'b' и опять все три и потом — буква 'c'. Можешь ввести этот код в консоль Питона и собственноручно убедиться, что всё так и будет.

```
>>> mylist = [ 'a', 'b', 'c' ]
>>> for i in mylist:
...     print(i)
...     for j in mylist:
...         print(j)
...
a
a
b
c
b
a
b
c
c
a
b
c
```

Кажется, я слышу вопрос, можно ли использовать циклы для чего-то немного более полезного, чем печать букв одной за одной? Можно, конечно, иначе бы и не стали циклы придумывать. Вспомни пример из второй главы, когда мы считали, сколько карманных денег накопится к концу года. Там тебе платили 5 рублей в неделю за домашние дела, 30 рублей в неделю за разнос газет, и ещё ты тратил 10 рублей в неделю.

К концу года копилась вот такая сумма:

```
>>> (5 + 30 - 10) * 52
```

(5 рублей + 30 рублей − 10 рублей, и всё это умножить на 52 недели в году).

Может захотеться посмотреть, сколько денег накопится в каждую из недель года, чтобы прикинуть, насколько быстро удастся накопить на ту или иную вещь, и выбрать: на что копить, чтобы было не слишком долго (и сколько именно придётся копить). Этого можно добиться, как раз используя цикл for. Однако прежде всего надо сохранить все эти числа в переменные[1]:

```
>>> chores = 5
>>> paper = 30
>>> spending = 10
```

Теперь исходный пример записывается так:

[1] В любых программах не принято использовать числа или строки просто так. Их сохраняют в переменные. Во-первых, так сразу видно, что значит число (если переменная названа понятно); во-вторых, его легко поменять всего в одном месте.

```
>>> (chores + paper - spending) * 52
1300
```

А увидеть, сколько денег накопится в каждую из недель, можно так:

```
1. >>> savings = 0
2. >>> for week in range(1, 53):
3. ...     savings = savings + chores + paper - spending
4. ...     print('В конце недели № %s будет %s рублей' % (week, savings))
5. ...
```

В строке 1 в переменную savings заносится начальное значение: 0 (ещё ничего не накопилось).

В строке 2 записан цикл `for`, который выполнится по разу для каждой из недель в году. На каждом шаге цикла переменная `week` будет иметь значение от 1 до 52[1].

В строке 3 к тому, что мы уже накопили, прибавляется ещё то, что накопится в очередную неделю. Иными словами, в переменную `savings` записывается столько, сколько там уже было, и ещё столько, сколько добавится на этой неделе. Знак равенства «=» говорит Питону посчитать всё, что справа от этого знака, и сохранить результат в переменную слева — и именно в таком порядке. Так что если и справа и слева от знака равенства есть одна и та же переменная, ничего страшного.

В строке 4 печатается результат расчётов: сколько денег будет в конце каждой недели, считая с нынешней.

Если теперь запустить эту программу, получится вот что:

```
В конце недели № 1 будет 25 рублей
В конце недели № 2 будет 50 рублей
В конце недели № 3 будет 75 рублей
В конце недели № 4 будет 100 рублей
В конце недели № 5 будет 125 рублей
В конце недели № 6 будет 150 рублей
В конце недели № 7 будет 175 рублей
В конце недели № 8 будет 200 рублей
В конце недели № 9 будет 225 рублей
В конце недели № 10 будет 250 рублей
В конце недели № 11 будет 275 рублей
В конце недели № 12 будет 300 рублей
В конце недели № 13 будет 325 рублей
В конце недели № 14 будет 350 рублей
В конце недели № 15 будет 375 рублей
```

...и так далее до 52-й недели.

[1] range(1,53) возвращает список чисел от 1 включительно до 53 не включительно, то есть [0,...,52].

5.2. Пока мы тут говорим про циклы...

Циклом `for` в питоне дело не ограничивается. Есть ещё цикл `while` (что по-русски значит «пока» или «в то время как»). В Питоне для цикла `for` заранее известно, когда он остановится: когда переберёт все элементы (или раньше, если его прервать, но никак не позже). Цикл `while` используется в тех случаях, когда заранее непонятно, когда остановиться.

Представь себе освещённую лестницу из 20 ступенек. И представь, что по ней нужно подняться, печатая номер каждой очередной ступеньки[1]. Если мы представим, что есть команда `step_up()`, которую нужно вызвать, чтобы сделать шаг, то всё это можно записать таким циклом `for`:

```
>>> for step in range(0,20):
...     step_up()
...     print(step)
```

А теперь представь, что эта лестница не освещена; неизвестно, сколько в ней ступенек, а идти нужно до первой двери, а потом уйти в эту дверь с лестницы (и вообще, это длинная-длинная винтовая лестница в старинной магической башне). Тут не получится использовать цикл `for`, потому что наперёд неизвестно, сколько всего ступенек. Тут-то и нужен `while`:

```
>>> step = 0
>>> while no_door():
...     step_up()
...     print(step)
...     if tired():
...         break
...     else:
...         step = step + 1
>>> print('Остановились на ступеньке %s' % step)
```

Этот код нет смысла запускать, потому что он использует выдуманные функции: `no_door()` — эта функция проверяет, есть ли дверь, и возвращает `True`, если её нет — и `tired()` — эта функция возвращает `True`, если сил идти вверх больше нет. Но этот код показывает смысл цикла `while`. Пока условие истинно, то есть пока `no_door()` — правда, цикл выполняется. Внутри цикла мы проверяем, есть ли ещё силы идти вверх, и если нет, то тут же и прерываем цикл (`break`) и переходим к самой последней строчке кода. Потом, за

[1] Между прочем, это уже вполне реальная ситуация. Когда роботов учат подниматься по лестнице, что даётся им непросто, после подъёма на каждую очередную ступеньку робот проверяет, что он не упал, и печатает номер ступеньки. Или понимает, что упал, и печатает номер ступеньки, на которой упал. Потом его программу улучшают так, чтобы он не падал.

циклом мы можем ещё раз проверить `no_door`, чтобы понять, почему цикл завершился: надо ли войти в дверь или отдохнуть и идти дальше. Если же силы идти есть, то номер ступеньки увеличивается, и мы делаем следующий шаг.

Итак, цикл `while` устроен следующим образом:

- ▷ проверить условие,
- ▷ выполнить блок кода,
- ▷ повторить всё с начала.

Часто цикл `while` проверяет сразу несколько условий. Например, так:

```
>>> x = 45
>>> y = 80
>>> while x < 50 and y < 100:
...     x = x + 1
...     y = y + 1
...     print(x, y)
```

Тут есть две переменные и два условия, по одному для каждой. В цикле переменная `x` изменяется от 45 до 50, а `y` — от 80 до 100. Когда первая из переменных достигнет своего максимального значения (50 для `x` и 100 для `y`), цикл тут же завершится.

```
46 81
47 82
48 83
49 84
50 85
```

Наверное, не стоит тебе надоедать излишне подробными объяснениями, почему напечатались именно эти строки[1].

Ещё `while` используют, чтобы создать бесконечный цикл или почти бесконечный, вот так:

```
>>> while True:
...     lots of code here
...     lots of code here
...     lots of code here
...     if some_condition == True:
...         break
```

[1] ...или всё-таки стоит? Вначале `x` равняется 45, а `y` — 80. Обе переменные увеличиваются на 1 на каждом шаге цикла и печатаются. Так и получается то, что напечатано. Когда `x` равно 50, условие `x < 50` перестаёт быть истинным (`True`), поэтому и всё вместе условие цикла «`x < 50 и y < 80`» тоже перестаёт быть истинным, и цикл завершается.

Условие для этого цикла `while` — просто «`True`». Это условие всегда выполняется, и тело цикла всегда будет запускаться, это вечный цикл. Но как только переменная `some_condition` тоже примет значение `True`, цикл прервётся инструкцией `break`.

Другой пример почти вечного цикла ты можешь подсмотреть в конце книги, в приложении **??** (там где про модуль `random`). А можешь подождать и сперва прочитать следующую главу, в которой как раз рассказывается про модули.

5.3. Как ещё развлечься

В этой главе мы увидели, как использовать циклы, чтобы выполнять повторяющиеся действия. Мы использовали блоки кода внутри циклов для записи этих действий.

Упражнение 1

Как ты думаешь, что произойдёт, если запустить этот код:

```
>>> for x in range(0, 20):
...     print('hello %s' % x)
...     if x < 9:
...         break
```

Упражнение 2

Если положить деньги в банк, то банк за них платит проценты — за то, что он может ими пользоваться на своё усмотрение и давать кому-то в долг (под большие проценты). Каждый месяц банк добавляет на твой счёт какую-то небольшую часть от того, что на нём уже лежит. Например, если положить в банк 1000 рублей, то банк добавляет каждый месяц ещё по 10 рублей (сколько именно, зависит от банка, но примерно столько или меньше).

Так вот, когда ты кладёшь деньги в банк, он сообщает тебе, сколько *процентов* в год банк добавит. Допустим, ты положил 1000 рублей, и банк добавит 5% в год — тогда через год у тебя будет $1000 + 1000 \times 0.05$ рублей. Если банк обещает добавить 10%, то через год у тебя будет $1000 + 1000 \times 0.10$ рублей — и так далее. Предположим, проценты выплачиваются в конце каждого года (хотя это и не так, но так проще считать и результаты получаются почти точные). Тогда, если ты положишь 1000 рублей под 7% годовых, через год у тебя будет $1000 + 1000 \times 0.07$. Потом ещё через год банк добавит 7% уже на эту

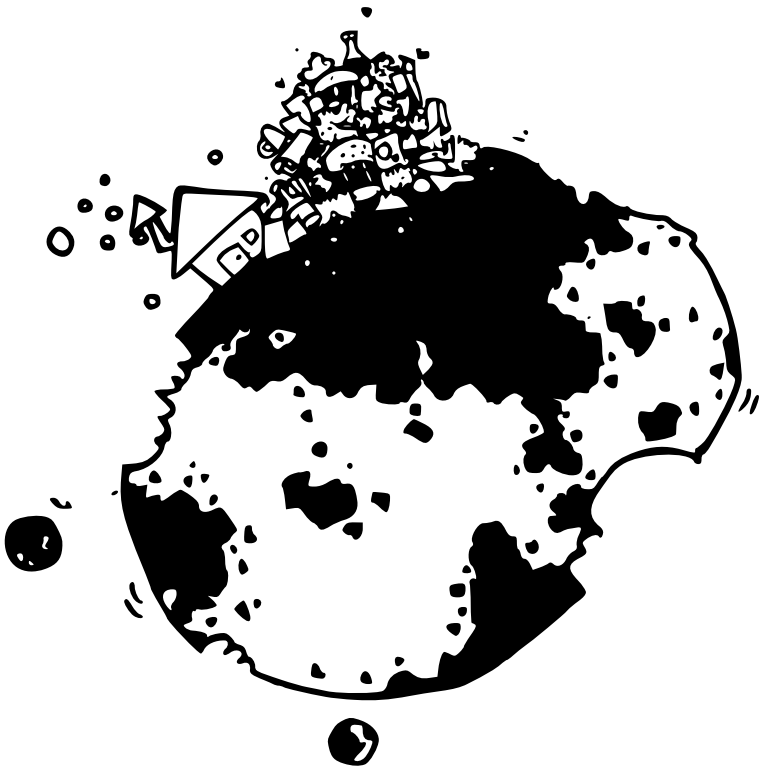
сумму. Ещё через год — на сумму, оставшуюся в конце второго года, — и так далее. В уме это посчитать сложно; а в жизни такая задача возникает часто.

Напиши программу, которая посчитает, сколько на счёте накопится денег через 10 лет, если положить 10 000 рублей под 15% годовых (не забудь, что каждый год проценты считаются от той суммы, что лежала в конце прошлого года, а не той, что лежала в самом начале всех этих 10 лет).

Глава 6

Повторное использование...

Представь, сколько мусора ты производишь каждый день. Всякие бутылочки от воды, обёртки от еды, пакетики от овощей, пластиковые коробочки, в которых продают мясо; а ещё, возможно, журналы, газеты или другая бумага. Ну и теперь представь, что получится, если этот весь мусор сложить в одну большую кучу.



Весь мусор, который ты выкидываешь, должен перерабатываться для повторного использования. И если повезёт, то таки перерабатывается[2], чтобы по пути в школу не приходилось перелазить через мусорные горы. Стеклянные

[2]Если не повезёт — то просто сжигается на мусоросжигательных заводах. Это вредно и не приносит пользы, но хотя бы очищает место от мусора.

бутылки переплавляются в новые стеклянные ёмкости, бумагу переваривают в упаковочный картон, пластик переплавляют в более плотные пластиковые упаковки и вещи — чтобы в твоём дворе не вырастали горы ненужного мусора. Вообще, переработка мусора — весьма полезная идея, чтоб не добывать одно и то же из земли, выгрызая там дырку, а вместо этого много раз использовать уже добытое.

Нет, я не сошла с ума и я всё ещё книжка про Питон. Просто в программировании такого сорта повторное использование тоже необходимо. Не стоит набирать один и тот же код много раз: нужно использовать повторно уже написанный раньше. Это сэкономит время, а также позволит найти и убрать все ошибки только из одного кусочка кода, а не из одинаковых кусочков по всей программе (и в каком-то точно ошибка забудется когда-нибудь). Кроме того, это упрощает понимание программы при её чтении: не нужно вчитываться в один и тот же код много раз.

В Питоне (да и во многих других языках программирования) есть несколько способов повторно использовать код. Один из них мы уже встречали — в главе 3, например: там была *функция* `range`. Функции пишутся один раз, а потом сколько угодно вызываются отовсюду.

Давай начнём знакомство с функциями с маленького примера:

```
>>> def myfunction(myname):
...     print('Здравствуй, %s' % myname)
...
```

Этот код *описывает функцию* с именем `myfunction` и с параметром `myname`. *Параметр*, или *аргумент функции* — это обычная переменная, которой можно пользоваться внутри функции (в *теле функции*), то есть внутри блока кода, который стоит после строчки, начинающейся с `def` (сокращения от английского `define`, «описывать»). Значение параметру задаётся снаружи, при вызове функции. Теперь нашу функцию можно вызвать так:

```
>>> myfunction('а где я?')
Здравствуй, а где я?
```

И переменная `myname` в функции приняла то начальное значение, которое мы написали в скобках, вызывая функцию.

Можно изменить эту функцию так, чтобы она принимала два параметра:

```
>>> def myfunction(firstname, lastname):
...     print('Привет, %s %s' % (firstname, lastname))
...
```

И тогда вызывать её вот так:

```
>>> myfunction('Гвидо', 'ван Россум')
Привет, Гвидо ван Россум
```

А можно создать переменные и передавать их в качестве значений аргументов:

```
>>> fn = 'Исаак'
>>> ln = 'Ньютон'
>>> myfunction(fn, ln)
Привет, Исаак Ньютон
```

Ещё функция может *возвращать значение*, для этого есть слово `return`. Вот так это всё используется:

```
>>> def savings(chores, paper, spending):
...     return chores + paper - spending
...
>>> print(savings(10, 10, 5))
15
```

Функция принимает три параметра, складывает значения первых двух и вычитает значение третьего из этой суммы. Полученный результат возвращается функцией — его можно, как в примере выше, напечатать или присвоить переменной (или передать в качестве параметра другой функции):

```
>>> my_savings = savings(20, 10, 5)
>>> print(my_savings)
25
```

Снаружи от функции любые переменные, которые мы создаём или изменяем внутри, недоступны (хотя и есть средство, чтобы они были доступны, его использование редко оправданно):

```
>>> def variable_test():
...     a = 10
...     b = 20
...     return a * b
...
>>> print(variable_test())
200
>>> print(a)
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
NameError: name 'a' is not defined
```

В примере выше мы создаём функцию `variable_test`, которая умножает две переменных, объявленных в этой же функции (`a` и `b`), а потом возвращает

результат. Чтобы вызвать эту функцию, надо написать её имя и пустые скобки: пустые — потому что функция не принимает параметров; но при каждом вызове любой функции надо писать скобки, чтобы отличить её от переменной. Так вот, если вызвать эту функцию, то мы получим ответ: 200. Но если после этого мы попытаемся напечатать значение переменной `a` (или переменной `b`, всё равно), то получим только ошибку. Это потому, что *область видимости* переменных, значения которым присваиваются в функции, ограничена телом этой функции.

Можно себе представить функцию как маленький остров посреди океана, куда никто не приплывёт. Но иногда мимо пролетает самолёт и сбрасывает коробочку с листами бумаги: на них записаны значения аргументов функции. Островитяне что-то делают с этими аргументами, а потом результат запечатывают в бутылку и кидают в океан в нужном месте: течение потом выносит этот результат на большую землю, где и ждут ответ. И что там происходит на острове, жители большой земли не знают — какие там переменные используются, чему они равны. А вот островитяне, наоборот, могут достать свой могучий бинокль и зорко глянуть на большую землю, чтобы подсмотреть значения переменных оттуда (но поменять их не могут). Примерно так:

```
>>> x = 100
>>> def variable_test2():
...     a = 10
...     b = 20
...     return a * b * x
...
>>> print(variable_test2())
20000
```

Переменные `a` и `b` созданы островитянами внутри функции и снаружи никак не видны. Переменная `x`, наоборот, создана снаружи (это *глобальная переменная*) и видна и с острова, она влияет на результат, но точно не меняется после вызова функции.

Теперь мы можем ещё вспомнить про цикл `for`, который мы использовали, чтобы узнать, когда сколько денег накопится. Его тоже можно добавить в функцию:

```
>>> def yearly_savings(chores, paper, spending):
...     savings = 0
...     for week in range(1, 53):
...         savings = savings + chores + paper - spending
...         print('К концу недели № %s накопится %s руб.' % (week, savings))
...
```

Попробуй ввести эту функцию в консоль и вызвать её с разными аргументами:

```
>>> yearly_savings (50, 100, 40)
К концу недели № 1 накопится 110 руб.
К концу недели № 2 накопится 220 руб.
К концу недели № 3 накопится 330 руб.
К концу недели № 4 накопится 440 руб.
К концу недели № 5 накопится 550 руб.
К концу недели № 6 накопится 660 руб.
К концу недели № 7 накопится 770 руб.
К концу недели № 8 накопится 880 руб.
К концу недели № 9 накопится 990 руб.
(... и так далее ...)
>>> yearly_savings (40, 120, 30)
К концу недели № 1 накопится 130 руб.
К концу недели № 2 накопится 260 руб.
К концу недели № 3 накопится 390 руб.
К концу недели № 4 накопится 520 руб.
К концу недели № 5 накопится 650 руб.
К концу недели № 6 накопится 780 руб.
К концу недели № 7 накопится 910 руб.
К концу недели № 8 накопится 1040 руб.
К концу недели № 9 накопится 1170 руб.
(... и так далее ...)
```

И это несколько удобнее, чем вводить заново `for` каждый раз, когда хочется запустить его с новыми значениями переменных.

Функции можно группировать в *модули*, и это то, что делает Питон по-настоящему очень удобным, а не просто удобным. Про модули подробнее мы поговорим чуть позже.

6.1. Кусочки и части

Когда Питон устанавливается на компьютер, вместе с ним устанавливается куча его модулей и функций. Некоторые функции можно использовать прямо сразу в консоли и программах, без каких-либо специальных предварительных инструкций. Так мы использовали функцию `range`. Есть ещё функция `open`, на которую мы сейчас взглянем подробнее.

Для этого давай откроем текстовый редактор. Напечатай туда несколько слов и сохрани этот новый файл в свою домашнюю папку, нажав, например, кнопку «сохранить» где-то слева в верху экрана (твои родители должны были показать тебе, как сохранять, где-то в первой главе) и выбрав в качестве места для сохранения домашнюю папку. Назови файл «test.txt» (должно получиться, например, как на рисунке 6.1)

Теперь снова открой питонью консоль и напечатай туда такие команды:

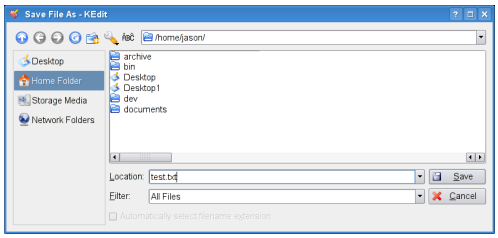


Рис. 6.1: Диалог сохранения может выглядеть, например, так

```
>>> f = open('test.txt')
>>> print(f.read())
```

Давай разберёмся, что этот кусочек кода делает. Первая строка вызывает функцию `open`, передавая ей имя файла в качестве параметра. Эта функция открывает файл, чтобы его потом использовать. Она возвращает специальное значение (объект), которое описывает этот файл. Потом у этого объекта можно вызывать функции, чтобы прочитать что-то из файла или записать в файл. Этот объект сохраняется в переменную `f` в нашей строчке кода.

Потом вторая строка вызывает у объекта `f` функцию `read` (используя точку после имени объекта: именно так вызываются функции у объектов), которая читает весь файл и возвращает результат как строку. Эта строка тут же и передаётся как параметр функции `print`, так что весь файл печатается в консоль.

> В приложении **??** (в конце книжки) написано ещё всякое разное про встроенные в Питон функции.

6.2. Модули

Ну что ж, с парой способов повторно использовать код мы уже познакомились. Во-первых, это функции, которые можно и самим создавать и вызывать, и использовать встроенные в Питон (`range`, `open`, `int`, `str`). Во-вторых, можно вызывать функции у объектов — про то, как их описывать, мы потом поговорим.

Ещё можно создавать модули: файлы, содержащие функции и объекты, сгруппированные вместе. Например, всё для работы со временем содержится в модуле `time`. Вот так мы говорим Питону, что будем в программе использовать функции и переменные из этого модуля:

```
>>> import time
```

Теперь мы можем обращаться к функциям из этого модуля, используя точку (точно так же, как к функциям какого-нибудь объекта):

```
>>> print(time.localtime())
(2006, 11, 10, 23, 49, 1, 4, 314, 1)
```

`localtime` — функция, которая возвращает текущие время и дату, сохранённые в кортеже (про кортежи было во второй главе раздел «Кортежи и списки», на странице 25): год, месяц, день, час, минуту, секунду, номер дня недели, номер дня года и переведены ли часы на летнее время (1, если да; 0 — если нет). Можно воспользоваться ещё одной функцией из модуля `time`, чтобы превратить этот кортеж в более понятную строку:

```
>>> t = time.localtime()
>>> print(time.asctime(t))
'Sun Aug 30 21:40:59 2015'
```

Можно это сделать в одну строку:

```
>>> print(time.asctime(time.localtime()))
'Sun Aug 30 21:41:16 2015'
```

А можно и ещё чуть сократить:

```
>>> time.asctime()
'Sun Aug 30 21:41:44 2015'
```

Потому что функция `asctime` как раз печатает текущее время, если никаких параметров ей не передавать.

Может случиться так, что тебе нужно будет попросить того, кто пользуется твоей программой, ввести значение. Для этого можно напечатать эту просьбу на экране и потом считать то, что твоей программе в ответ напечатают. Функция, чтобы читать то, что вводят с клавиатуры, содержится в модуле `sys`:

```
import sys
```

В этом модуле есть объект `stdin` (сокращение от английского «standard input», то есть «стандартный ввод»). А у этого объекта есть функция `readline` — считать строку текста, то есть всё, что напечатано до нажатия клавиши Enter. Клавиша Enter заканчивает строку (точно так же, как в любом текстовом редакторе). Можешь проверить, как эта функция работает, введя вот это в консоль:

```
>>> print(sys.stdin.readline())
```

Напечатай что-нибудь и нажми Enter. Питон ровно эту строку и напечатает.

Теперь можем вернуться к условному оператору `if` из четвёртой главы. Там был такой код:

```
>>> if age >= 10 and age <= 13:
...     print('Я знаю, тебе %s лет' % age)
... else:
...     print('Столько люди не живут.')
```

Вместо того чтобы заранее создавать и задавать значение переменной `age`, можно попросить кого-то ввести возраст с клавиатуры. Но сперва давай выделим этот код в функцию:

```
>>> def your_age(age):
...     if age >= 10 and age <= 13:
...         print('Я знаю, тебе %s лет' % age)
...     else:
...         print('Столько люди не живут.')
...
```

Эту функцию можно вызвать, передав ей возраст как параметр. Прежде всего, давай убедимся, что функция работает как надо:

```
>>> your_age(20)
Столько люди не живут.
>>> your_age(10)
Я знаю, тебе 10 лет
```

Итак, мы теперь знаем, что проблем с функцией нет — давай сделаем так, чтобы она считывала с клавиатуры, сколько человеку лет:

```
>>> def your_age():
...     print('Сколько тебе лет? Введи число и нажми Enter: ')
...     age = int(sys.stdin.readline())
...     if age >= 10 and age <= 13:
...         print('Я знаю, тебе %s лет' % age)
...     else:
...         print('Столько люди не живут.')
...
```

Тут в коде вызывается функция `int`. Этот потому что `readline` возвращает текст (строку, то есть), а нам результат надо сравнить с числами, то есть использовать как число (на странице 40 про это подробнее). Теперь можешь потестировать эту функцию: просто вызови её.

```
>>> your_age()
Сколько тебе лет? Введи число и нажми Enter:
10
Я знаю, тебе 10 лет
>>> your_age()
Сколько тебе лет? Введи число и нажми Enter:
15
Столько люди не живут.
```

Да, хотя ты печатаешь вроде как число, Питон считывает строку, со-

стоящую из цифр, но не число. Нужно не забыть превратить эту строку в число функцией `int`.

> `sys` и `time` — всего лишь два из кучи модулей, которые устанавливаются вместе с Питоном. Ещё про некоторые (не про все) написано в конце книжки, в приложении **??**.

6.3. Как ещё развлечься

Итак, в этой главе мы изучили, как в Питоне повторно использовать код: для этого придумали функции и модули. Мы немного обсудили область видимости переменных: что переменные, объявленные вне функции, можно использовать изнутри неё (но изменения не будут видны снаружи), а переменные, объявленные в функции, только в ней и видны. Ещё мы научились создавать функции, используя инструкцию `def`.

Упражнение 1

В предыдущей главе во втором упражнении мы использовали цикл `for`, чтобы посчитать, сколько денег накопится на нашем банковском счёте за 10 лет, если положить туда 10 000 рублей. Было бы удобно прикинуть, сколько накопится денег при других условиях. Преврати код из того упражнения в функцию, которая принимает три параметра:

1. Сколько денег ты кладёшь в банк;
2. На сколько лет;
3. Сколько процентов годовых обещает банк.

В результате эту функцию должно быть можно вызвать так, и она должна давать такие ответы:

```
>>> calculate_interest (1000, 1, 11)
1110.0
>>> calculate_interest (1200, 5, 13)
2210.922215159999
>>> calculate_interest (1500, 10, 15)
6068.33660356186
```

Упражнение 3

Вместо того чтобы задавать значения параметров из программы, лучше сделать так, чтобы функция сама их спрашивала. Тогда тому, кто будет пользоваться этой программой, надо будет просто ввести значения с клавиатуры, а не менять программный код, передавая другие значения функции (можешь спросить, например, у мамы, что ей проще).

Используй функции `sys.stdin.readline` и `int`, чтобы считывать эти данные с клавиатуры внутри функции `calculate_interest`. Функцию теперь можно вызывать вот так:

```
calculate_interest()
```

Глава 7

Коротенькая глава про файлы

Может статься, ты уже слышал, что такое файлы (тем более, что в прошлой главе мы их использовали). Но я на всякий случай расскажу.

Наверное, тебе доводилось сталкиваться с такими плоскими полиэтиленовыми папками для бумаг на кольцах — они обычно называются файлами. В них можно вкладывать какие-то записи на русском, английском, удмуртском языках (например) и ещё приклеивать на них наклейку с названием. И потом их можно складывать в большие картонные папки, чтобы все эти файлики лежали рядом.

В компьютере всё примерно так же, только файлов можно создавать из ничего сколько угодно (ну... почти, пока место в компьютере не кончится). И у каждого файла всегда обязательно есть название. В файле может быть записано что угодно любой программой — мы, например, записывали в файлы текст простым текстовым редактором в предыдущих главах. И потом файл можно открыть любой программой — главное, чтобы программа, которая открывает файл, понимала тот «язык» (он называется «формат»), которым пользовалась программа, которая создавала файл. Файлы в компьютере можно складывать в папки, или директории[2]. Если аккуратно разложить все файлы в компьютере по папкам, то в них существенно проще ориентироваться, чем если они лежат кучей в одной папке или в беспорядке в разных папках.

Мы уже использовали объект, описывающий файл в Питоне, в предыдущей главе, таким образом:

```
>>> f = open('test.txt')
>>> print(f.read())
```

Так мы открываем для чтения файл «test.txt». Файл при этом находится в *текущей папке*. Когда ты запускаешь Питон из консоли, то текущая папка

[2]На самом деле, директория — тоже файл: в нём просто перечислен список, какие именно файлы лежат внутри.

остаётся той, которая была. Если ты не делал специальных манипуляций, то это просто домашняя папка. Если бы тебе хотелось напечатать файл «письмо.txt», который лежит в папке «почта», то надо было бы написать так:

```
>>> f = open ('почта/письмо.txt')
>>> print(f.read())
```

Файловый объект (он же объект типа `file`) не ограничивается только функцией `read`. В конце концов, если бы в папку с документами нельзя было добавить новых, мало пользы было бы от такой папки.

Мы можем создать новый пустой файл, передав ещё один параметр функции `open`, вот так:

```
>>> f = open('myfile.txt', 'w')
```

Параметр «w» — сокращение от английского «write» — записать. Так мы Питону говорим, что нам из этого файла не надо ничего читать, надо только записывать. Поэтому если файл с таким именем уже есть, то он будет тут же очищен, а если его нет — то будет создан новый пустой.

Можем что-нибудь записать в этот файл:

```
>>> f = open('myfile.txt', 'w')
>>> f.write('что-нибудь')
```

Теперь открой этот файл в текстовом редакторе. Файл будет пустой. Почему — мы выясним чуть позже.

Чтобы в файле появилось всё, что нужно, надо сказать Питону закрыть файл. Потому что к этому моменту он запомнил, что *надо будет* записать в файл `f` всё, что мы попросили, но ещё не записывал. Запоминать быстро, а записывать в файл — долго. По-настоящему Питон запишет в файл, только если программа завершится (то есть если закрыть консоль Питона, в которой мы открыли файл); или если помнить придётся слишком много — тогда какую-то часть Питон запишет в файл, а остальную будет всё ещё держать в уме; или если явно попросить Питон записать в файл всё, что нужно — например, функцией `close`:

```
>>> f = open('myfile.txt', 'w')
>>> f.write('что-нибудь')
>>> f.close()
```

Если теперь этот файл открыть текстовым редактором, то там будет записанный текст. Можно, опять-таки, воспользоваться Питоном для чтения текста:

```
>>> f = open('myfile.txt')
>>> print(f.read())
что-нибудь
```

После того как файл закрыт функцией `close`, в него больше нельзя ничего записать. Если потом захочется дописать что-то в конец этого файла, то надо его снова открыть, но вот так:

```
>>> f = open('myfile.txt', 'a')
```

Параметр «a» — сокращение от английского «append», то есть «добавить». Если теперь записывать в файл `f`, то всё записанное появится в конце файла после всего того, что там уже было.

Почему бы не записывать сразу?

Наверное, кажется странным, зачем такие хитрости с запоминанием и записыванием только потом. Простой ответ — чтобы было быстрее.

Дело в том, что записывать в файл может быть в тысячу раз медленнее, чем запоминать (в *оперативную память*). Чтобы хоть как-то это исправить, в файл пишут сразу большой кусочек данных за раз, так получается быстрее. Всё, что меньше, чем размер кусочка, Питон держит в памяти. Ещё бывает, что нужно что-то записать в файл, потом в другом месте программы это прочитать из файла, а потом файл и вовсе удалить за ненадобностью. Так вот если данных немного, то их можно и совсем не записывать в файл, а только запомнить, что сильно ускоряет работу программы. Вообще, подобным образом с файлами обращается не только Питон, но и большинство программ.

7.1. Как ещё развлечься

В этой главе мы обсудили, что же такое файл, наконец, а ещё как файлы читать и как в них записывать. Узнали, что Питон записывает в файл не сразу и почему он так делает: чтобы было быстрее.

Упражнение

Часто людям приходится копировать файлы, и ты наверняка с этим сталкивался — например, копируя файлы на флэшку, чтобы отнести в школу.

Напиши функцию `copy` с двумя параметрами: именем файла, который надо скопировать, и именем нового файла. Функция должна посчитать количество строк в файле и потом скопировать один файл в другой, строчка за строчкой, выводя после каждой строчки, сколько ещё осталось копировать.

Чтобы перебрать все строчки файла `f` одну за одной, надо использовать вот такой код:

```
for line in f:
... # очередная строчка файла записана в переменную line
```

Чтобы ещё раз перебрать все строки файла, надо его ещё раз открыть.

Функция `write` возвращает значение: количество записанных символов. Если оно печатается в консоль при выполнении функции, просто присваивай это значение какой-нибудь переменной (например, так: `w = newfile.write (line)`).

Чтобы проверить, работает ли функция, вызови её:

```
>>> copy ('myfile.txt', 'mynewfile.txt')
```

Открой файл «mynewfile.txt» в текстовом редакторе и проверь, что внутри у него то же самое, что и в «myfile.txt». Если нет, проверь, не забыл ли ты закрыть новый файл (`close`), чтобы Питон записал туда всё, что держит в уме.

Глава 8

Черепаховое изобилие

Пришло время вернуться к черепахе из модуля `turtle`, о которой мы говорили в главе 3. Чтобы создать холст, на котором бы рисовала черепаха, мы использовали вот такой код:

```
>>> import turtle
>>> t = turtle.Pen()
```

Раньше, когда мы не знали про циклы (`for` и `while`) и условный оператор (`if`), мы использовали самые простые функции для рисования. Вот так мы рисовали квадрат:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

Теперь этот код можно здорово сократить, если использовать цикл:

```
>>> t.reset()
>>> for x in range(1,5):
...     t.forward(50)
...     t.left(90)
...
```

Так куда меньше нужно набирать. К тому же, если читать этот код, то гораздо быстрее можно понять, что тут черепашка четыре раза идёт вперёд и поворачивает налево. Без цикла это не так сразу ясно.

Используя цикл, можно довольно просто изменить код так, чтобы рисовалась какая-нибудь фигура похитрее. Например, попробуй запустить вот этот код:

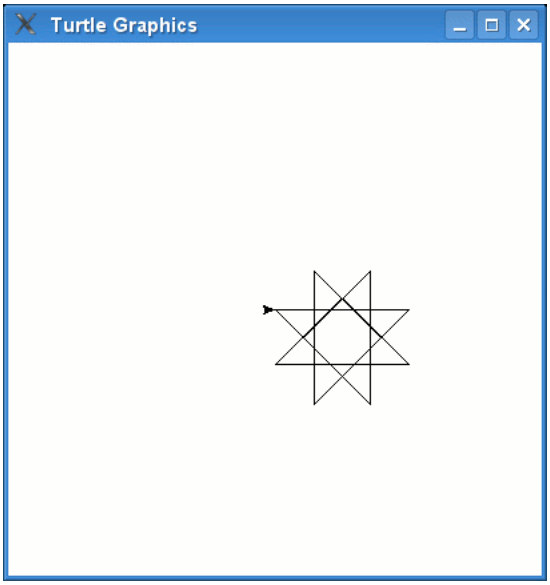


Рис. 8.1: Восьмиконечная звезда в исполнении черепашки.

```
>>> t.reset()
>>> for x in range(1,9):
...     t.forward(100)
...     t.left(225)
...
```

Он рисует восьмиконечную звезду, как на картинке 8.1. Каждый раз, когда черепашка проходит 100 пикселей, она поворачивает влево на 225 градусов.

Если ещё немного поменять угол (до 175 градусов) и сделать длиннее цикл (37 шагов), то можно нарисовать звезду, у которой будет ещё больше лучей (как на картинке 8.2):

```
>>> t.reset()
>>> for x in range(1,38):
...     t.forward(100)
...     t.left(175)
...
```

Или вот можно нарисовать что-то среднее между звездой и спиралью, как на рисунке 8.3:

```
>>> for x in range(1,20):
...     t.forward(100)
...     t.left(95)
...
```

Давай теперь попробуем нарисовать что-нибудь, что рисовать сложнее. Например, вот так:

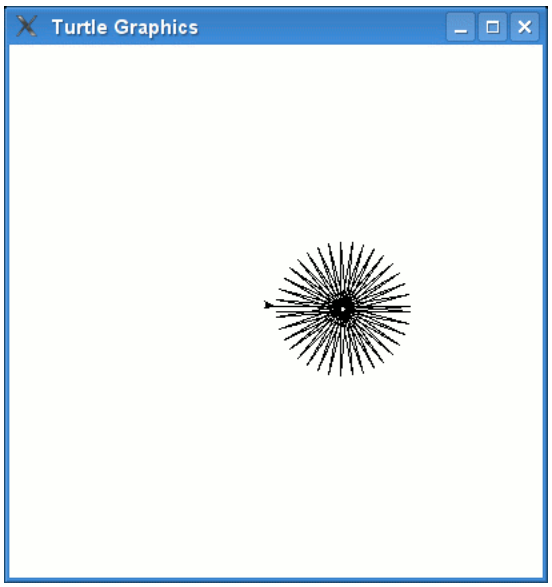


Рис. 8.2: Звезда, у которой много лучей.

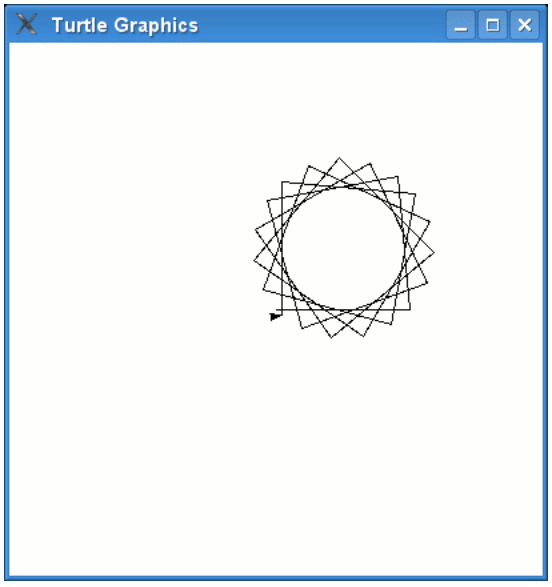


Рис. 8.3: Звезда с кучей лучей.

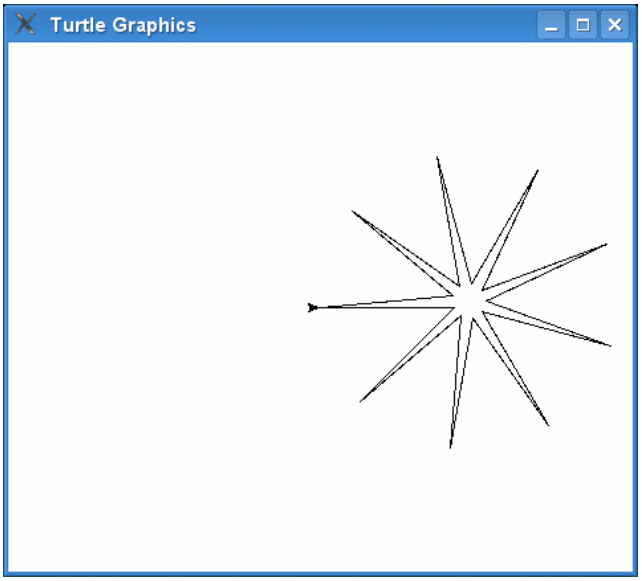


Рис. 8.4: Звезда с девятью лучами.

```
>>> t.reset()
>>> for x in range(1,19):
...     t.forward(100)
...     if x % 2 == 0:
...         t.left(175)
...     else:
...         t.left(225)
...
```

В этом коде мы проверяем, содержится ли в переменной `x` чётное число. Для этого нам пригодится операция «%», то есть получение остатка от деления: число ведь нечётное, если делится на два с остатком, и чётное — если без, то есть если остаток равен 0. Говоря по-питоньи, `if x % 2 == 0`, то `x` — чётное число.

Если запустить код, который написан выше, то получится звезда с девятью лучами, как на рисунке 8.4.

Умение черепашки рисовать не ограничивается просто чёрно-белыми линиями и способностью начинать и заканчивать рисовать. Можно делать и цветные картинки, например такие:

```
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

Код выше — долгий-долгий, утомительный и бессмысленный способ нарисовать что-то вроде машинки (она на картинке 8.5). Этот код ценен не машинкой. Он показывает, какие ещё функции есть у черепашки: `color` — меняет цвет фломастера, которым черепашка рисует; `fill` — закрашивает область на холсте всю одним цветом; `circle` — рисует кружочек заданного размера (диаметра).

8.1. Добавим цвета

Функция `color` принимает 3 параметра. Первый — интенсивность красного цвета, второй — зелёного, третий — синего.

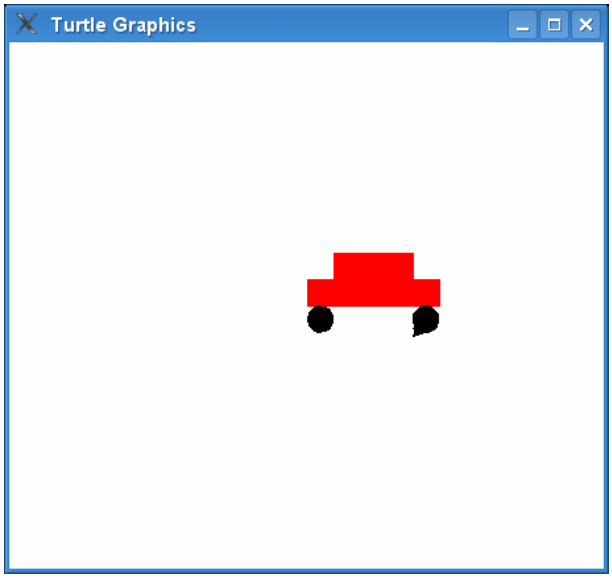


Рис. 8.5: Не стоит пытаться рисовать машину черепахой!

Почему красный зелёный и синий? И что это за интенсивность такая?

Если тебе доводилось смешивать краски, то ты можешь уже догадываться, как черепашка (и вообще, любая программа, рисующая на экране компьютера) получает цвета. Если ты смешаешь краски двух цветов, то получишь третий цвет, которым можно что-нибудь нарисовать[1]. Так, если смешать синюю и красную краски, то получится сиреневая, а если намешать слишком много красок, то может получиться серо-буро-малиновый цвет или просто какая-то коричневая грязька. В компьютере можно так же смешивать цвета: если взять одинаковое количество красного и зелёного, например, то получится жёлтый. Но в компьютере мы смешиваем цвета, которыми светит сам экран, а не цвета краски, которой мы рисуем по бумаге.

Давай на секундочку вернёмся к смешиванию краски. Представь, что у нас есть три чашки с краской. Одна чашка с красной, другая — с зелёной и третья — с синей. Каждая чашка полна краски до краёв. Так вот, если мы хотим взять целую чашку краски для получения цвета, мы должны для этого цвета передать черепашке значение 1 (то есть 100%). Если краску из этой чашки мы брать не хотим — то значение 0. Если хотим взять полчашки, например, — то 0.5 (то есть $\frac{1}{2}$). Если взять целую чашку красной и зелёной краски, то в результате получится жёлтая. Давай нарисуем жёлтый круг:

[1]Когда рисуешь красками, то нужно чтобы были красный, жёлтый и синий цвета, из них можно получить любой другой. Компьютеру нужны красный, зелёный и синий, потому что на экране он рисует не красками, а наоборот, светом.

```
>>> t.color(1,1,0)
>>> t.begin_fill()
>>> t.circle(50)
>>> t.end_fill()
```

Тут мы вызываем функцию `color`, передавая ей значение 1 (100%) для красного и зелёного цвета и 0 — для синего. Давай для дальнейших экспериментов с цветом заведём функцию, которая будет нам рисовать тем цветом, которым мы попросим:

```
>>> def draw_circle(red, green, blue):
...     t.color(red, green, blue)
...     t.begin_fill()
...     t.circle(50)
...     t.end_fill()
...
```

Можем нарисовать ярко-зелёный круг, взяв целую чашку зелёной краски.

```
>>> draw_circle(0, 1, 0)
```

А можем взять только полчашки зелёной краски...

```
>>> draw_circle(0, 0.5, 0)
```

...и круг получится тёмно-зелёным. Это может показаться странным: сколько краски ни возьми в реальном мире, круг всё равно будет одинаково зелёным. Но хитрость в том, что на самом деле нет. Если нарисовать тот же круг на белой бумаге, взяв мало краски, то сквозь круг будет просвечивать белая бумага, и круг будет белее, чем цвет самой краски. А компьютер, наоборот, рисует светом на чёрном экране. Мы сказали ему взять 0.5 чашки краски, то есть «половинное количество краски», и круг стал темнее, как будто сквозь него просвечивает чёрный экран.

Можешь понаблюдать за таким же эффектом, если рисовать синим и красным цветом: круг будет ярче и светлее, если «брать целую чашку краски», то есть передавать 1 в качестве значения интенсивности цвета:

```
>>> draw_circle(1, 0, 0)
>>> draw_circle(0.5, 0, 0)

>>> draw_circle(0, 0, 1)
>>> draw_circle(0, 0, 0.5)
```

Используя разные сочетания красного, зелёного и синего цветов, можно получить очень много цветов, почти все, которые человек может различить. Например, если смешать 100% красного света, 85% зелёного, и не брать синего, то получится золотистый свет:

```
>>> draw_circle(1, 0.85, 0)
```

А малиновый можно получить, взяв 100% красного, 70% зелёного и 75% синего:

```
>>> draw_circle(1, 0.70, 0.75)
```

Оранжевый — взяв 100% красного и 65% зелёного; коричневый — 60% красного, 30% зелёного и 15% синего:

```
>>> draw_circle(1, 0.65, 0)
>>> draw_circle(0.6, 0.3, 0.15)
```

Для того, чтобы узнать, сколько красного, синего и зелёного в нужном цвете, есть специальные программы-палитры. Например, kcolorchooser.

И я просто напомню, что всегда можно очистить холст, написав `t.clear()`.

8.2. Темнота

Ты же знаешь, что произойдёт если ночью выключить весь свет? Всё станет чёрным.

Так и в компьютере происходит. Если не брать никакого света для рисования, то рисунок будет чёрным (цвета выключенного монитора):

```
>>> draw_circle(0, 0, 0)
```

Этот код рисует чёрный круг, как на картинке 8.6.

А если взять для рисования весь свет, то есть 100% красного, 100% зелёного и 100% синего, то получится белый. Вот этот код уберёт обратно в небытие чёрный кружок (закрасит его белым):

```
>>> draw_circle(1,1,1)
```

8.3. Закрашиваем рисунки

Возможно, ты уже догадался, что функция `begin_fill` включает заливку цветом, а `end_fill` — выключает. На самом деле, именно в момент выключения Питон и заливает нужную часть рисунка, а до тех пор только запоминает, куда потом лить краску. Зная это, мы можем нарисовать закрашенный квадрат. Вот код, который рисует просто квадрат:

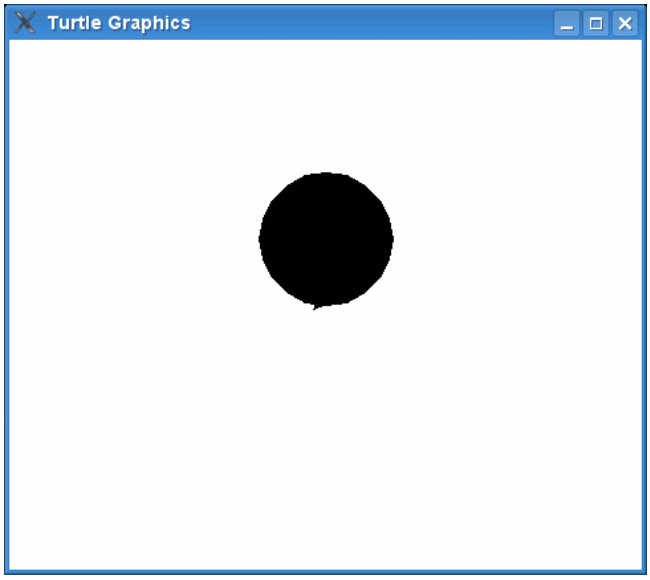


Рис. 8.6: Чёрная дыра!

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

Чтобы было проще рисовать много квадратов, да ещё и разного размера, завернём этот код в функцию:

```
>>> def mysquare(size):
...     t.forward(size)
...     t.left(90)
...     t.forward(size)
...     t.left(90)
...     t.forward(size)
...     t.left(90)
...     t.forward(size)
...     t.left(90)
```

Всё ли вышло успешно? Давай проверим.

```
>>> mysquare(50)
```

У меня квадрат нарисовался.

...в воображении. В книге нет консоли питона, но даже у книги есть воображение.

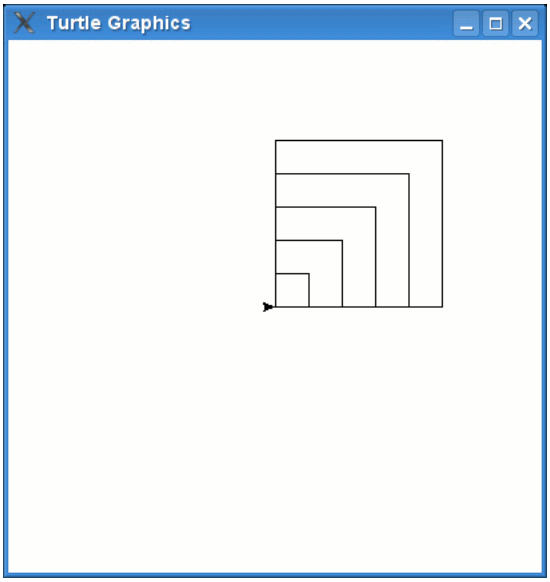


Рис. 8.7: Квадратики. Много.

Если присмотреться к этому коду, можно заметить, что в нём кучу раз повторяются одни и те же две строки. А если ещё пристальнее присмотреться, то даже удастся сосчитать, сколько раз они повторяются: четыре. Так что тут хорошо бы смотрелся цикл — вот такой:

```
>>> def mysquare(size):
...     for x in range(0,4):
...         t.forward(size)
...         t.left(90)
```

Можешь попробовать нарисовать квадратов всякого разного размера:

```
>>> t.reset()
>>> mysquare(25)
>>> mysquare(50)
>>> mysquare(75)
>>> mysquare(100)
>>> mysquare(125)
```

Должно получиться что-то вроде рисунка 8.7.

Ну а теперь можно приступить к рисованию закрашенных квадратов. Прежде всего, надо очистить холст от тех шедевров, что на нём уже есть:

```
>>> t.reset()
```

Потом включить заливку и нарисовать квадрат:

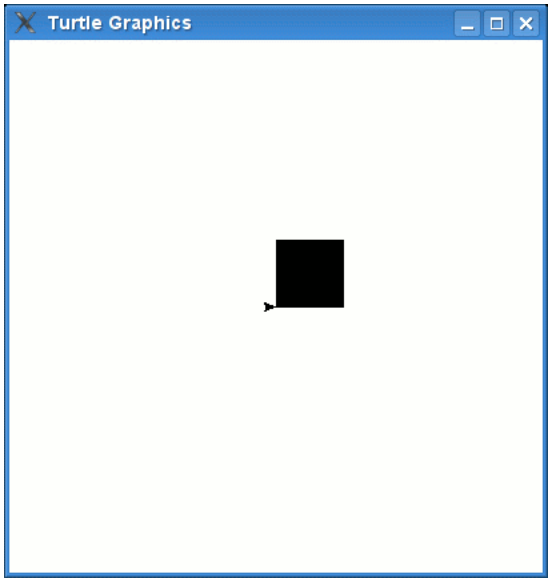


Рис. 8.8: Чёрный квадрат.

```
>>> t.begin_fill()
>>> mysquare(50)
```

...и потом наконец закрасить нарисованное:

```
>>> t.end_fill()
```

В результате получается картинка 8.8.

Можно сделать рисование закрашенных квадратов чуточку удобнее: добавить в функцию рисования квадрата параметр, в зависимости от которого квадрат будет получаться или закрашенным, или нет:

```
>>> def mysquare(size, filled):
...     if filled == True:
...         t.begin_fill()
...     for x in range(0,4):
...         t.forward(size)
...         t.left(90)
...     if filled == True:
...         t.end_fill()
...
```

Первые две строки проверяют, равна ли переменная `filled` «истине» (`True` по-английски), то есть просили ли нас закрашивать. Если да, то включается заливка цветом. Потом рисуется квадрат, потом заливка завершается, если нас вообще просили закрашивать. Теперь закрашенный квадрат можно нарисовать так:

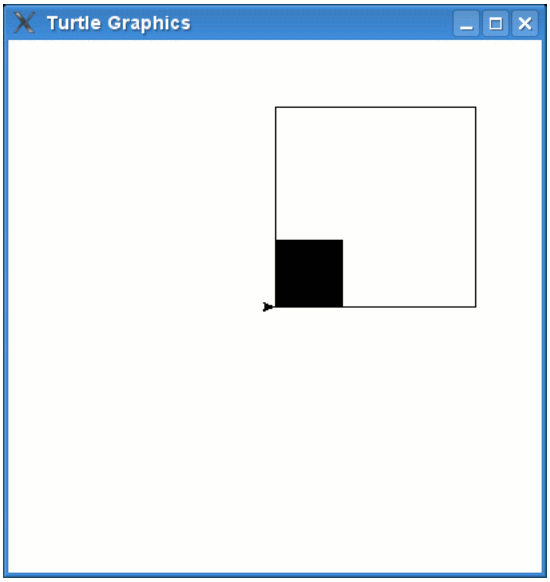


Рис. 8.9: Квадратный глаз

```
>>> mysquare(50, True)
```

...а не закрашенный — вот так:

```
>>> mysquare(150, False)
```

И результат получается, как на рисунке 8.9... мне он напоминает безумный квадратный глаз.

Примерно так можно рисовать и всё остальное, что в голову взбредёт, и закрашивать цветом. Например, можем сделать функцию рисования звезды, которую мы в начале главы рисовали:

```
>>> for x in range(1,19):
...     t.forward(100)
...     if x % 2 == 0:
...         t.left(175)
...     else:
...         t.left(225)
...
```

Заворачиваем это в функцию, добавляем параметр, указывающий, надо ли закрашивать звезду (и ещё размер добавим, как для квадрата тоже), и получаем такой код:

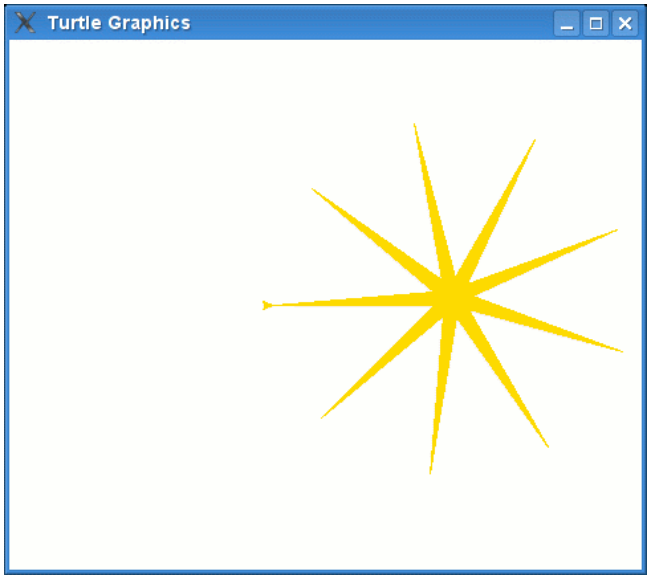


Рис. 8.10: Золотая звезда.

```
1.  >>> def mystar(size, filled):
2.  ...     if filled:
3.  ...         t.begin_fill()
4.  ...     for x in range(1,19):
5.  ...         t.forward(size)
6.  ...         if x % 2 == 0:
7.  ...             t.left(175)
8.  ...         else:
9.  ...             t.left(225)
10. ...     if filled:
11. ...         t.end_fill()
```

Давай теперь проверим, что эта функция на самом деле делает: попробуем нарисовать золотистую звезду:

```
>>> t.color(1, 0.85, 0)
>>> mystar(120, True)
```

Черепашка должна изобразить рисунок 8.10. Теперь давай добавим к этой звезде чёрный контур: поменяем цвет, выключим заливку и нарисуем звезду того же размера ещё раз:

```
>>> t.color(0,0,0)
>>> mystar(120, False)
```

Теперь звезда должна выглядеть так: 8.11.

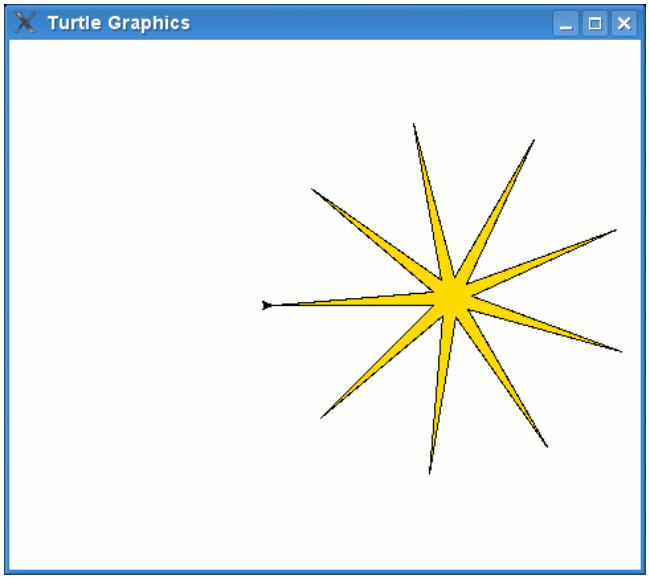


Рис. 8.11: Звёздочка с контуром.

8.4. Как ещё развлечься

В этой главе мы посмотрели на разные функции черепашки — модуля turtle Питона — и с её помощью нарисовали некоторые геометрические фигуры. Мы создали функции для рисования фигур, а также для закраски их цветом.

Упражнение 1

Ну хорошо, вот мы нарисовали звёздочки, квадратики, прямоугольники. А как насчёт восьмиугольника? Попробуй нарисовать его. (Поворачивай на 45 градусов, это ключ к успеху).

Упражнение 2

Создай функцию для рисования закрашенных и не закрашенных восьмиугольников разного размера.

Глава 9

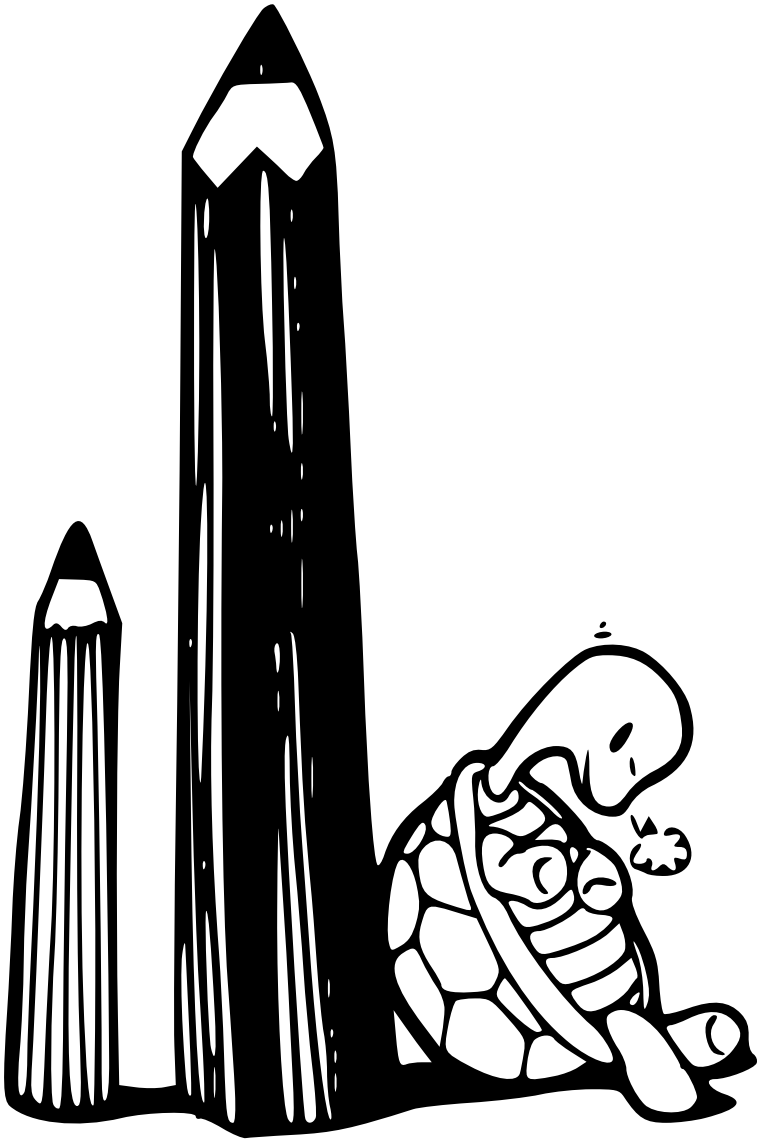
A bit graphic

Когда просишь черепашку что-нибудь нарисовать, сталкиваешь с одной проблемой. Дело в том... что черепашка... ... очень... ... ... медленная.

Даже если попросить черепашку рисовать так быстро, как возможно, она всё равно будет делать это медленнее, чем хотелось бы. Для черепах вообще это не проблема — у них полно времени, — но вот рисовать на экране нужно быстрее, чтобы изображение не дёргалось и не замирало. Вспомни на секунду, какие бывают игры для компьютера или для переносных устройств типа Gameboy, Nintendo; для мобильных телефонов. У них бывает двухмерная графика: какие-нибудь плоские фигурки, которые движутся по плоскому полю — именно такая графика у большинства игр для телефонов, например. Бывает псевдо-трёхмерная графика: когда фигурки и поле нарисованы не плоскими, а как будто мы их видим сверху сбоку, под углом. Двигаются при этом они тоже по плоскому полю, и такие игры почти и не отличаются от двухмерных. А бывают игры, в которых изображение похоже на реальный трёхмерный мир — обычно это игры для компьютера, но и в телефонах бывают.

У всех этих видов графики общее одно: рисовать на экране компьютера надо очень быстро. Сейчас объясню, почему. Вот представь, что ты рисуешь на бумаге свой маленький мультфильм. Для этого нужна стопка чистой бумаги — ну и карандаш, например. На первом листочке из стопки сбоку надо нарисовать какую-нибудь картнику — допустим, кота. На втором листочке — точно такую же картинку, но чуть пошевелившую хвостом. На следующей картинке хвост кота должен двинуться ещё дальше — и так далее. Потом надо держать эту стопку бумаги за тот край, где нет рисунков, а другой рукой пролистывать быстро стопку, и тогда будет казаться, что кот в самом деле шевелит хвостом. Именно так работает вся анимация: фильмы, мультики, игры в компьютере. Быстро-быстро (по крайней мере 24 раза в секунду) одна картинка, или один кадр, сменяет другой, похожий. Черепашка с такими скоростями рисовать просто не умеет.

Трёхмерная графика рисуется похожим образом, хотя и по-другому. Там точно так же часто-часто сменяются кадры, но вот что именно должно быть нарисовано в каждом кадре высчитывается отдельно (достаточно хитрым способом). К тому моменту, как черепашка бы только посчитала, что нужно нарисовать в очередном кадре, уже было бы пора рисовать следующий.



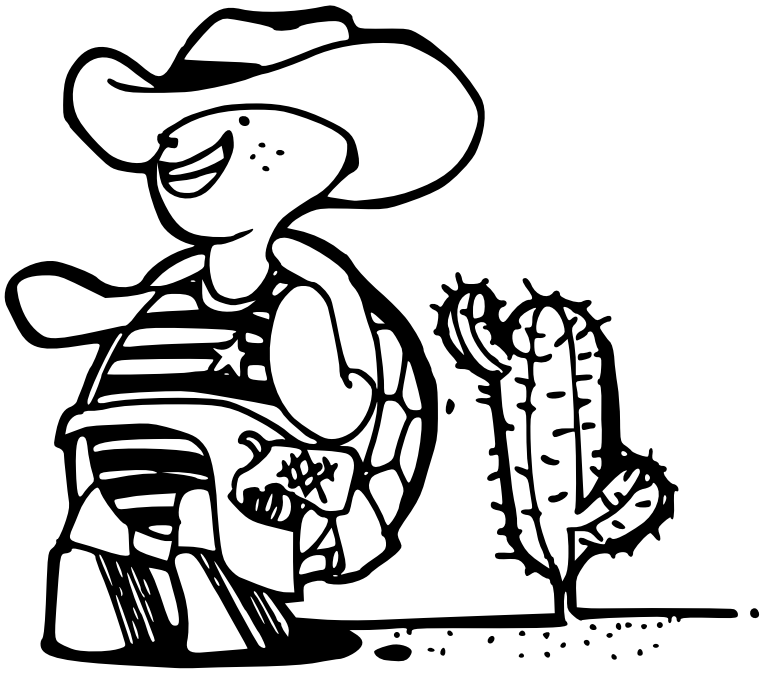
9.1. Quick Draw

В каждом языке программирования есть свой способ рисовать на экране. Некоторые способы позволяют рисовать быстро, некототрые — медленно, так что программисты, которые пишут игры, пользуются только подходящими языками программирования.

В Питоне можно рисовать несколькими способами (включая ту же черепашку), но быстрее всего рисуют на экране библиотеки, которые не устанав-

ливаются сразу вместе с Питоном (потому что они нужны совсем не всем). Я про них не буду рассказывать, потому что они сложные, и только после пары лет программирования будет вообще понятно, как ими пользоваться.

К счастью, с Питоном устанавливается модуль, который вполне можно использовать, чтобы рисовать на экране несложные элементы управления — например, кнопки и окна. Пожалуй даже, этот способ рисует настолько быстро, что его можно назвать быстрой черепашкой.



Модуль этот называется `tkinter` (сокращение от 'Tk interface', что бы это ни значило). При помощи этого модуля можно создавать полноценные программы (например, вполне получится создать редактор текста), и ещё можно рисовать несложные картинки. Сложные тоже можно, но это долго программировать, и они будут медленно рисоваться на экране.

Можно создать программу, которая рисует на экране кнопку, вот так:

```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> btn = Button(tk, text='жми сюда')
4. >>> btn.pack()
```

В первой строчке мы говорим Питону импортировать всё из модуля `Tk`, чтобы использовать оттуда потом функции. Потом, во второй строке, мы тут же и используем одну функцию: `Tk()`, она нам создаёт окно, которое мы сохраняем в переменную `tk` и которое тут же появляется на экране. В третьей строке мы создаём кнопку функцией `Button`, она тоже из модуля `Tk`, и её записываем в переменную `btn`. Функции создания кнопки мы говорим, для

какого окна её создавать и что должно быть написано на кнопке (для этого мы передаём *именованный параметр* `text`). И наконец, в четвёртой строке мы отправляем кнопку в её окно, где она и появляется, а окно уменьшает свой размер до размеров кнопки.

Именованные параметры и значения по умолчанию

Здесь мы первый раз использовали *именованные параметры*. Эти параметры отличаются от обычных тем, что их можно указывать в любом порядке, и при этом нужно писать имя параметра.

Возьмём, например, функцию, которая рисует прямоугольник:

```
def draw_rectangle (length, width):
    for i in [0, 1]:
        t.forward(length)
        t.left(90)
        t.forward(width)
        t.left(90)
```

Теперь представим, что чаще всего нам нужно рисовать квадраты шириной 50 пикселей. Тогда мы можем объявить нашу функцию вот как:

```
def draw_rectangle (length = 50, width = 50):
....
```

Дальше всё остаётся по-прежнему. Такую новую функцию можно вызывать вообще без параметров:

```
draw_rectangle()
```

Тогда и длина, и ширина будут равны 50 пикселям.

Можно указать только длину, оставив ширину равной 50 пикселям:

```
draw_rectangle(100)
```

А можно указать только ширину, оставив длину равной 50 пискелям. Как? Как раз с помощью именованных параметров функции:

```
draw_rectangle(width = 100)
```

Можно указать длину и ширину в обратном порядке:

```
draw_rectangle(width = 100, length = 150)
```

У функции Button из модуля Tk, которую мы вызвали выше, много параметров и почти все имеют значение по умолчанию. Чтобы указать новые значения только тем параметрам, которые нам нужны, мы и используем именованные параметры функций.

Результат запуска нашего кода выглядит как-то так:

Кнопка не делает ничего, но по крайней мере, появилась на экране и нажимается. Можно для начала сделать чтобы что-нибудь печаталось в консоль после каждого нажатия кнопки. Для этого нам понадобится функция — например, такая:

```
>>> def hello():
...     print('Кто-то потрогал кнопку')
```

Теперь надо закрыть окно с кнопкой и сделать новое, с другой кнопкой. Этой другой кнопке нужно объяснить, что по нажатию на неё надо вызывать функцию hello:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="жми сюда", command=hello)
>>> btn.pack()
```

В качестве параметра «command» мы передаём кнопке функцию, которую надо вызвать по нажатию. Можешь нажать на кнопку и убедиться, что она честным образом каждый раз пишет в консоль, что её нажали.

9.2. Немного порисуем

При помощи одних лишь кнопок не получится нарисовать на экране что угодно, поэтому Tk есть и другой элемент: холст (canvas), почти как у черепашки. При создании холста надо указать его ширину и высоту, вот так:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
```

Так же, как в предыдущем примере, после второй строки появится окно. А после четвёртой строки с вызовом функции pack оно увеличится в размерах.

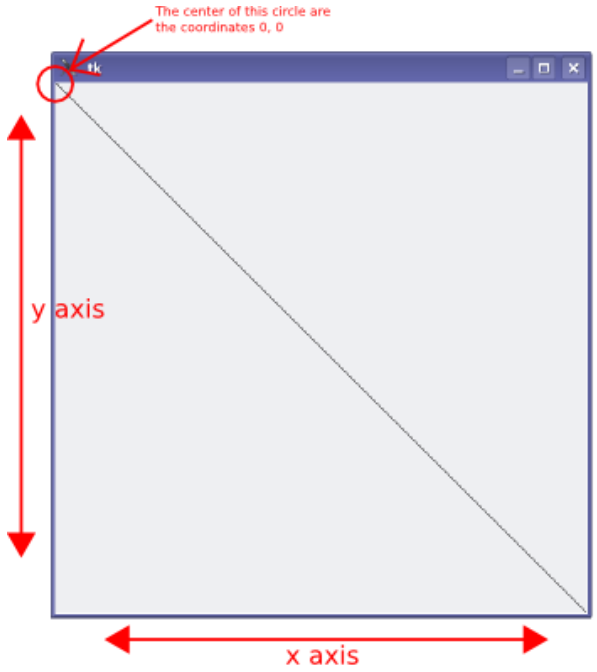


Рис. 9.1: Воображаемые оси координат на холсте

Теперь можно рисовать в этом окне на холсте, используя координаты. У каждой точки две координаты. Первая из них говорит, сколько пикселей от точки до левой границы холста, а вторая координата — сколько пикселей до верхней границы. Эти координаты называют x и y («икс» — до левой границы — и «игрек» — до верхней границы).

Мы сделали холст размером 500 пикселей шириной и 500 высотой, так что координаты его правого нижнего угла — 500 и 500. Линию, которая есть на рисунке 9.1, можно нарисовать, указав координаты начальной точки: 0, 0 — и конечной точки: 500, 500:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 500, 500)
```

У каждой точки всегда первая координата — по горизонтали (x), а вторая — по вертикали (y).

Если бы мы хотели добиться того же самого от черепашки, пришлось бы говорить ей гораздо больше всего:

```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> t = turtle.Pen()
>>> t.up()
>>> t.goto(-250,250)
>>> t.down()
>>> t.goto(500,-500)
```

Рисовать линию при помощи tkinter уже проще, чем прибегая к услугам черепашки. А помимо рисования линий tkinter может исполнять для нас много разных других функций, про некоторые из которых мы сейчас поговорим.

9.3. Нарисуем прямоугольники

Когда мы объясняли черепашке, что нам нужен на экране прямоугольник, мы говорили ей продвинуться вперёд, повернуться, ещё продвинуться, ещё повернуться — и так далее. В результате всего этого получался квадрат или прямоугольник — смотря на сколько мы говорили черепашке двигаться вперёд. tkinter'у нам нужно просто сказать, что мы хотим видеть прямоугольник, и задать координаты его углов:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400,height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 50)
1
```

Здесь мы создали квадратный холст размером 400×400 пикселей и нарисовали на нём квадрат. Левый верхний угол этого квадрата имеет координаты (10, 10), то есть от него 10 пикселей до левой и до верхней границ холста. Правый нижний — координаты (50, 50).

Функция create_rectangle напечатала нам на экран число 1: это номер фигуры на холсте, которую мы только что нарисовали. Потом нам пригодятся такие номера.

Функции create_rectangle мы передали четыре параметра. Первая пара параметров — координаты левого верхнего угла прямоугольника, а вторая пара — координаты правого нижнего. Прямоугольники все получаются с горизонтальными и вертикальными сторонами, их нельзя повернуть, используя только эту функцию. Координаты левой верхней точки обычно обозначают как «x1» (горизонтальная координата) и «y1» (вертикальная координата). Координаты правой нижней точки обычно обозначают как «x2» и «y2».

Если мы увеличим координату x2, то получится прямоугольник:

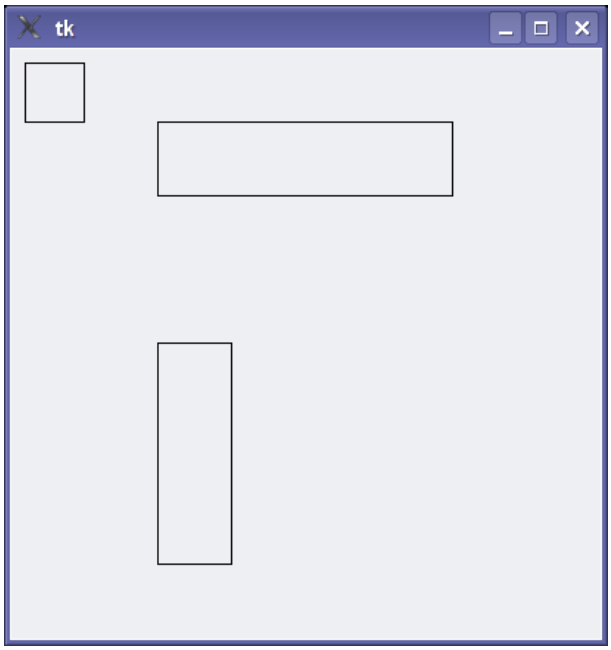


Рис. 9.2: Прямоугольнички от tkinter.

```
>>> canvas.create_rectangle(100, 100, 300, 50)
```

Можно вытянуть квадрат в другую сторону, получится другой прямоугольник:

```
>>> canvas.create_rectangle(100, 200, 150, 350)
```

Последняя из команд дословно значит вот что: отступи на 100 точек вправо от границы холста (левой верхней), и 200 точек вниз. Потом нарисуй прямоугольник шириной 150 точек направо и высотой 350 точек вниз. В результате выполнения этой команды холст должен выглядеть, как на рисунке 9.2.

Если теперь тебе в какой-то момент захочется очистить рисунок (то есть удалить всё с него), используй такую команду:

```
>>> canvas.delete('all')
```

Можем теперь попробовать нарисовать геометрическую абстрактную картинку, заполнив холст кучей разных прямоугольников. Для этого нам пригодится модуль `random`. Вначале его надо импортировать:

```
>>> import random
```

Кроме того, для удобства создадим функцию, которая будет нам рисовать прямоугольник случайного размера в случайном месте. Для этого потребуется вызвать функцию `randrange`:

```
>>> def random_rectangle(width, height):
...     x1 = random.randrange(width)
...     y1 = random.randrange(height)
...     x2 = random.randrange(x1 + random.randrange(width))
...     y2 = random.randrange(y1 + random.randrange(height))
...     canvas.create_rectangle(x1, y1, x2, y2)
```

В первых двух строчках функции мы создаём переменные, определяющие левый верхний угол будущего прямоугольника. Функция `randrange` возвращает нам число от 0 до числа, которое мы ей передали как параметр (не включая это число). Она умеет ещё несколько других параметров принимать, подробнее об этом написано в приложении **??**. Если мы вызовем `randrange(10)`, то получим какое-то число от 0 до 9. Если вызовем `randrange(100)` — какое-то число от 0 до 99.

В следующих двух строках мы таким же образом определяем координаты правого нижнего угла нашего прямоугольника (может, это даже будет квадрат, кто знает): добавляем случайные числа к каждой из координат левого верхнего угла. После всего этого мы наконец создаём прямоугольник при помощи функции `create_rectangle`.

Можешь попробовать, как эта функция работает:

```
>>> random_rectangle(400, 400)
```

Можно ещё сделать сотню разных прямоугольничков:

```
>>> for x in range(0, 100):
...     random_rectangle(400, 400)
```

Получается куча линий, что-то вроде того, что на рисунке 9.3.

Ты можешь помнить, что в последней главе мы говорили черепашке, каким цветом рисовать, задавая долю каждого из трёх цветов: красного, синего и зелёного. Используя `tkinter`, можно задать цвет примерно так же, но записав это по-другому. Прежде всего, давай научим нашу функцию рисовать разноцветно:

```
>>> def random_rectangle(width, height, fill_colour):
...     x1 = random.randrange(width)
...     y1 = random.randrange(height)
...     x2 = random.randrange(x1 + random.randrange(width))
...     y2 = random.randrange(y1 + random.randrange(height))
...     canvas.create_rectangle(x1, y1, x2, y2, fill=fill_colour)
```

Функции `create_rectangle` можно передать параметр `fill`, значение которого — цвет, которым заполнять прямоугольник. Теперь можно сделать разноцветный беспорядок:

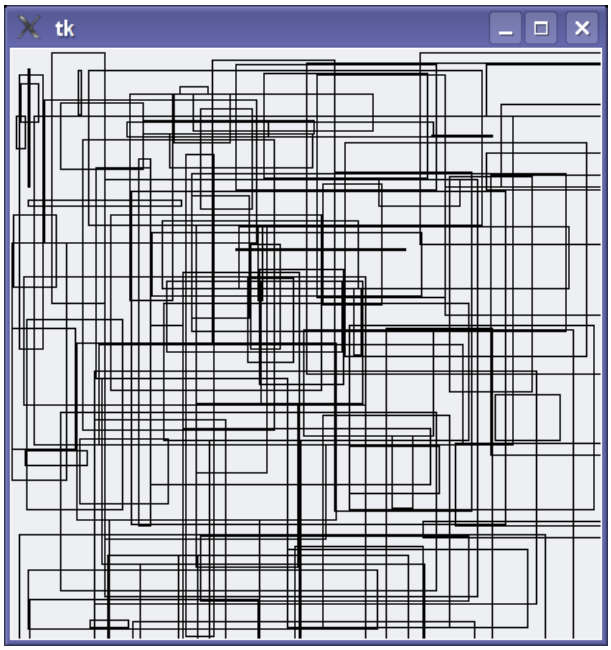


Рис. 9.3: Прямоугольный беспорядок.

```
>>> random_rectangle(400, 400, 'green')
>>> random_rectangle(400, 400, 'red')
>>> random_rectangle(400, 400, 'blue')
>>> random_rectangle(400, 400, 'orange')
>>> random_rectangle(400, 400, 'yellow')
>>> random_rectangle(400, 400, 'pink')
>>> random_rectangle(400, 400, 'purple')
>>> random_rectangle(400, 400, 'violet')
>>> random_rectangle(400, 400, 'magenta')
>>> random_rectangle(400, 400, 'cyan')
```

Не пугайся, если некоторые функции не выполнились, а только напечатали сообщение об ошибке. Какие именно цвета `tkinter` узнаёт по названию, зависит от операционной системы (Windows, Mac OS, Linux) и от версии этого пакета. Тут-то и встаёт вопрос: как нарисовать цвет, названия которого `tkinter` не знает? Примерно так же, как мы просили этого от черепашки: сказать, сколько в этом цвете красного, зелёного и синего. Только `tkinter` ждёт этого записанного в другом виде. Например, золотой цвет (100% красного, 85% зелёного и 0% синего) задаётся вот так:

```
>>> random_rectangle(400, 400, '#ffd800')
```

Эти цифры и буквы после решёточки — шестнадцатеричные[1]. Первые две цифры — ff — соответствуют красному (максимальное значение для него), вто-

[1]Шестнадцатеричных цифр 16, по порядку от 0 до 9 и потом a, b, c, d, e, f.

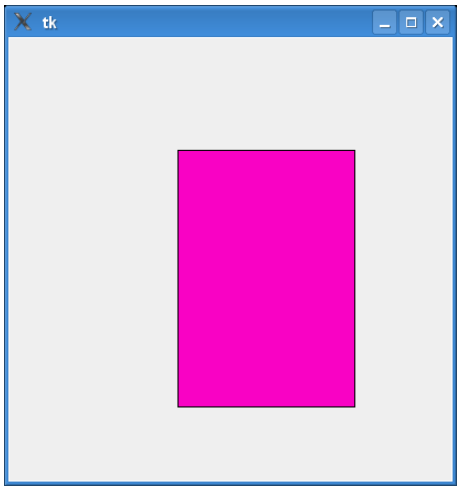


Рис. 9.4: Сиреневый прямоугольник.

рые две — d8 — зелёному, а последние две — 00 — синему (его нет). Разбираться в этом всём нет большой необходимости, потому что можно просто написать функцию, которая будет нам выдавать строки, понятные tkinter'у из того, что понятно нам. Функция эта будет выглядеть вот так:

```
>>> def hexcolor(red, green, blue):
...     red = 255*(red/100.0)
...     green = 255*(green/100.0)
...     blue = 255*(blue/100.0)
...     return '#%02x%02x%02x' % (red, green, blue)
```

Если попросить её закодировать золотистый цвет, то получится...

```
>>> print(hexcolor(100, 85, 0))
#ffd800
```

... как раз то, что надо.

Или можно попросить её закодировать сиреневый цвет (98% красного, 1% зелёного, 77% синего):

```
>>> print(hexcolor(98, 1, 77))
#f902c4
```

Теперь этот цвет можно передавать в нашу функцию random_rectangle:

```
>>> random_rectangle(400, 400, hexcolor(98, 1, 77))
```

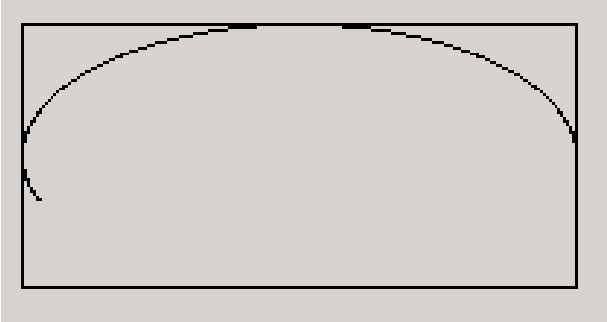


Рис. 9.5: Дуга в прямоугольнике.

9.4. Нарисуем дугу

Дуга — это часть эллипса (ну или круга — круг тоже является эллипсом), но чтобы нарисовать её при помощи `tkinter`, надо задать координаты прямоугольника — левого верхнего и правого нижнего угла, как и раньше. Это может показаться странным, но смысл тут вот в чём: мы задаём прямоугольник, в который вписан эллипс, чью дугу нам надо нарисовать, как это нарисовано на картинке 9.5. Помимо прямоугольника мы говорим ещё, какую часть дуги рисовать: 359 — значит весь эллипс, 180 — значит половину эллипса, а 0 — совсем ничего.

Чтобы нарисовать картинку 9.5, подойдёт вот такой код:

```
canvas.create_arc      (10, 10, 200, 100, extent=200, style=ARC)
canvas.create_rectangle(10, 10, 200, 100)
```

Параметры функции `create_arc` такие: сначала мы задаём четыре координаты прямоугольника (так же, как обычно); потом мы говорим, сколько градусов дуги нам надо рисовать (от 0 до 359); потом мы говорим, что мы хотим видеть только саму дугу[1]. Ниже для примера нарисованы дуги с разным количеством градусов, результат — на рисунке 9.6:

```
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```

[1] Другие варианты — `style=PIESLICE` и `style=CHORD`.

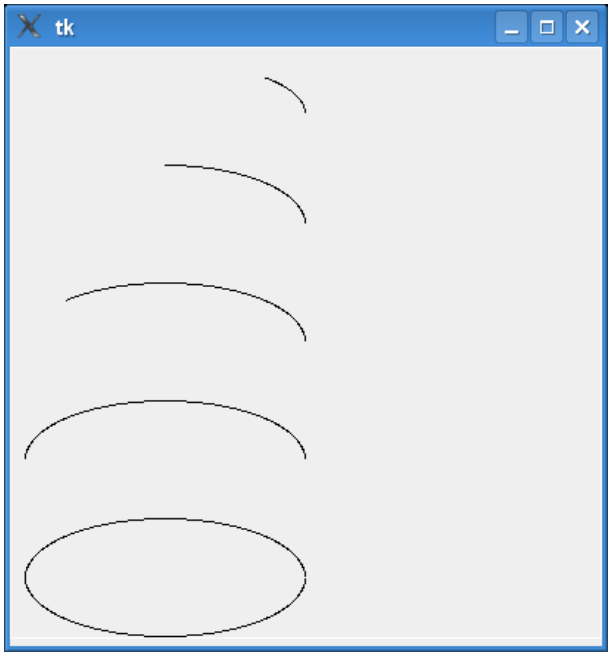


Рис. 9.6: Дуги разной длины.

9.5. Нарисуем эллипс

Последняя из команд в предыдущем примере рисует аккурат целый эллипс (он же овал[1]). Чтобы рисовать овалы, есть и специальная команда, покороче: `create_oval`. Так же, как и для рисования дуги, для рисования овала надо задать ограничивающий прямоугольник. Например, вот такой:

```
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400,height=400)
>>> canvas.pack()
>>> canvas.create_oval(1, 1, 300, 200)
```

Этот пример рисует овал в воображаемом прямоугольнике который занимает пространство от точки (1, 1) до точки (300, 200). Если мы теперь нарисуем прямоугольник с теми же координатами, то увидим, что овал как раз аккуратно в него вписан (и получится у нас рисунок 9.7):

```
>>> canvas.create_rectangle(1, 1, 300, 200, outline=''#ff0000'')
```

Чтобы нарисовать окружность[1], нам нужно в качестве ограничивающего прямоугольника задать квадрат (как на картинке 9.9):

[1]На самом деле, эллипсом называют саму линию, а овалом — то, что она ограничивает.

[1]Тут как с овалом: окружность — это линия, а круг можно вырезать из бумаги по этой линии — это всё, что внутри линии.

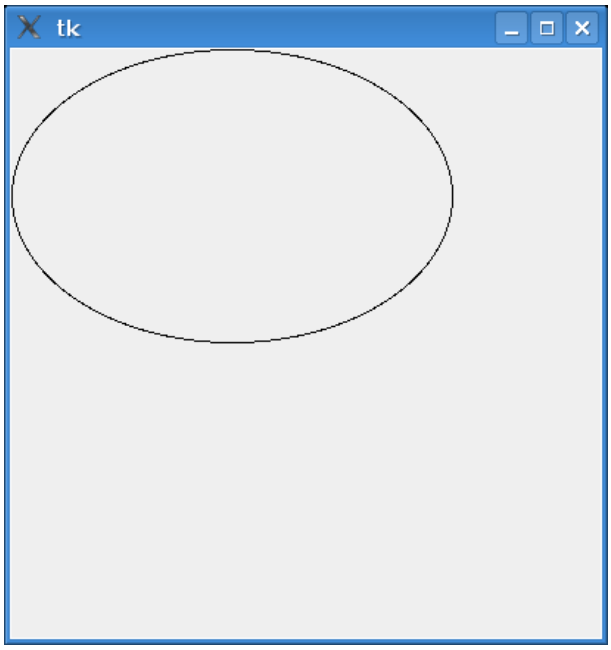


Рис. 9.7: Эллипс

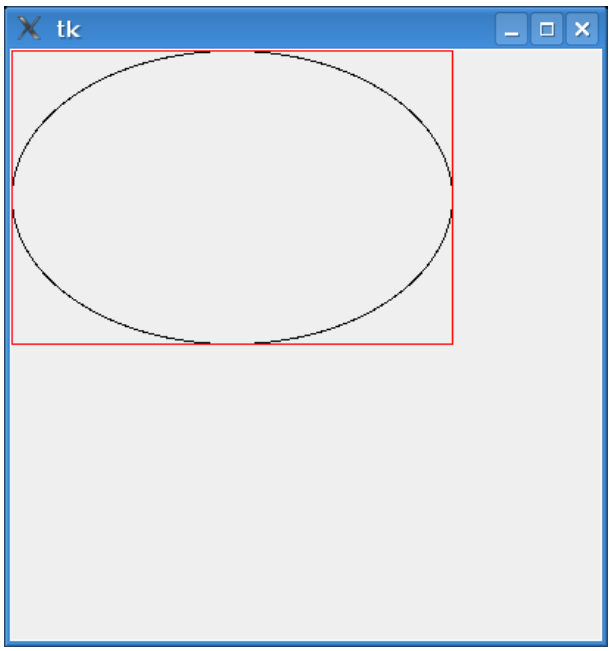


Рис. 9.8: Эллипс в рамочке.

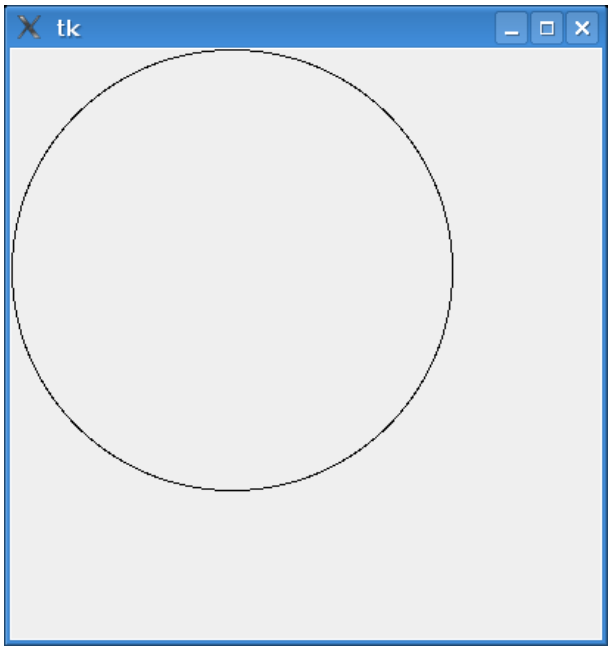


Рис. 9.9: Окружность

```
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400,height=400)
>>> canvas.pack()
>>> canvas.create_oval(1, 1, 300, 300)
```

9.6. Нарисуем многоугольник

Многоугольник — это какая-то геометрическая фигура с углами, у которой больше трёх сторон. Треугольники, прямоугольники, пятиугольники — это всё примеры многоугольников. Помимо этих правильных форм, можно создавать и более неуклюжие фигуры. Для этого есть функция `create_polygon`. Ей нужно передать столько пар координат, сколько вершин у нашего многоугольника, а она соединит все эти вершины и, если попросить, закрасит получившуюся фигуру.

Начнём с треугольника. У него три вершины, у каждой по две координаты; закрашивать его мы не хотим; а линии пусть рисуются чёрными. Всё это на питоньем языке выглядит так (а в результате получается рисунок 9.10):

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400,height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 100, 10, 100, 50, fill='"', outline="black")
```

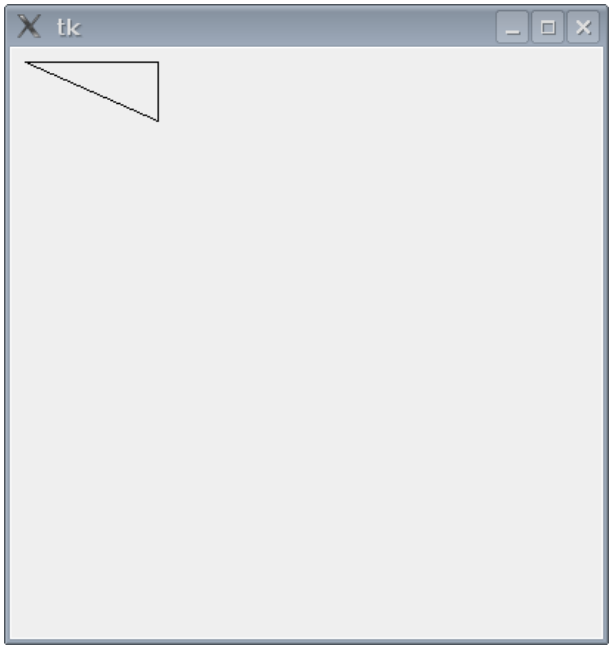


Рис. 9.10: Треуголник.

Чтобы нарисовать какой-то четырёхугольник, надо передать уже восемь координат (четыре пары):

```
>>> canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120, fill='', outline='black')
```

На рисунке 9.11 есть треугольник и ещё четырёхугольник (который выглядит как пара треугольников вместе).

9.7. Нарисуем картину

Питону можно сказать вывести на экран целую картинку сразу (например, фотографию). Для этого ему нужно сказать загрузить в память картинку и вывести её на холст при помощи функции `create_image`:

```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> canvas = Canvas(tk, width=400, height=400)
4. >>> canvas.pack()
5. >>> myimage = PhotoImage(file='test.gif')
6. >>> canvas.create_image(0, 0, image=myimage, anchor=NW)
```

В первых четырёх строчках мы, как всегда, создаём холст. В пятой строке загружаем картинку из файла «test.gif» в переменную myimage. В последней строке говорим нарисовать эту картинку в левом верхнем углу холста.

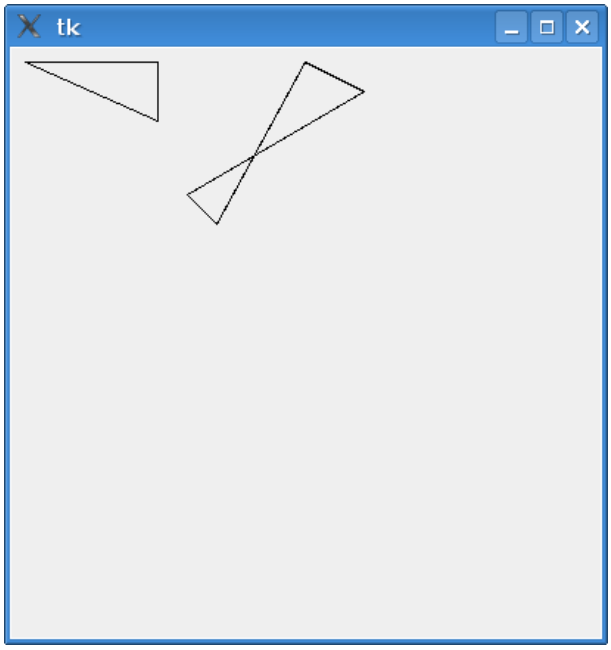


Рис. 9.11: Треугольник. И четырёхугольник.

Чтобы всё получилось, картинка должна лежать там, где Питон её ищет. Спросить у него, где он ищет, можно, вызвав функцию `getcwd()` из модуля `os`:

```
>>> import os
>>> print(os.getcwd())
```

Скорее всего, Питон напечатает тебе что-то вроде «/home/yourname», заменив yourname на имя пользователя. Например, если твоего пользователя зовут john, то на экране появится «/home/john».

Положи нужную тебе картинку в эту папку и запусти код из примера выше. Если всё сделать правильно, то получится-то что-нибудь наподобие рисунка 9.12.

Функция `PhotoImage` может загружать файлы форматов gif, ppm и pgm. Если тебе захочется загрузить другой файл — скажем, jpg или png, — то тебе пригодится модуль, который это умеет. Скажем, Python Imaging Library (PIL)[2] — этот модуль умеет загружать всевозможные картинки, а также обрезать их, менять цвета, поворачивать и всё в таком духе. Я не буду объяснять, как установить этот модуль, для этого бы потребовалось написать много того, что не относится к познаванию Питона.

[2]Его можно загрузить здесь: http://www.pythonware.com/products/pil/index.htm

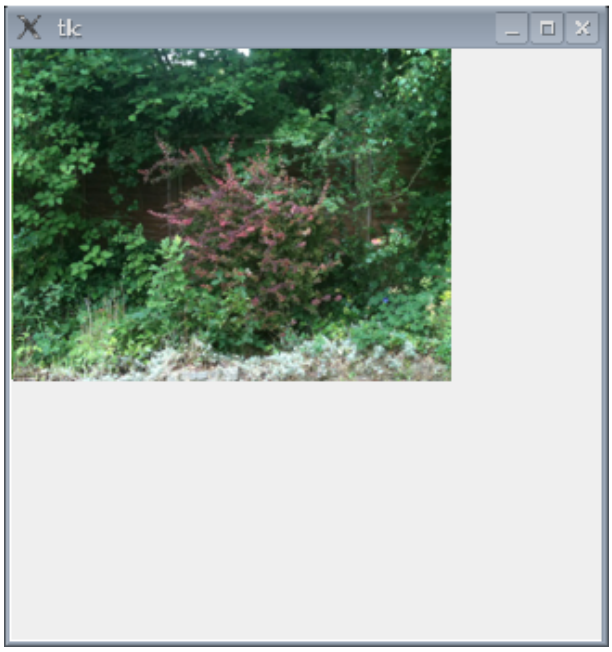


Рис. 9.12: Фотография.

9.8. Простая анимация

До сих пор мы рисовали неподвижные картинки. Этим возможности Питона не ограничиваются, Tk может нарисовать нам и движущиеся фигуры, хотя это не его сильная сторона. Например, мы вполне можем от него добиться

So far, we've seen how to do static drawing—that's pictures that don't move. What about animation? Animation is not necessarily Tk's strong suit, but you can do the basics. For example, we can create a filled triangle and then make it move across the screen using the following code:

```
1.  >>> import time
2.  >>> from tkinter import *
3.  >>> tk = Tk()
4.  >>> canvas = Canvas(tk, width=400, height=400)
5.  >>> canvas.pack()
6.  >>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
7.  1
8.  >>> for x in range(0, 60):
9.  ...     canvas.move(1, 5, 0)
10. ...     tk.update()
11. ...     time.sleep(0.05)
```

The moment you press the Enter key after typing the last line, the triangle will start moving across the screen (you can see it half-way across in figure 9.13).

How does it work?

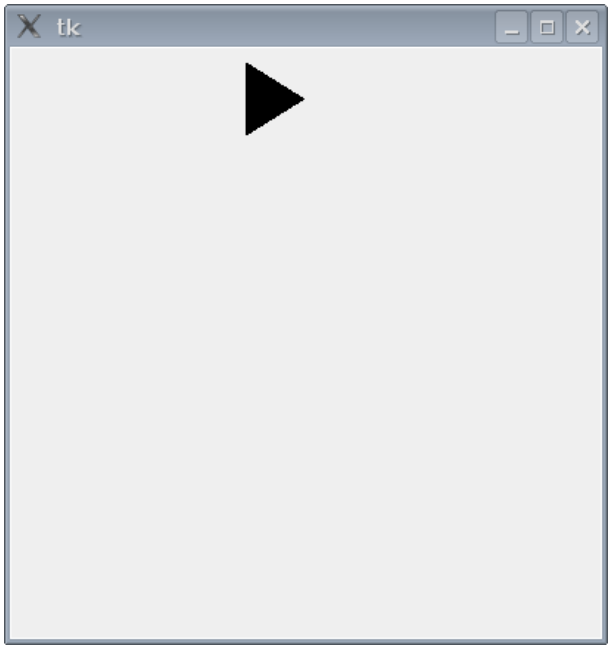


Рис. 9.13: The triangle moving across the screen.

Lines 1 to 5 we've seen before—it's just the basic setup to display a canvas—and in line 6, we create the triangle (using the `create_polygon` function), and in line 7 you can see the identifier (the number 1) that is returned by this function. In line 8, we setup a simple for-loop to count from 0 to 59.

The block of lines (9 to 11) is the code to move the triangle. The `move` function on the canvas object will move any drawn object by adding values to the object's x and y coordinates. For example, in line 9 we move the object with id 1 (the identifier for the triangle) 5 pixels across and 0 pixels down. If we wanted to move the back again we might use canvas.move(1, -5, 0).

The function `update` on the `tk` object forces it to update (if we didn't use `update`, tkinter would wait until the loop had finished before moving the triangle, which means you wouldn't see it move). Finally line 11 tells Python to sleep for 1/20th of a second (0.05), before continuing. We can change this code, so the triangle moves diagonally down the screen, by calling `move(1, 5, 5)`. First, close the canvas (by clicking on the X button on the window), then try this code:

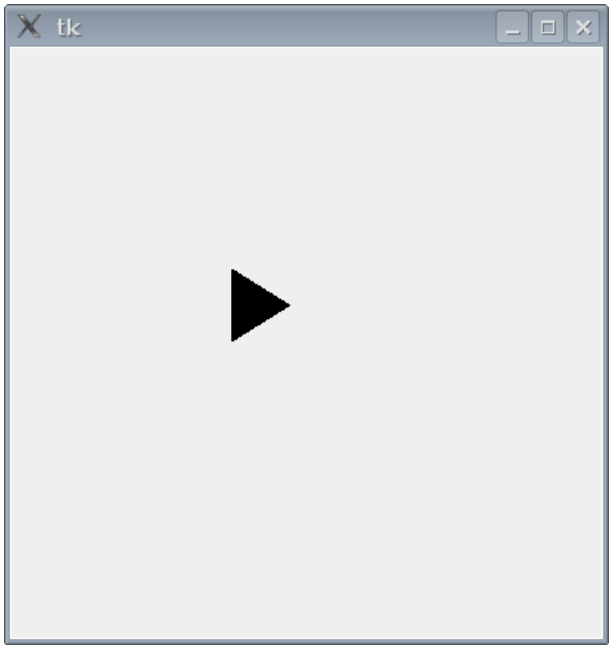


Рис. 9.14: The triangle moving down the screen.

```
>>> import time
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> for x in range(0, 60):
...     canvas.move(1, 5, 5)
...     tk.update()
...     time.sleep(0.05)
...
```

Figure 9.14 shows the triangle part way down the screen. Move the triangle diagonally back up the screen to its starting position, by using -5, -5:

```
>>> import time
>>> for x in range(0, 60):
...     canvas.move(1, -5, -5)
...     tk.update()
...     time.sleep(0.05)
```

9.9. Reacting to events...

We can also make the triangle react when someone hits a key, by using what are called *event bindings*. Events are things that occur while a program is running,

such as someone moving the mouse, hitting a key, or even closing a window. You can setup Tk to look out for these events, and then do something in response. To begin handling events we need to start by creating a function. Suppose we want the triangle to move when the enter key is pressed? We can define a function to move the triangle:

```
>>> def movetriangle(event):
...     canvas.move(1, 5, 0)
```

The function needs to have a single parameter (event), which is used by Tk to send information to the function about what has happened. We then tell Tk that this function should be used for a particular event, using the bind_all function on the canvas. The full code looks like this:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> def movetriangle(event):
...     canvas.move(1, 5, 0)
...
>>> canvas.bind_all('<KeyPress-Return>', movetriangle)
```

The first parameter in the bind_all function describes the event which we want Tk to look out for. In this case, it's the event <KeyPress-Return> (which is a press of the enter key). We tell Tk that the movetriangle function should be called when this key-press event occurs. If you run this code, click on the Tk canvas with your mouse, and then try hitting the Enter (or Return) key on your keyboard.

How about changing the direction of the triangle depending upon different key presses, such as the arrow keys? First of all we change the move triangle function to the following:

```
>>> def movetriangle(event):
...     if event.keysym == 'Up':
...         canvas.move(1, 0, -3)
...     elif event.keysym == 'Down':
...         canvas.move(1, 0, 3)
...     elif event.keysym == 'Left':
...         canvas.move(1, -3, 0)
...     else:
...         canvas.move(1, 3, 0)
```

The event object that is passed to movetriangle, contains a number of *properties*[3].

[3]Properties are named values, which describe something—for example, a property of the sky is that it's blue (sometimes), a property of a car is that it has wheels. In programming terms, a property has a name and a value.

One of these properties is `keysym`, which is a string holding the value of the actual key pressed. If `keysym` contains the string 'Up', we call `canvas.move` with the parameters (1, 0, -3); if it contains down we call with the parameters (1, 0, 3), and so on. Remember that the first parameter is the identifying number for the shape drawn on the canvas, the second parameter is the value to add to the x (horizontal) coordinate, and the last parameter is the value to add to the y (vertical) coordinate. We then tell Tk that the `movetriangle` function should be used to handle events from 4 different keys (up, down, left and right). So, the code now looks like this:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> def movetriangle(event):
...     if event.keysym == 'Up':
...         canvas.move(1, 0, -3)
...     elif event.keysym == 'Down':
...         canvas.move(1, 0, 3)
...     elif event.keysym == 'Left':
...         canvas.move(1, -3, 0)
...     else:
...         canvas.move(1, 3, 0)
...
>>> canvas.bind_all('<KeyPress-Up>', movetriangle)
>>> canvas.bind_all('<KeyPress-Down>', movetriangle)
>>> canvas.bind_all('<KeyPress-Left>', movetriangle)
>>> canvas.bind_all('<KeyPress-Right>', movetriangle)
```

With this example, the triangle now moves in the direction of the arrow key that you press.