

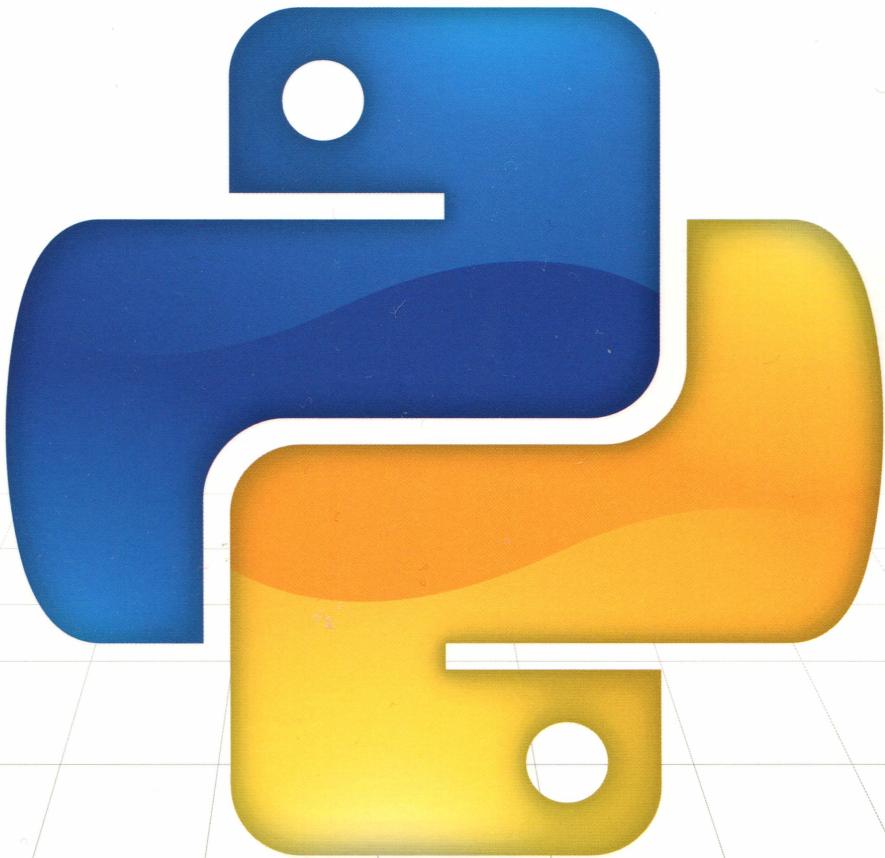
Майкл Доусон



Программируем на

python

Перевод третьего издания бестселлера



ПИТЕР®

COURSE TECHNOLOGY
CENGAGE Learning®

Научитесь программировать на Python играючи!

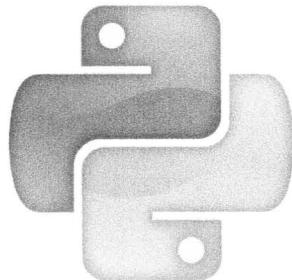
Для прочтения книги предварительных знаний не требуется

Michael Dawson

Python Programming

for the Absolute Beginner

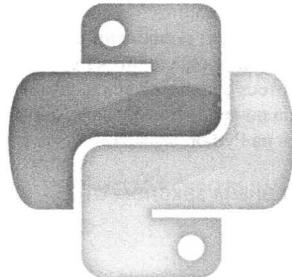
3rd Edition



COURSE TECHNOLOGY
CENGAGE Learning™

Майкл Доусон

Программируем на Python



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2014

Доусон М.

Программируем на Python. — СПб.: Питер, 2014. — 416 с.: ил.

Эта книга — идеальное пособие для начинающих изучать Python. Руководство, написанное опытным разработчиком и преподавателем, научит фундаментальным принципам программирования на примере создания простых игр. Вы приобретете необходимые навыки для разработки приложений на Python и узнаете, как их применять в реальной практике.

Для лучшего усвоения материала в книге приведено множество примеров программного кода. В конце каждой главы вы найдете проект полноценной игры, иллюстрирующий ключевые идеи изложенной темы, а также краткое резюме пройденного материала и задачи для самопроверки. Прочитав эту книгу, вы всесторонне ознакомитесь с языком Python, усвойте базовые принципы программирования и будете готовы перенести их на почву других языков, за изучение которых возьмется.

Научитесь программировать на Python играючи!

Права на издание получены по соглашению с Course Technology. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Краткое содержание

Благодарности	25
Об авторе	26
Введение	27
От издательства.....	29
Глава 1. Начало работы. Программа Game Over.....	30
Глава 2. Типы, переменные и основы ввода-вывода. Программа «Бесполезные факты»	43
Глава 3. Ветвление, циклы с оператором while и псевдокод. Игра «Отгадай число»	72
Глава 4. Циклы с оператором for, строки и кортежи. Игра «Анаграммы»	104
Глава 5. Списки и словари. Игра «Виселица»	134
Глава 6. Функции. Игра «Крестики-нолики»	165
Глава 7. Файлы и исключения. Игра «Викторина»	194
Глава 8. Программные объекты. Программа «Моя зверюшка».....	219
Глава 9. Объектно-ориентированное программирование. Игра «Блек-джек»	245

Глава 10. Разработка графических интерфейсов.	
Программа «Сумасшедший сказочник»	277
Глава 11. Графика. Игра «Паника в пиццерии»	308
Глава 12. Звук, анимация, разработка больших программ.	
Игра «Прерванный полет»	343
Приложение А. Сайт-помощник	387
Приложение В. Справка по пакету livewires	388
Алфавитный указатель	400

Оглавление

Благодарности	25
Об авторе	26
Введение	27
От издательства	29
Глава 1. Начало работы. Программа Game Over	30
Изучаем программу Game Over	30
Знакомство с Python.	31
Python прост в использовании.	31
Python — мощный язык.	32
Python — объектно-ориентированный язык	32
Python — «склеивающий» язык.	32
Python работает на всех платформах	33
Сообщество программистов на Python.	33
Python — бесплатная система с открытым исходным кодом	33
Установка Python в Windows	33
Установка Python в других операционных системах	34
Знакомство с IDLE	35
Программирование в интерактивном режиме	35
Программирование в сценарном режиме.	38
Вернемся к программе Game Over	40
Использование комментариев.	41
Пустые строки	41
Печать строки.	42
Ожидаем пользователя.	42
Резюме	42

Глава 2. Типы, переменные и основы ввода-вывода.

Программа «Бесполезные факты»	43
Знакомство с программой «Бесполезные факты»	43
Строки и кавычки	44
Знакомство с программой Game Over 2.0.	44
Кавычки внутри строк	45
Вывод на экран нескольких значений	46
Задание завершающей строки при выводе	47
Создание строк в тройных кавычках	47
Escape-последовательности в строках	48
Знакомство с программой «Воображаемые благодарности»	48
Вставка табуляционного отступа	49
Вывод обратного слеша	49
Вставка пустой строки	49
Вставка кавычек	50
Звук системного динамика	50
Сцепление и повторение строк	51
Знакомство с программой «Забавные строки»	51
Сцепление строк	52
Символ продолжения строки	52
Повторение строк	53
Работа с числами	53
Знакомство с программой «Текстовые задачи»	53
Числовые типы данных	55
Применение математических операторов	55
Переменные	56
Знакомство с программой «Привет»	56
Создание переменных	56
Использование переменных	57
Имена переменных	57
Получение пользовательского ввода	58
Знакомство с программой «Персональный привет»	59
Применение функции input()	59
Применение строковых методов	61
Знакомство с программой «Манипуляции с цитатой»	61
Создание новых строк с помощью строковых методов	62
Правильно выбранный тип	63

Знакомство с программой «Рантье» (версия с ошибкой)	64
Обнаружение и устранение логических ошибок.	65
Конвертация значений.	66
Знакомство с программой «Рантье» (версия без ошибки)	66
Преобразование строк в целые числа	67
Составные операторы присвоения	68
Вернемся к программе «Бесполезные факты»	68
Начальные комментарии.	69
Получение пользовательского ввода.	69
Вывод name на экран в нижнем и верхнем регистре	69
Пятикратный вывод имени	70
Подсчет количества секунд.	70
Вычисление значений moon_weight и sun_weight	70
Ожидание выхода.	70
Резюме	71

Глава 3. Ветвление, циклы с оператором while и псевдокод.

Игра «Отгадай число»	72
Генерирование случайных чисел	72
Знакомство с программой «Кости»	73
Импорт модуля random	74
Применение функции randint()	74
Применение функции randrange()	75
Условные конструкции с if	75
Знакомство с программой «Пароль»	75
Разбираемся, как работает конструкция if	77
Создание условий	77
Операторы сравнения	77
Создание блоков кода с помощью отступов	78
Создание собственных условных конструкций	79
Конструкции if с условием else	79
Знакомство с программой «Открыто/Закрыто»	79
Разбираемся в условиях else	80
Использование выражения elif	80
Знакомство с программой «Компьютерный датчик настроения»	80
Разбираемся в условиях elif	82

Создание циклов с использованием while	84
Знакомство с программой «Симулятор трехлетнего ребенка»	84
Разбираемся в работе цикла while	85
Инициализация управляющей переменной	85
Проверка значения управляющей переменной	86
Изменение значения управляющей переменной	86
Борьба с бесконечными циклами	86
Знакомство с программой «Проигранное сражение»	87
Трассировка программы	88
Условия, которые могут становиться ложными	88
Значения как условия	89
Знакомство с программой «Метрдотель»	90
Истинные и ложные значения	91
Намеренное создание бесконечных циклов	91
Знакомство с программой «Привередливая считалка»	92
Выход из цикла с помощью команды break	93
Команда continue и возврат к началу цикла	93
Как пользоваться командами break и continue	93
Составные условия	93
Знакомство с программой «Эксклюзивная сеть»	94
Логический оператор not	96
Логический оператор and	96
Логический оператор or	97
Планирование программ	98
Алгоритмы на псевдокоде	98
Пошаговая доработка алгоритма	99
Вернемся к игре «Отгадай число»	100
План программы	100
Начальный блок комментариев	101
Импорт модуля random	101
Объяснение правил	101
Установка начальных значений	101
Цикл отгадывания	102
Поздравления победителю	102
Ожидание выхода	102
Резюме	102

Глава 4. Циклы с оператором for, строки и кортежи.

Игра «Анаграммы»	104
Знакомство с игрой «Анаграммы»	104
Применение циклов for	105
Знакомство с программой «Слово по буквам»	105
Разбираемся в работе цикла for	106
Создание цикла for	107
Счет с помощью цикла for	107
Знакомство с программой «Считалка»	107
Счет по возрастанию	108
Счет по числам, кратным пяти	109
Счет по убыванию	109
Операторы и функции для работы с последовательностями:	
применение к строкам	109
Знакомство с программой «Анализатор текста»	110
Применение функции len()	110
Применение оператора in	111
Индексация строк	111
Знакомство с программой «Случайные буквы»	111
Позиции с положительными номерами	112
Позиции с отрицательными номерами	113
Случайный элемент строки	114
Что такое неизменяемость строк	115
Конструируем новую строку	116
Знакомство с программой «Только согласные»	116
Создание констант	117
Создание новых строк из существующих	118
Срезы строк	119
Знакомство с программой «Резчик пиццы»	119
Значение None	121
Разбираемся в срезах	121
Создание срезов	122
Сокращения в записи срезов	122
Создание кортежей	123
Знакомство с программой «Арсенал героя»	123
Создание пустого кортежа	124
Кортеж как условие	125

Создание непустого кортежа	125
Вывод элементов кортежа на экран	125
Перебор элементов кортежа	126
Использование кортежей	126
Знакомство с программой «Арсенал героя 2.0»	126
Настройка программы	127
Применение функции len() к кортежам	127
Применение оператора in к кортежам	128
Индексация кортежей	128
Срезы кортежей	128
Неизменяемость кортежей	129
Сцепление кортежей	129
Вернемся к игре «Анаграммы»	130
Настройка программы	130
Как сформировать анаграмму	131
Создание пустой строки для анаграммы	131
Настройка цикла	131
Выбор случайной позиции в слове	132
Новая версия jumble	132
Новая версия word	132
Программа приветствует игрока	132
Получение пользовательского ввода	132
Поздравление с правильно отгаданным словом	133
Конец игры	133
Резюме	133
Глава 5. Списки и словари. Игра «Виселица»	134
Знакомство с игрой «Виселица»	134
Использование списков	135
Знакомство с программой «Арсенал героя 3.0»	136
Создание списка	137
Применение функции len() к спискам	137
Применение оператора in к спискам	137
Индексация списков	138
Срезы списков	138
Сцепление списков	138
Изменяемость списков	138

Присвоение нового значения элементу, выбранному по индексу	139
Присвоение новых значений срезу списка	139
Удаление элемента списка	140
Удаление среза списка	140
Применение списочных методов	140
Знакомство с программой «Рекорды»	140
Настройка программы	141
Отображение меню	141
Выход из программы	142
Отображение списка рекордов	142
Добавление рекорда	142
Удаление рекорда	143
Сортировка списка рекордов	143
Обработка ошибочного выбора	144
Ожидаем пользователя	144
Когда использовать кортежи, а когда — списки	144
Вложенные последовательности	145
Знакомство с программой «Рекорды 2.0»	145
Создаем вложенные последовательности	146
Доступ к вложенным элементам	147
Распаковка последовательности	147
Настройка программы	148
Вывод результатов, содержащихся во вложенных кортежах	148
Добавление результата как вложенного кортежа	149
Обработка ошибочного выбора	149
Ожидаем пользователя	149
Распределенные ссылки	149
Использование словарей	152
Знакомство с программой «Переводчик с гикского на русский»	152
Создание словарей	152
Доступ к значениям в словаре	153
Настройка программы	155
Поиск значения	156
Добавление пары «ключ — значение»	156
Замена пары «ключ — значение»	157
Удаление пары «ключ — значение»	157

Обработка ошибочного выбора	158
Особенности словарей	158
Вернемся к игре «Виселица»	159
Настройка программы	159
Создание констант	159
Инициализация переменных	162
Создание основного цикла	162
Получение ответа игрока	163
Проверка наличия буквы в слове	163
Завершение игры	163
Резюме	164
Глава 6. Функции. Игра «Крестики-нолики»	165
Знакомство с игрой «Крестики-нолики»	165
Создание функций	165
Знакомство с программой «Инструкция»	167
Обявление функции	168
Документирование функции	168
Вызов нестандартной функции	169
Что такое абстракция	169
Параметры и возвращаемые значения	169
Знакомство с программой «Принимай — возвращай»	170
Передача данных с помощью параметров	170
Возврат значений функциями	171
Что такое инкапсуляция	172
Функции, которые и принимают и возвращают значения	172
Что такое повторное использование кода	173
Именованные аргументы и значения параметров по умолчанию	174
Знакомство с программой «День рождения»	174
Позиционные параметры и позиционные аргументы	175
Позиционные параметры и именованные аргументы	175
Значения параметров по умолчанию	176
Использование глобальных переменных и констант	177
Что такое области видимости	177
Знакомство с программой «Доступ отовсюду»	179
Чтение глобальной переменной внутри функции	180
Затенение глобальной переменной внутри функции	180

Изменение глобальной переменной внутри функции	181
Когда использовать глобальные переменные и константы	181
Вернемся к игре «Крестики-нолики»	182
План программы «Крестики-нолики»	182
Настройка программы	184
Функция <code>display_instruct()</code>	185
Функция <code>ask_yes_no()</code>	185
Функция <code>ask_number()</code>	185
Функция <code>pieces()</code>	186
Функция <code>new_board()</code>	186
Функция <code>display_board()</code>	186
Функция <code>legal_moves()</code>	186
Функция <code>winner()</code>	187
Функция <code>human_move()</code>	188
Функция <code>computer_move()</code>	188
Функция <code>next_turn()</code>	191
Функция <code>congrat_winner()</code>	191
Функция <code>main()</code>	192
Запуск программы	192
Резюме	192
Глава 7. Файлы и исключения. Игра «Викторина»	194
Знакомство с игрой «Викторина»	194
Чтение текстового файла	194
Знакомство с программой «Прочитаем»	195
Открытие и закрытие файла	197
Чтение текстового файла	198
Посимвольное чтение строки	199
Чтение всех строк файла в список	200
Перебор строк файла	200
Запись в текстовый файл	200
Знакомство с программой «Запишем»	201
Запись строк в файл	201
Запись списка строк в файл	202
Хранение структурированных данных в файлах	203
Знакомство с программой «Законсервируем»	203
Консервация данных и запись в файл	204

Чтение и расконсервация данных из файла	205
Полка для хранения консервированных данных	206
Извлечение консервированных данных через интерфейс полки.	207
Обработка исключений	208
Знакомство с программой «Обработаем»	208
Применение конструкций try/except	208
Типы исключений	209
Обработка нескольких типов исключений	211
Аргумент исключения	212
Добавление блока else	212
Вернемся к игре «Викторина»	212
Как организованы данные в текстовом файле	213
Функция open_file()	214
Функция next_line()	215
Функция next_block()	215
Функция welcome()	216
Настройка игры	216
Задание вопроса	216
Получение ответа	216
Проверка ответа	217
Переход к следующему вопросу	217
Завершение игры	217
Запуск функции main()	217
Резюме	217
Глава 8. Программные объекты. Программа «Моя зверюшка»	219
Знакомство с программой «Моя зверюшка»	219
Основы объектно-ориентированного подхода	221
Создание классов, методов и объектов.	221
Знакомство с программой «Просто зверюшка»	221
Обявление класса	222
Обявление метода	223
Создание объекта	223
Вызов метода	224
Применение конструкторов	224
Знакомство с программой «Зверюшка с конструктором»	224

Создание конструктора	225
Создание нескольких объектов	225
Применение атрибутов	226
Знакомство с программой «Зверюшка с атрибутом»	226
Инициализация атрибутов	227
Доступ к атрибутам	228
Вывод объекта на экран	228
Применение атрибутов класса и статических методов	229
Знакомство с программой «Классово верная зверюшка»	229
Создание атрибута класса	231
Доступ к атрибуту класса	231
Создание статического метода	232
Вызов статического метода	232
Что такое инкапсуляция объектов	232
Применение закрытых атрибутов и методов	233
Знакомство с программой «Закрытая зверюшка»	233
Создание закрытых атрибутов	234
Доступ к закрытым атрибутам	234
Создание закрытых методов	235
Доступ к закрытым методам	235
Соблюдаем приватность	236
В каких случаях нужны закрытые атрибуты и методы	237
Управление доступом к атрибутам	237
Знакомство с программой «Зверюшка со свойствами»	237
Создание свойств	238
Доступ к свойствам	239
Вернемся к программе «Моя зверюшка»	240
Класс Critter	241
Создание зверюшки	242
Создание меню	243
Запуск программы	243
Резюме	244
Глава 9. Объектно-ориентированное программирование.	
Игра «Блек-джек»	245
Знакомство с игрой «Блек-джек»	245
Отправка и прием сообщений	246

Знакомство с программой «Гибель пришельца»	247
Отправка сообщения	248
Прием сообщения	249
Сочетание объектов.	249
Знакомство с программой «Карты»	249
Создание класса Card	250
Создание класса Hand.	250
Применение объектов-карт	251
Сочетание объектов-карт в объекте Hand	251
Создание новых классов с помощью наследования.	252
Расширение класса через наследование	253
Знакомство с программой «Карты 2.0»	253
Создание базового класса	253
Наследование от базового класса	255
Расширение производного класса	255
Применение производного класса	256
Переопределение унаследованных методов	258
Знакомство с программой «Карты 3.0»	258
Создание базового класса	258
Переопределение методов базового класса	259
Вызов методов базового класса	260
Применение производного класса	261
Что такое полиморфизм.	262
Создание модулей	262
Знакомство с программой «Простая игра»	263
Пишем модуль	263
Импорт модулей	265
Применение импортированных функций и классов	265
Вернемся к игре «Блек-джек»	266
Модуль cards	266
Продумаем систему классов	267
Напишем псевдокод для основного цикла игры	268
Импорт модулей cards и games	269
Класс BJ_Card	269
Класс BJ_Deck	270
Класс BJ_Hand	270
Класс BJ_Player	272

Класс BJ_Dealer	272
Класс BJ_Game	273
Функция main()	275
Резюме	276

Глава 10. Разработка графических интерфейсов.

Программа «Сумасшедший сказочник»	277
Знакомство с программой «Сумасшедший сказочник»	277
GUI в подробностях	279
Что такое событийно-ориентированное программирование	280
Базовое окно	281
Знакомство с программой «Простейший GUI»	281
Импорт модуля tkinter	282
Создание базового окна	283
Изменение вида базового окна	283
Запуск событийного цикла базового окна	283
Применение меток	284
Знакомство с программой «Это я, метка»	284
Настройка программы	284
Создание рамки	284
Создание метки	285
Запуск событийного цикла базового окна	285
Применение кнопок	285
Знакомство с программой «Бесполезные кнопки»	286
Настройка программы	286
Создание кнопок	286
Запуск событийного цикла базового окна	287
Создание GUI с помощью класса	287
Знакомство с программой «Бесполезные кнопки — 2»	288
Импорт модуля tkinter	288
Объявление класса Application	288
Объявление метода-конструктора	288
Объявление метода, создающего элементы управления	289
Создание объекта класса Application	289
Связывание элементов управления с обработчиками событий	290
Знакомство с программой «Счетчик щелчков»	290
Настройка программы	290
Связывание обработчика с событием	291

Создание обработчика события	291
Обертка программы	291
Текстовые поля и области. Менеджер размещения Grid	291
Знакомство с программой «Долгожитель»	292
Настройка программы	293
Размещение элементов управления с помощью менеджера Grid	293
Создание текстового поля	294
Создание текстовой области	295
Текстовые элементы: извлечение и вставка данных	295
Обертка программы	296
Применение флажков	296
Знакомство с программой «Киноман»	297
Настройка программы	297
Ссылка только на родительский объект элемента управления	297
Создание флажков	298
Получение статуса флажка	299
Обертка программы	300
Применение переключателей	300
Знакомство с программой «Киноман-2»	300
Настройка программы	301
Создание переключателя	301
Доступ к значениям в переключателе	303
Обертка программы	303
Вернемся к программе «Сумасшедший сказочник»	303
Импорт модуля tkinter	303
Метод-конструктор класса Application	304
Метод create_widgets() класса Application	304
Метод tell_story() класса Application	305
Основная часть программы	306
Резюме	307
Глава 11. Графика. Игра «Паника в пиццерии»	308
Знакомство с игрой «Паника в пиццерии»	308
Знакомство с пакетами pygame и livewires	308
Создание графического окна	310
Знакомство с программой «Новое графическое окно»	310
Импорт модуля games	311

Инициализация графического экрана	312
Запуск основного цикла	312
Назначение фоновой картинки	313
Знакомство с программой «Фоновая картинка»	313
Загрузка изображения	313
Установка фона	315
Что такое система координат графики	315
Отображение спрайта	315
Знакомство с программой «Спрайт-пицца»	317
Загрузка изображения для спрайта	318
Создание спрайта	318
Добавление спрайта на экран	319
Отображение текста	320
Знакомство с программой «Ничего себе результат!»	320
Импорт модуля color	321
Создание объекта Text	322
Добавление объекта Text на экран	322
Вывод сообщения	323
Знакомство с программой «Победа»	323
Импорт модуля color	324
Создание объекта Message	324
Ширина и высота графического экрана	325
Добавление объекта Message на экран	325
Подвижные спрайты	325
Знакомство с программой «Летающая пицца»	326
Настройка скорости движения спрайта	327
Учет границ экрана	327
Программа «Скачущая пицца»	327
Настройка программы	328
Создание подкласса Sprite	328
Переопределение метода update()	329
Обертка программы	329
Обработка ввода с помощью мыши	330
Знакомство с программой «Подвижная сковорода»	330
Настройка программы	330
Чтение координат указателя	331
Настройка видимости указателя	332

Перенаправление ввода в графическое окно.	332
Обертка программы	332
Регистрация столкновений.	333
Знакомство с программой «Ускользающая пицца»	333
Настройка программы	333
Регистрация столкновений	335
Обработка столкновений	335
Обертка программы	335
Вернемся к игре «Паника в пиццерии».	336
Настройка программы	336
Класс Pan	336
Класс Pizza	338
Класс Chef	340
Функция main()	341
Резюме	342

Глава 12. Звук, анимация, разработка больших программ.

Игра «Прерванный полет»	343
Знакомство с игрой «Прерванный полет»	343
Чтение с клавиатуры	345
Знакомство с программой «Читаю с клавиатуры»	345
Настройка программы	346
Регистрация нажатий	346
Обертка программы	347
Вращение спрайта	347
Знакомство с программой «Крутящийся спрайт»	347
Применение свойства angle у спрайтов	349
Создание анимации	349
Знакомство с программой «Взрыв»	349
Посмотрим на картинки	350
Настройка программы	350
Создание списка изображений	351
Создание анимированного объекта	352
Работа со звуком и музыкой.	352
Знакомство с программой «Звук и музыка»	352
Работа со звуком	353
Работа с музыкой	355
Обертка программы	357

Разработка игры «Прерванный полет»	357
Функциональность игры	357
Классы игры	358
Медиаресурсы	358
Создание астероидов	358
Программа «Прерванный полет — 1»	359
Настройка программы	359
Класс Asteroid	360
Функция main()	361
Вращение корабля	361
Программа «Прерванный полет — 2»	361
Класс Ship	362
Инстанцирование класса Ship	362
Движение корабля	363
Программа «Прерванный полет — 3»	363
Импорт модуля math	364
Добавление переменной и константы в класс Ship	364
Изменение метода update() объекта Ship	364
Стрельба ракетами	365
Программа «Прерванный полет — 4»	366
Изменение метода update() объекта Ship	366
Класс Missile	367
Управление плотностью огня	369
Программа «Прерванный полет — 5»	369
Добавление константы в класс Ship	370
Создание метода-конструктора в классе Ship	370
Изменение метода update() объекта Ship	370
Обработка столкновений	371
Программа «Прерванный полет — 6»	371
Изменение метода update() объекта Missile	371
Добавление метода die() объекту Missile	371
Изменение метода update() объекта Ship	372
Добавление метода die() объекту Ship	373
Добавление константы в класс Asteroid	373
Добавление метода die() объекту Asteroid	373
Добавление взрывов	373
Программа «Прерванный полет — 7»	374
Класс Wrapper	374

Класс Collider	375
Изменение класса Asteroid	376
Изменение класса Ship	376
Изменение класса Missile.	377
Класс Explosion	377
Уровни, ведение счета, музыкальная тема	378
Программа «Прерванный полет — 8»	378
Импорт модуля color	378
Класс Game	379
Добавление переменной и константы в класс Asteroid.....	383
Изменение метода-конструктора в классе Asteroid.....	383
Изменение метода die() объекта Asteroid.....	383
Добавление константы в класс Ship	384
Изменение метода-конструктора в классе Ship	384
Изменение метода update() объекта Ship.....	384
Добавление метода die() объекту Ship.....	385
Функция main().	385
Резюме	385
Приложение А. Сайт-помощник	387
Приложение В. Справка по пакету livewires.....	388
Пакет livewires	388
Классы модуля games	388
Класс Screen.....	389
Класс Sprite	390
Класс Text.	391
Класс Message	392
Класс Animation	393
Класс Mouse	393
Класс Keyboard	394
Класс Music.	394
Функции модуля games	395
Константы модуля games	396
Константы модуля color	399
Алфавитный указатель.....	400

*Посвящается моим родителям,
прочитавшим все, что я когда-либо написал.*

Благодарности

Говорят, написать книгу — то же самое, что родить ребенка. Это сравнение кажется мне очень верным. Вот почему хочется поблагодарить всех, кто помог мне выпустить в свет мое милое детище.

Спасибо Дженнин Дэвидсон (Jenny Davidson) за двойную работу, которую она взяла на себя: корректуру и общую редакцию. Я признателен ей за критический подход и внимание к мелочам.

Спасибо Роберту Хоугу (Robert Hoag) за профессиональное выполнение технической редакции книги, а также за беседы нетехнического характера, да и вообще за приятную компанию.

Кроме того, спасибо Питу Шиннерсу (Pete Shinners) (автору модуля Pygame) и всем разработчикам LiveWires. Благодаря вам люди, которые только сегодня начинают программировать на Python, уже могут создавать настоящие мультимедийные программы, особенно игры.

Напоследок хотелось бы поблагодарить Мэтта за помощь с аудиофайлами, Криса — за музыкальный дар, а Дэйва — за согласие надеть поварской колпак.

Об авторе

Майкл Доусон (Michael Dawson) много лет посвятил разработке игр: на этом по-прище он работал и программистом, и дизайнером. Майк — выпускник университета Южной Калифорнии, бакалавр информатики. Сейчас он учит программированию игр на одном из факультетов Кинематографической школы в Лос-Анджелесе. Как преподаватель, Майк в свое время был участником Программы расширения UCLA и Академии цифровых СМИ в Стэнфорде. Кроме книги, которую вы держите в руках, он написал еще три учебника: «Программирование игр на C++ для начинающих» (*Beginning C++ through Game Programming*), «Руководство по программированию на Python» (*Guide to Programming with Python*) и «C++: Создание игр с текстовым интерфейсом» (*C++ Projects: Programming with Text-Based Games*). На странице Майкла Доусона в Сети (www.programgames.com) вы можете подробнее прочитать о нем самом и о его книгах.

Введение

Я без труда узнал картинку на экране: это было лицо — мое собственное лицо. От грубой прорисовки оно не переставало быть моим. С холодным любопытством профессионала я следил за тем, как на этом лице, словно разрываемом чем-то изнутри, постепенно пропадало человеческое выражение. Наконец, из моей головы на экране выбрался инопланетный пришелец, и голос из колонок спросил: «Хочешь увидеть это еще раз?»

Нет, то был не кошмарный сон, а моя работа. Я трудился в компании, которая разрабатывала и выпускала компьютерные игры, и вот в первом релизе одной нашей игры мне пришлось «торговать лицом». Смысл этой приключенческой забавы состоял в том, чтобы поймать указателем мыши картинку, хаотично движущуюся по экрану. Если игрок не укладывался в отведенное время... в общем, происходили уже известные вам события.

В другой период своей жизни я работал программистом крупного интернет-сервиса и время от времени переезжал из города в город. Казалось бы, две столь несхожие линии в карьере, но та база, которая позволила мне в обоих случаях добиться успеха, была заложена еще в детстве, когда я на своем домашнем компьютере начал писать простые игры.

Цель этой книги — научить вас языку программирования Python тем же самым путем, то есть через программирование несложных игр. Учиться, создавая свои собственные развлекательные программы, — одно удовольствие. Несмотря на развлекательный характер примеров, демонстрируется вполне серьезная техника программирования. В этой книге изложены все те основополагающие темы, которые содержатся в большинстве руководств для начинающих программистов, и освещается еще множество вопросов. В частности, я буду обращать ваше внимание на те идеи и методы, которые находят применение в типичных проектах.

Если в программировании вы новичок, то ваш выбор можно только одобрить. Python — лучший вариант для начинающих. У него ясный и простой синтаксис, который позволит вам очень скоро начать писать полезные программы. Кроме того, у Python имеется интерактивный режим с возможностью тестирования свежих идей буквально на лету.

Если у вас уже есть опыт программирования, вы тоже на верном пути. Python располагает всей той мощью и гибкостью, которую можно ожидать от современного объектно-ориентированного языка. Но при несомненной мощи Python, программы на нем пишутся удивительно быстро. В сущности, путь от идеи к компьютерной реализации сокращен настолько, что Python даже называют языком «программирования со скоростью мысли».

Как и любое хорошее руководство, эта книга начинается с самого начала. В первую очередь я намереваюсь рассмотреть установку интерпретатора Python под Windows. Вслед за тем я буду постепенно, шаг за шагом, вводить новые понятия и идеи. Каждый новый элемент демонстрируется в небольшой программе. К концу изучения этой книги вы освоите несколько тем, названия которых, возможно, пока звучат для вас необычно: структуры данных, работа с файлами, исключения, объектно-ориентированная разработка, программирование GUI и мультимедиа. Я надеюсь познакомить вас не только с программированием, но и с основами проектирования. Вы научитесь организовывать свою работу, разбивать задачу на удобные для реализации фрагменты, улучшать ранее написанный код. Время от времени вам может быть непросто, однако совсем уж невообразимых сложностей в этой книге нет. Полагаю, вы будете учиться с удовольствием и между делом напишете несколько простых, но симпатичных компьютерных игр.

Код всех программ, представленных в этой книге, а также необходимые вспомогательные файлы можно скачать с сайта-помощника www.courseptr.com/downloads. На сайте также имеются установочные файлы программного обеспечения (ПО), которое понадобится вам для запуска программ. Подробнее обо всем, что доступно на сайте, читайте в приложении А.

Важные моменты в книге отмечаются особым образом.

ПОДСКАЗКА

Опытные программисты любят делиться с новичками полезными знаниями, такими как эти.

ЛОВУШКА

Кое в чем начинающие часто ошибаются. Все подводные камни я буду помечать в такой врезке.

ХИТРОСТЬ

Под этой рубрикой будет рассказываться о методах и приемах, которые упрощают работу программиста.

НА САМОМ ДЕЛЕ

Образцы программ в этой книге — преимущественно игры. При их рассмотрении я буду показывать, какое применение те же самые идеи и понятия находят вне разработки игр.

ЗАДАЧИ

В конце каждой главы я буду предлагать вам написать несколько программ, в которых найдут применение навыки, только что освоенные вами.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitksi@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

1 Начало работы. Программа Game Over

Программировать — значит, по сути, заставлять компьютер что-то делать. Таково далекое от технических деталей, но очень точное определение. Изучая Python, вы научитесь писать программы: от простых игр и небольших утилит до бизнес-приложений с богатым графическим интерфейсом (GUI). Эти программы будут плодами лично вашего труда, и работать они будут именно так, как вы им предпишете.

В программировании есть доля науки, доля искусства и огромная примесь приключенческого романа. С этой главы начнутся ваши приключения в мире Python. Здесь вам предстоит узнать:

- что представляет собой Python и чем он интересен;
- как установить Python на ваш компьютер;
- как выводить текст на экран;
- что такое комментарии в коде и как ими пользоваться;
- как с помощью интегрированной среды разработки Python писать, редактировать, запускать и сохранять программы.

Изучаем программу Game Over

Программа Game Over, которой посвящена эта глава, выводит на экран два слова, печально известных всем любителям компьютерных игр: Game Over. На рис. 1.1 показано, как работает программа.

На рис. 1.1 показана так называемая консоль — окно, способное отображать только текст. Консольные приложения гораздо моне симпатичны, чем программы с графическим интерфейсом (GUI), но они проще в написании, что делает их удобными для начинающего программиста.

Программа Game Over исключительно проста; в сущности, это одна из самых простых программ, которые вообще можно написать на языке Python. Вот почему я и посвятил ей эту главу. В ходе разработки столь скромной программы вы без помех выполните всю подготовительную работу, которая нужна, чтобы начать программировать на Python. В частности, понадобится установить интерпретатор языка в вашей операционной системе. Вам также предстоит проработать весь цикл написания, сохранения и запуска программы. Окунувшись в первый раз в эту рутину, вы подготовитесь к созданию более крупных и полезных приложений.

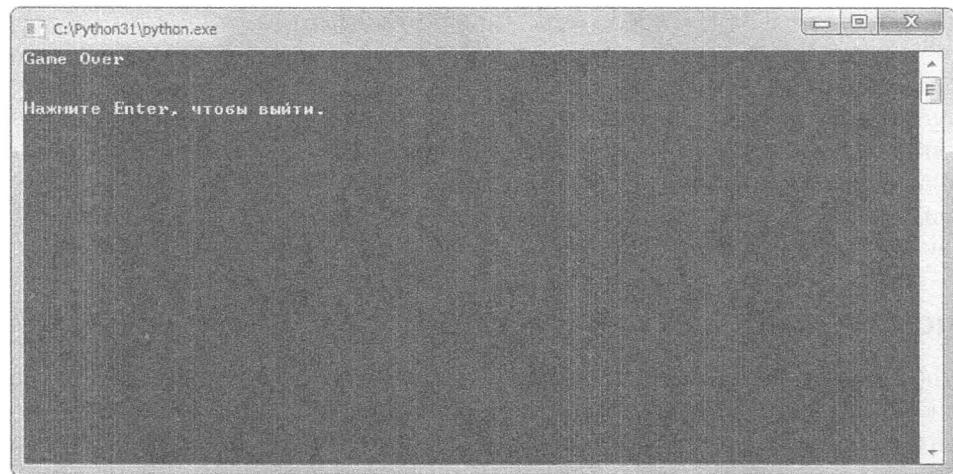


Рис. 1.1. Каждому, кто играл на компьютере, эти слова знакомы

НА САМОМ ДЕЛЕ

Программа Game Over — разновидность традиционной программы Hello World, выводящей на экран слова Hello World («Привет, мир»). Обычно Hello World — самая первая программа, которую пишет начинающий программист при знакомстве с новым для него языком. Эта практика столь широко распространена, что Hello World — термин, хорошо понятный каждому программисту.

Знакомство с Python

Python — мощный и простой в использовании язык программирования, разработанный Гвидо ван Россумом (Guido van Rossum). Первый релиз системы вышел в 1991 году. На Python можно быстро написать небольшой проект, а вообще он применим к проектам любого масштаба, в том числе коммерческим приложениям и программам, нацеленным на ответственные задачи. При знакомстве с документацией Python вас может удивить обилие упоминаний про число 42, спам и яйца. Все это осталось в память об английском коллективе комических актеров «Монти Пайтон» (Monty Python), в честь которого и был назван язык. Хотя Гвидо ван Россум произвел название Python от названия комик-группы, официальным символом языка стала змейка питон (других вариантов, пожалуй, и не оставалось, ведь значок на Рабочем столе слишком мал, чтобы уместить на нем лица шести английских юмористов).

Существует много других языков программирования. Чем же так привлекателен Python? Судите сами.

Python прост в использовании

Базовая цель любого языка программирования — построить «мостик» между мозгом программиста и компьютером. Те популярные языки, о которых вы, вероятно,

слышали, — Visual Basic, C#, Java и др. — принято называть *языками высокого уровня*. Этот термин предполагает, что они ближе к человеческому языку, чем к машинному. Так оно и есть. Но Python с его простыми и ясными правилами еще приближается к английскому языку. Создание программ на Python настолько незамысловатый процесс, что о нем иногда говорят как о «программировании со скоростью мысли». С точки зрения профессионального программиста, легкость Python — залог высокой производительности труда: программы на Python короткие и требуют меньше времени на разработку, чем программы на многих других популярных языках.

Python — мощный язык

Python обладает всеми возможностями, которых следует ожидать от современного языка программирования. Читая эту книгу, вы научитесь пользоваться разнообразными структурами данных и писать программы с GUI и файлами состояния.

Благодаря своей мощности Python привлекает разработчиков со всего мира. Им пользуются крупнейшие компании: Google, IBM, Industrial Light + Magic, Microsoft, NASA, Red Hat, Verizon, Xerox и Yahoo!. Профессиональные разработчики игр также применяют Python. Electronic Arts, 2K Games и Disney Interactive Media Group — все эти компании публикуют игры с кодом на Python.

Python — объектно-ориентированный язык

Объектно-ориентированное программирование (ООП) — современный подход к решению задач с помощью вычислительных машин. В рамках ООП собственная информация программы и команды, которые она передает компьютеру, записываются интуитивно понятным образом. Это, конечно, не единственный способ разработки программ, но в больших проектах, как правило, предпочтительный.

C#, Java и Python — объектно-ориентированные языки. Но у Python есть преимущество перед первыми двумя. В C# и Java ООП-подход проводится неукоснительно. Это делает короткие программы избыточно сложными; прежде чем начинающий программист сумеет сделать что-либо толковое, его нужно долго учить языку. В Python заложена другая модель: ООП-приемами пользоваться не обязательно. Вся их мощь по-прежнему находится в вашем распоряжении, но только вам дано решать, когда именно воспользоваться этой мощью. Ваша простенькая программа не требует ООП? Все в порядке. Ваш большой проект с целой командой программистов жизненно нуждается в ООП? Тоже не проблема. Python совмещает в себе функциональность и гибкость.

Python — «склеивающий» язык

Python легко интегрировать с другими языками, например C, C++ или Java. Таким образом, программист на Python может пользоваться разработками, уже имеющимися на других языках, и обращать в свою пользу сильные стороны этих языков (скажем, быстродействие C/C++), не поступаясь простотой разработки — отличительной чертой Python.

Python работает на всех платформах

Python-программу можно запустить на любой машине: от миниатюрного Palm до суперкомпьютера Cray. Если для суперкомпьютера вы недостаточно богаты, то пользоваться Python сможете на ПК с операционной системой Windows, Macintosh, Linux — и это лишь часть списка.

Программы на Python *независимы от платформы*, то есть неважно, какой операционной системой пользовался разработчик программы: код может быть исполнен на любом компьютере с установленным интерпретатором Python. Если, например, вы, работая на PC-совместимом компьютере, написали программу, то ничто не мешает вам переслать копии этой программы другу-линуксоиду и бабушке — обладательнице машины Macintosh. Если на компьютерах друга и бабушки установлен Python, программа будет функционировать.

Сообщество программистов на Python

Почти каждому языку программирования посвящена хотя бы новостная группа. Среди ресурсов о Python есть особая рассылка Python Tutor, в рамках которой новички могут в свободной форме задавать вопросы о языке. Адрес этой рассылки: <http://mail.python.org/mailman/listinfo/tutor>. Несмотря на то что в названии есть слово Tutor («наставник»), на вопросы может отвечать кто угодно: и эксперт, и начинающий.

Существуют и другие сообщества, посвященные разным вопросам использования Python. Их общая черта в том, что все они чрезвычайно открытые, дружелюбные по отношению к новым участникам. Неудивительно, раз уж язык так прост для освоения с нуля.

Python — бесплатная система с открытым исходным кодом

Интерпретатор Python бесплатен. Чтобы установить его и пользоваться им, не надо платить ни копейки. Лицензия, под которой доступен Python, разрешает и многие другие интересные вещи. Можно делать копии Python, в том числе модифицированные. Если угодно, можно даже продать Python (но не спешите бросать чтение — вы еще успеете получить свою прибыль!). Следование концепции открытого исходного кода — одна из многих особенностей, делающих Python столь популярным и успешным.

Установка Python в Windows

Прежде чем писать свою первую программу на Python, вы должны установить систему программирования на своем компьютере. Не беспокойтесь: вся процедура установки Python на компьютере с операционной системой Windows сейчас будет изложена.

Для установки интерпретатора Python в Windows сделайте следующее.

- Скачайте установочный файл Python Windows с сайта-помощника (www.courseptr.com/downloads). Файл находится в подкаталоге **Python** каталога **Software** и носит название **python-3.1.msi**.
- Запустите **python-3.1.msi** — установочный файл Python для Windows. Процесс установки показан на рис. 1.2.
- Подтвердите конфигурацию установки по умолчанию. Как только вы это сделаете, Python 3.1 появится на вашем компьютере.

ПОДСКАЗКА

www.courseptr.com/downloads — сайт-помощник, на котором размещены сопроводительные материалы к этой книге. Там, в частности, можно найти код всех законченных программ, о которых здесь пойдет речь, а также все необходимые установочные файлы и документацию. Более подробное описание см. в приложении А.

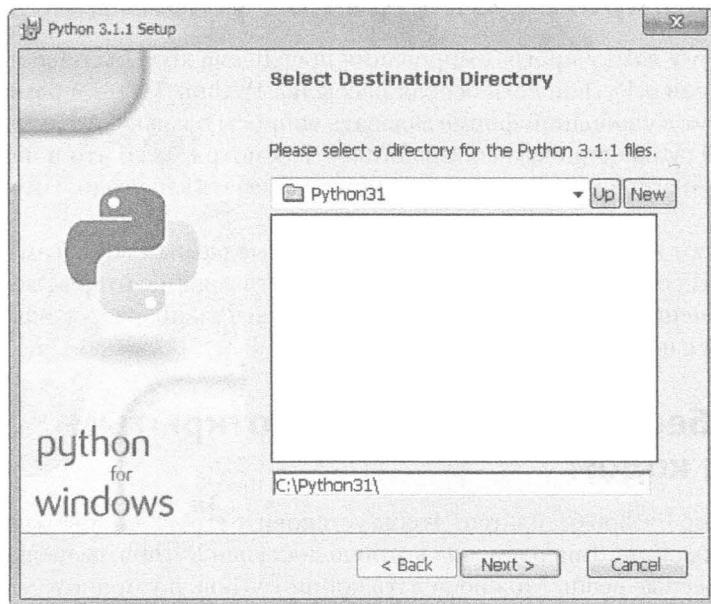


Рис. 1.2. Уже совсем скоро Python поселятся и на вашем компьютере

Установка Python в других операционных системах

Python реализован буквально для нескольких десятков операционных систем. Поэтому, если вы пользуетесь не Windows, вам достаточно будет посетить официальный сайт Python <http://www.python.org> и скачать оттуда последнюю версию интерпретатора для вашей платформы. Вид главной страницы сайта Python показан на рис. 1.3.

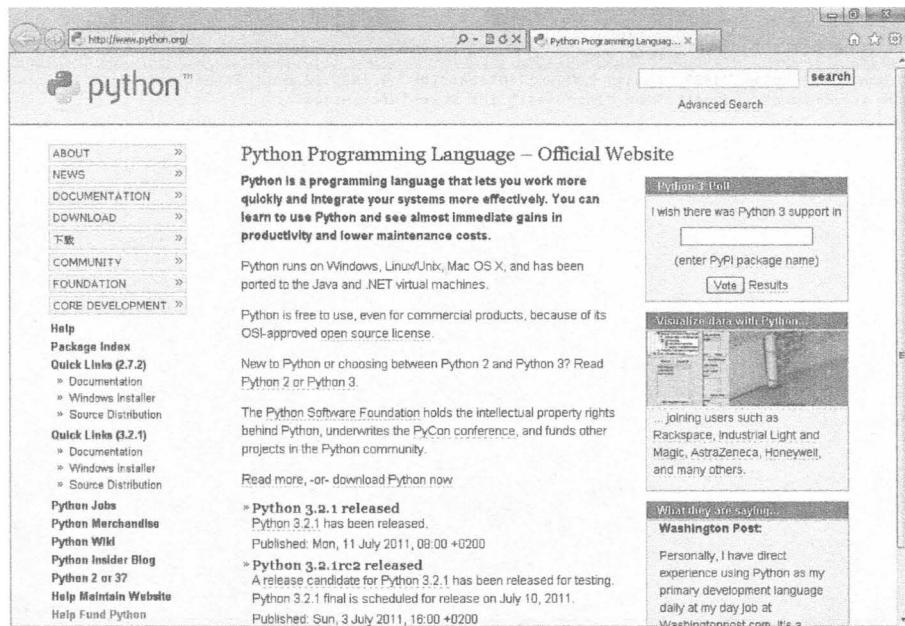


Рис. 1.3. Посетив официальный сайт Python, вы сможете скачать самую свежую версию системы и узнать много нового о языке

ЛОВУШКА

Может быть так, что на вашем компьютере уже установлен интерпретатор Python. Помните, что для корректной работы программ, представленных в этой книге, нужен Python версии 3.

Знакомство с IDLE

В комплект поставки Python входит интегрированная среда разработки под названием IDLE. *Средой разработки* называется совокупность инструментов, которые упрощают создание программ. Это в каком-то роде текстовый процессор для ваших программ. Но среда разработки позволяет не только писать, сохранять и редактировать программы. У IDLE есть два режима работы: интерактивный и сценарный.

Программирование в интерактивном режиме

Вот и пришло для вас время узнать, каково оно на вкус — настоящее программирование на Python. Для этого запустим Python в интерактивном режиме, в котором пользователь сообщает системе, что надо сделать, а та немедленно отвечает.

Ваша первая программа

Запустить интерактивную сессию можно так. В меню Пуск выберите Программы ▶ Python 3.1 ▶ IDLE (Python GUI). На экране появится окно, очень похожее на то, которое показано на рис. 1.4.

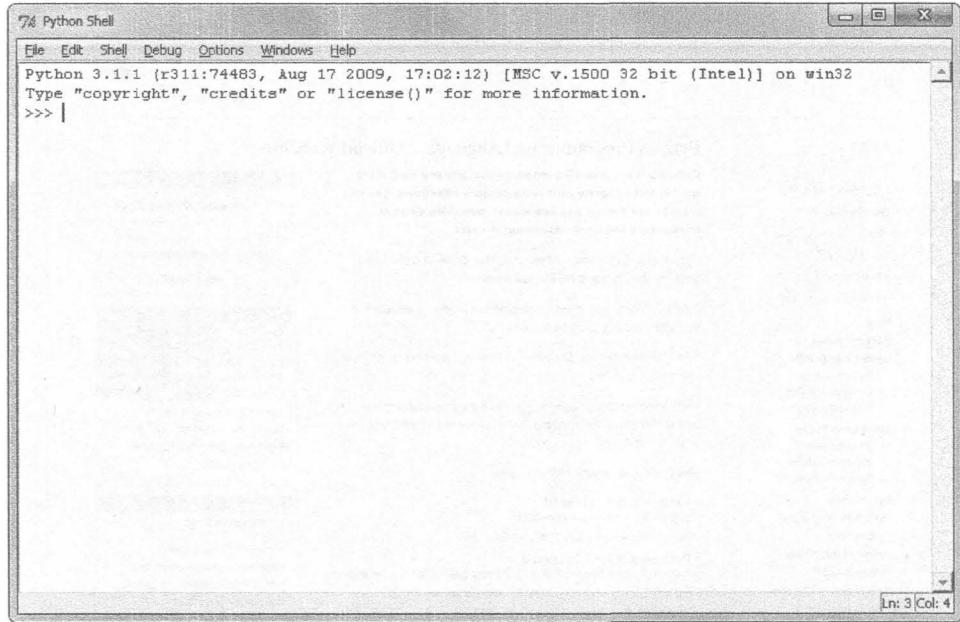


Рис. 1.4. Интерактивная сессия работы с Python. Система ожидает ваших команд

Это так называемый терминал Python (Python Shell). В командной строке (>>>) введите `print("Game Over")` и затем нажмите клавишу **Enter**. В ответ интерпретатор выведет на экран текст:

Game Over

Итак, вы только что написали свою первую программу на Python! Теперь можно считать вас настоящим программистом. Правда, надо еще кое-чему научиться, но это касается нас всех.

Использование функции `print`

Присмотритесь к строке, которую вы ввели: `print ("Game Over")`. Она очень незатейлива, как видите. Не зная ничего о программировании (и владея английским хотя бы на базовом уровне), можно догадаться, что этот код значит. Таков по своей сути весь Python. Его визитная карточка — краткость и ясность, которые вам еще предстоит оценить по достоинству в ходе изучения кое-чего посложнее.

Функция `print()` выводит на экран текст, который пользователь, заключив в кавычки, помещает внутрь скобок. Если ничем не заполнить скобки, будет напечатана пустая строка.

ЛОВУШКА

Python чувствителен к регистру. Существует договоренность, в силу которой названия функций пишутся строчными буквами. Поэтому запись `print(«Game Over»)` сработает, а `Print(«Game Over»)` или `PRINT(«Game Over»)` — нет.

УЧИМ ЖАРГОН

Теперь вы программист и, конечно же, должны понимать все эти забавные слова, известные только программистам. Итак, **функция** — это такая миниатюрная программа, которая выполняет определенную задачу, предписанную ей. Задача функции `print()` — выводить на экран значение или последовательность значений. Чтобы запустить, или же **вызвать**, функцию, надо написать ее имя и вслед за ним пару скобок. Это вы и сделали, введя в командную строку `print("Game Over")`. Некоторым функциям можно передавать значения, которые те будут обрабатывать. Такие значения, называемые **аргументами**, заключаются в скобки. В вашей первой программе функции `print()` был передан аргумент "Game Over". Приняв этот аргумент, функция отобразила на экране текст Game Over.

ПОДСКАЗКА

Функции в Python могут также возвращать данные в ту часть программы, которая эти функции вызывает. В таком случае говорят о возвращаемых значениях. Подробнее о возвращаемых значениях мы поговорим в главе 2.

В данном случае можно еще уточнить формулировку и сказать, что значение "Game Over", переданное функции `print()`, представляет собой *строку*. Это просто последовательность символов, таких как буквы на клавиатуре. Возможно, по сравнению с более понятными «текст» и «слово» название «строка» покажется не самым удачным, но оно исторически закрепилось по той причине, что всякий текст представляет собой строку (последовательность) символов. Говоря технически, "Game Over" — это строка: последовательность символов, образующих слова.

Код, введенный вами в окне интерпретатора, состоит из одного *выражения*. Если в естественных языках выражение — это законченная мысль, то в Python — законченная инструкция, которая велит системе что-либо сделать. Любая программа состоит из выражений.

Теперь, став программистом, вы можете похвастаться друзьям, что пишете код на Python. Слово «код», использованное здесь уже не в первый раз, означает всего лишь последовательность команд-выражений. Вместо глагола «программировать» можно говорить «кодировать». Вы можете сказать друзьям, например, так: «Вчера я весь вечер грыз чипсы и в бешеном темпе кодировал».

Создание ошибки

Компьютер все понимает буквально. Если в имени функции набрать неправильно хотя бы один символ, то компьютер даже не попытается предположить, что вы имели в виду. Так, например, если при работе в интерактивном режиме напечатать `print("Game Over")`, интерпретатор в ответ напишет так:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print("Game Over")
NameError: name 'print' is not defined
```

В вольном переводе на русский это значит: «Чего-чего?!» В сообщении об ошибке главный интерес представляет строка `NameError: name 'print' is not defined`.

(Ошибка в названии: имя 'primit' не определено). Интерпретатор тем самым признается, что не понимает команду `print`. Человек мог бы, не обратив внимания на опечатку, правильно понять смысл моих слов. Но компьютеры не такие. К счастью, ошибку программирования, подобную показанной выше, легко устранить: достаточно исправить опечатку.

Учимся понимать подсветку синтаксиса

Вы, вероятно, заметили, что слова на экране (на экране вашего компьютера, а не в иллюстрациях книги, конечно) отображаются разными цветами. Такое цветовое кодирование называется *подсветкой синтаксиса* и упрощает понимание того, что именно вы печатаете. Каждое слово окрашивается в один из цветов, и отнюдь не бессистемно.

Служебные слова, зарезервированные в языке Python, — такие как `print` — отображаются фиолетовым цветом. Строки, например, "Game Over" имеют зеленый цвет. Результат работы программы интерпретатор выводит на экран шрифтом голубого цвета. Когда вы приметесь за написание программ большего размера, эта цветовая схема станет вашим надежным помощником: она делает код понятным с первого взгляда и эффективно диагностирует ошибки.

Программирование в сценарном режиме

При работе в интерактивном режиме система отвечает вам мгновенно. С одной стороны, это очень хорошо: так можно сразу видеть результат. С другой стороны, для создания программ, которые бы сохранялись и потом повторно запускались, этот режим не подходит. К счастью, среда разработки IDLE имеет еще и так называемый сценарный режим. В нем можно писать, редактировать, загружать и сохранять программы. Это своего рода текстовый процессор для вашего кода, который позволяет выполнять и такие хорошо знакомые задачи, как поиск, замена, вырезка, вставка текста.

И снова ваша первая программа

В сценарный режим можно перейти из интерактивного окна, которым вы пользовались до сих пор. В меню `File` (Файл) выберите `New Window` (Новое окно). Появится новое окно, как показанное на рис. 1.5.

В этом новом (сценарном) окне напечатайте `print("Game Over")` и затем нажмите клавишу `Enter`. Ноль реакции! Это все потому, что вы сейчас находитесь в сценарном режиме и создаете список выражений, которые компьютер затем будет исполнять. Чтобы запустить полученную программу, надо прежде сохранить ее.

Сохранение и запуск программы

Для сохранения программы выберите пункт меню `File > Save As` (Файл > Сохранить как). Своей копии программы я присвоил имя `game_over.py` и, чтобы в дальнейшем было проще к ней обращаться, сохранил ее на Рабочем столе.

ПОДСКАЗКА

При сохранении обязательно присваивайте своим программам расширение `.py`. Это позволит многим программам, в том числе IDLE, распознавать данные файлы как программы на языке Python.

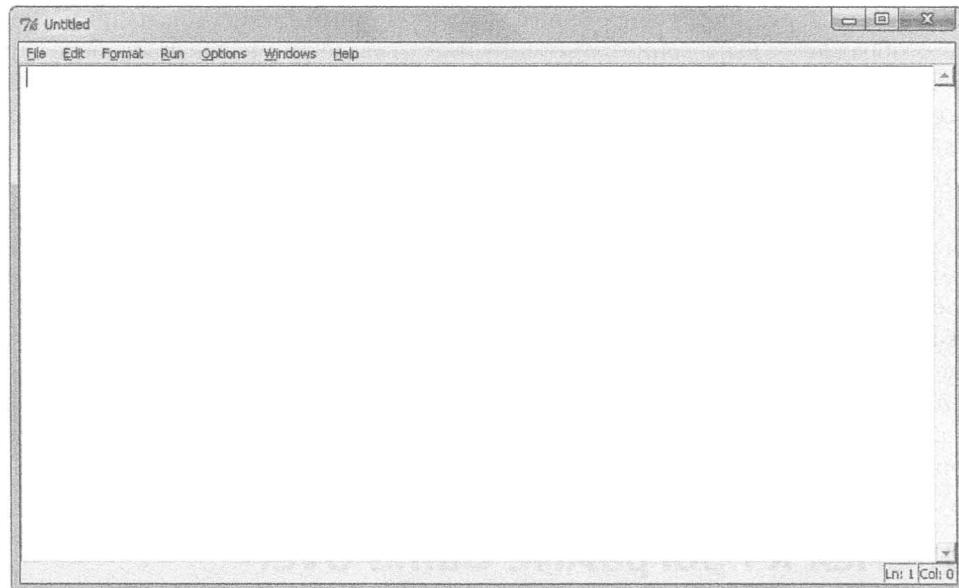


Рис. 1.5. Чистый лист ожидает ваших действий. Это и есть окно сценарного режима Python

Запустить программу Game Over можно, выбрав пункт меню Run ▶ Run Module (Выполнить ▶ Выполнить модуль). Результаты работы программы отобразятся в интерактивном окне. Как оно выглядело на экране моего компьютера, показывает рис. 1.6.

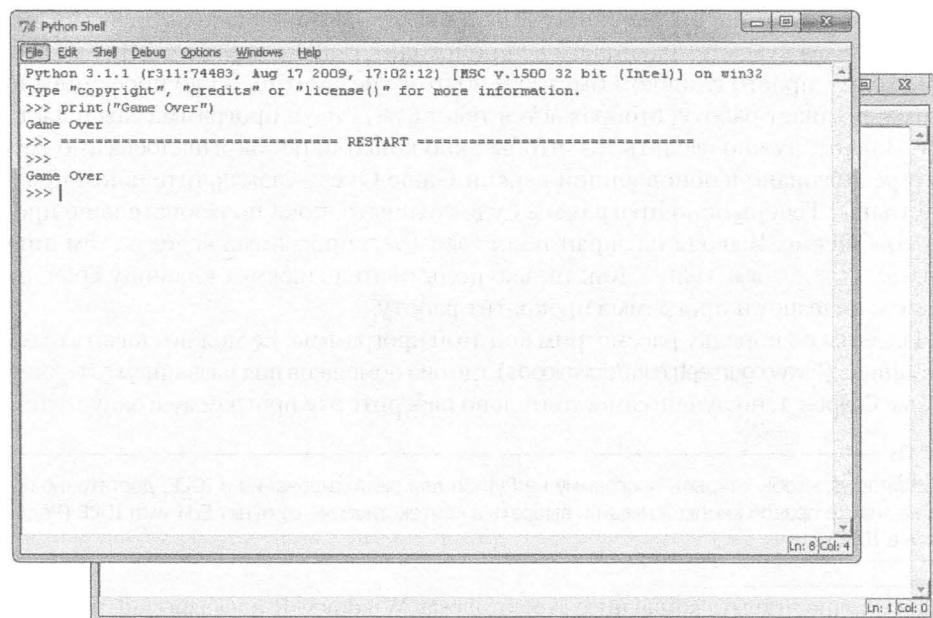


Рис. 1.6. Результат работы программы Game Over, запущенной с помощью IDLE

Заметьте, что интерактивное окно содержит «старый» текст. Так, например, все еще отображается выражение `print("Game Over")`, введенное ранее при работе в интерактивном режиме, и результат его работы — сообщение Game Over. Под этим текстом можно видеть сообщение о перезагрузке (RESTART), а еще ниже — результат запуска программы в сценарном режиме: Game Over. Повторю: чтобы запустить программу в IDLE, вы должны сначала ее сохранить.

Интерактивный режим подходит для быстрого тестирования отдельных идей, а сценарный — для написания программ, которые можно запускать позже. Оба режима в связке — комбинация, беспрогрышная для программиста. К примеру, мне, чтобы составить программу, нужен только сценарный режим. Но я все время держу открытым интерактивное окно, потому что при работе в сценарном режиме мне то и дело нужно проверять мелкие фрагменты кода на правильность использования функций. В сценарном окне текст программы обретает окончательную отделку, а интерактивное окно — такой черновик, где можно экспериментировать. Сочетание этого и другого позволяет мне программировать быстрее и качественнее.

Вернемся к программе Game Over

Итак, вы запустили одну из версий программы Game Over с помощью IDLE. Пока программа еще пишется, ее запуск через IDLE — удобный вариант. Но вы ведь наверняка хотите, чтобы готовый продукт работал точно так же, как и любая другая программа на вашем компьютере, то есть чтобы пользователю для запуска приложения достаточно было дважды щелкнуть на его значке.

Попробовав запустить таким образом программу Game Over в ее версии, которая разобрана выше, вы бы увидели, как на экране появляется и исчезает окно. Возможно, вы думаете, что ничего не происходит, однако все иначе. Что-то все же происходит, просто слишком быстро и поэтому незаметно. За доли секунды программа начинает работу, отображается текст Game Over и программа завершает работу. Значит, нужно сделать так, чтобы окно консоли не закрывалось само собой, что и реализовано в обновленной версии Game Over — заключительном проекте этой главы. Теперь окно программы будет открыто, пока пользователь не прочитает сообщение. Выводя на экран текст Game Over, программа вслед за тем пишет: Нажмите Enter, чтобы выйти.. Как только пользователь нажмет клавишу Enter, окно консоли исчезнет и программа прекратит работу.

Далее мы по порядку рассмотрим код этой программы. Ее можно скачать с сайта-помощника (www.courseptr.com/downloads), где она помещена под названием game_over.py в папке Chapter 1, но лучше самостоятельно наберите эту программу и запустите¹.

ХИТРОСТЬ

В ОС Windows, чтобы открыть программу на Python для редактирования в IDLE, достаточно щелкнуть на значке правой кнопкой мыши и выбрать в контекстном меню пункт Edit with IDLE (Редактировать в IDLE).

¹ Достаточно открыть командную строку, нажав Windows+R и набрав cmd, и запустить эту программу там. Работая с Python, очень полезно держать под рукой дежурную консоль. — Примеч. науч. ред.

Использование комментариев

Вот первые две строки программы:

```
# Game Over  
# Демонстрирует работу функции print
```

Эти строки — не выражения, которые компьютер должен исполнять. В сущности, компьютер их совершенно игнорирует. Такого рода пометки предназначены для людей и называются *комментариями*. Комментарии объясняют содержание кода на английском или любом другом пригодном для таких целей человеческом языке. С точки зрения других программистов, которым когда-либо придется разбираться в вашем коде, роль комментариев поистине бесцenna. Но велика их польза и лично для вас, потому что комментарий зачастую способен напомнить, как именно вам удалось решить неочевидную задачу.

Комментарий начинается с символа `#`. Все символы правее него (если только `#` не внутри строки) образуют комментарий. Интерпретатор нечувствителен к комментариям. Обратите внимание на то, что в IDLE комментарии для лучшей видимости подсвечиваются красным цветом. Считается хорошим тоном начинать программу с нескольких комментариев, как в моем примере. Полезно будет в числе этих комментариев указать название программы и ее назначение. Я не указал имя программиста и дату создания кода, но вообще-то это тоже принято комментировать.

Многие начинающие думают примерно так: «И зачем мне эти комментарии? Я написал программу, я знаю, как она работает, — чего же больше?» Можно услышать такое даже спустя месяц после разработки, но опытные программисты знают, что рано или поздно, обычно через несколько месяцев, автору приложения его собственный код перестает быть совершенно ясным. Впрочем, достаточно нескольких удачных комментариев, чтобы доработка старой программы стала гораздо проще.

НА САМОМ ДЕЛЕ

Особенно высоко будут ценить комментарии те ваши коллеги, которым поручат править ваш код. В мире профессионального программирования это типичная ситуация. Было подсчитано, что средний программист тратит большую часть своего рабочего времени на доводку уже существующих программ. Довольно часто разработчику поручают переделать код, написанный кем-то другим; далеко не всегда в таких условиях можно связаться с автором исходной программы, чтобы прояснить какие-либо вопросы. Так что хорошие комментарии оказываются критически важными.

Пустые строки

Строго говоря, следующая строка программы — пустая строка. Интерпретатор не обращает внимания на пустые строки, как и на комментарии: это нужно только людям, читающим код. Программу, которая удачно разделена пустыми строками, легче читать. Я обычно группирую близкие по содержанию строки кода и отделяю пустой строкой каждую такую группу от следующей. Например, в данной программе я отдал комментарии от вызова функции `print`.

Печать строки

Следующая команда вам уже знакома:

```
print("Game Over")
```

Это наш добрый друг — функция `print`. Как и при работе в интерактивном режиме, эта строка выводит на экран сообщение `Game Over`.

Ожидаем пользователя

Исполняя последнюю строку программы:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

интерпретатор выводит на экран слова `Нажмите Enter, чтобы выйти.` и ожидает, пока пользователь нажмет `Enter`. После нажатия `Enter` программа прекращает работу. Удобно таким способом удерживать окно консоли открытым до тех пор, пока пользователь не пожелает выйти из программы.

Надо сказать, уже пришло время разъяснить точнее и подробнее, что же происходит в этой строке. Но уж извините, до следующей главы подержу вас в напряжении, а уж там вы, несомненно, по достоинству оцените функцию `input()`.

Резюме

В этой главе вы постигли много нового и важного. Первым делом вы узнали, что такое Python и каковы его сильные стороны. Затем вы установили на свой компьютер интерпретатор языка и чуть-чуть попрактиковались в работе с ним. Вы научились выполнять отдельные команды в интерактивном режиме Python и увидели, как пользоваться сценарным режимом для написания, редактирования, сохранения и запуска программ большего объема. Вы узнали, как выводить текст на экран и ожидать, пока пользователь закроет окно консоли. Итак, начальные шаги в ваших странствиях по миру Python благополучно сделаны.

ЗАДАЧИ

- Научитесь вызывать ошибку: в интерактивном режиме введите название своего любимого сорта мороженого. Потом исправьте ошибку и создайте команду, которая будет правильным образом выводить на экран название этого сорта мороженого.
- Напишите и сохраните программу, которая будет выводить ваше имя и дожидаться, пока пользователь нажмет `Enter` для выхода. Запустите эту программу, дважды щелкнув на ее значке.
- Напишите программу, которая будет выводить на экран ваш любимый афоризм. Не забудьте о том, что автор афоризма должен быть упомянут на отдельной строке.

2 Типы, переменные и основы ввода-вывода. Программа «Бесполезные факты»¹

Теперь вы более или менее представляете себе, как сохранять и запускать программу. Настал час заглянуть поглубже и научиться чуть более сложным вещам. В этой главе я расскажу, какие есть способы категоризации и хранения данных в компьютере, а также (что, безусловно, важнее) как пользоваться данными в программах. Вы научитесь получать информацию от пользователя и таким образом создавать интерактивные приложения.

Итак, предстоит узнать:

- о применении строк в тройных кавычках и escape-последовательностей, которые дают больше функциональности в работе с текстом;
- о математических операциях в Python;
- о хранении данных в памяти компьютера;
- о переменных, с помощью которых можно получать доступ к данным и манипулировать ими;
- о пользовательском вводе и создании интерактивных программ.

Знакомство с программой «Бесполезные факты»

Освоив при изучении этой главы все нужные навыки, вы сумеете самостоятельно написать программу «Бесполезные факты», рабочее окно которой показано на рис. 2.1.

¹ В этой главе и далее мы переводим в образцах кода все английские высказывания (комментарии и текст, отображаемый на экране) на русский язык. В отличие от Python 2.x, Python 3.1 не требует в таких случаях включать русскую локаль, но в начале кода может понадобиться директива `# coding: cp1251` или `# coding: utf-8.` — Примеч. пер.

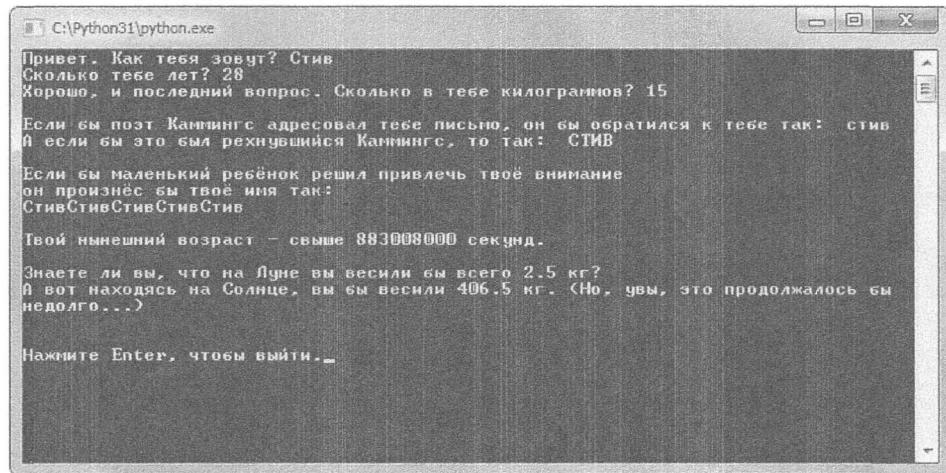


Рис. 2.1. Прежде чем посетить Солнце, 28-летнему Стиву не помешало бы сбросить лишний вес!

Из пользовательского ввода программа получает информацию об имени, возрасте пользователя и массе его тела. На основе этих нехитрых данных программа способна сгенерировать несколько забавных, но совершенно бесполезных фактов о пользователе: например, сколько тот будет весить на Луне.

На вид, как, впрочем, и внутренне, программа «Бесполезные факты» очень проста. Занимательность ей придает пользовательский ввод. Вы с нетерпением ждете момента, когда на экране появятся результаты, потому что эти результаты относятся лично к вам и ни к кому более. Большинство программ — от игр до бизнес-приложений — работают точно так же.

Строки и кавычки

В предыдущей главе приводился пример строки: "Game Over". Но строковые данные могут быть намного сложнее и объемнее. Вообразим, например, что надо довести до сведения пользователя инструкцию из нескольких абзацев или показать на экране особым образом отформатированный текст. Все эти задачи способны решать строки в кавычках.

Знакомство с программой Game Over 2.0

Game Over 2.0 — улучшенная версия программы-предшественницы Game Over: теперь уведомление о том, что игра подошла к концу, отображается на экране более впечатльно. Как работает эта программа, показано на рис. 2.2.

Эта программа показывает, что с использованием кавычек текст можно оформлять разными способами. Код программы доступен на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2; файл называется game_over2.py.

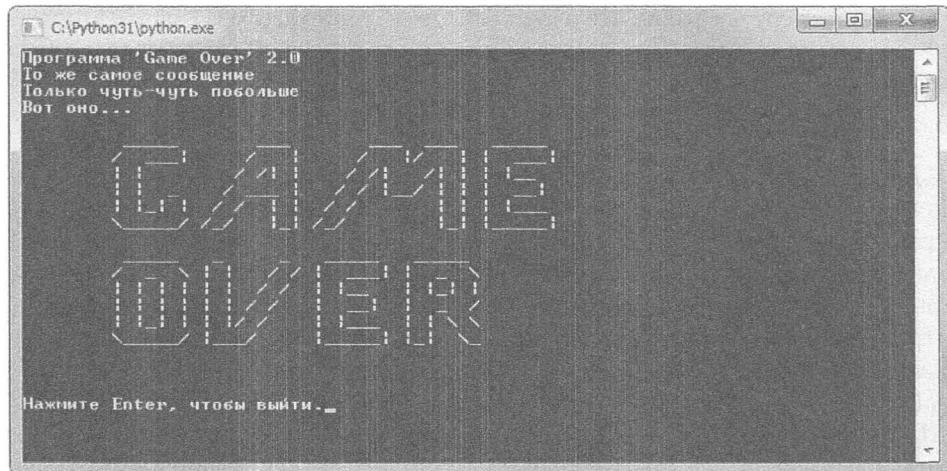


Рис. 2.2. Так бы сразу и сказали...

```
# Game Over - версия 2
# Показывает разные приемы работы со строками
print("Программа 'Game Over' 2.0")
print("То же", "самое", "сообщение")
print("Только",
      "чуть-чуть",
      "побольше")
print("Вот", end=" ")
print("оно...")
print(
      """
      / \   / \   / \ / \ / \
      |   |   |   |   |
      \ / / \ / / \ / / \
      |   |   |   |   |
      \ / / \ / / \ / / \
      |   |   |   |   |
      \ / / \ / / \ / / \
      """)
)
input("\n\nНажмите Enter, чтобы выйти.")
```

Кавычки внутри строк

Вы уже умеете создавать простейшие строки: для этого достаточно окружить текст парой кавычек, причем можно воспользоваться или апострофами (''), или кавычками

("") — компьютеру все равно. Поэтому 'Game Over' — та же самая строка, что и "Game Over". Но обратите внимание на код, в котором эта строка обнаруживается в нашей программе:

```
print("Программа 'Game Over' 2.0")
```

Здесь использованы кавычки обоих типов. А теперь вернитесь к рис. 2.2. На нем видны только апострофы, потому что они такая же часть строкового значения, как, например, буква G. Двойные кавычки не входят в строку. Они в каком-то роде аналог книжной обложки: компьютер по ним узнает, где строка начинается и где заканчивается. Итак, если в роли ограничителей у вас выступает пара кавычек, то внутри ограниченной ими строки будет использовано сколько угодно апострофов. И наоборот: если строка ограничена апострофами, то в ее состав может входить сколько угодно кавычек.

Применив кавычки одного типа в роли ограничителей, вы уже не сможете пользоваться ими внутри строки. Это целесообразно, ведь второе по порядку вхождение открывающей кавычки компьютер считает концом строки. Так, например, "Словами 'Хьюстон, у нас проблемы' Джим Ловелл вошел в историю американской астронавтики" — это корректная строка. Напротив, "Словами "Хьюстон, у нас проблемы" Джим Ловелл вошел в историю американской астронавтики" — некорректная строка. Как только компьютер видит вторую по счету кавычку, он считает, что строка закончилась, и распознает в качестве таковой лишь "Словами ". Вслед за тем упоминается Хьюстон — не служебное слово и не имя переменной. Компьютер даже не подозревает, что это за Хьюстон, и, конечно, выдаст ошибку.

Вывод на экран нескольких значений

Печатать на экране несколько значений можно одним вызовом функции print(). Для этого достаточно задать аргументы списком, перечислив их через запятую. Именно так сделано в коде нашей программы:

```
print("То же". "самое". "сообщение")
```

Здесь функции передаются три аргумента: "То же", "самое" и "сообщение". Этот код выводит на экран слова То же самое сообщение. Заметьте, что в роли разделителя между значениями на экране выступает пробел. Функция print() по умолчанию ведет себя именно так.

Аргументы, заданные списком, вы можете разбить на несколько строк кода так, чтобы эти строки (кроме последней) заканчивались разделителем-запятой. Следующий код образует единое выражение, которое выводит на экран одну текстовую строку Только чуть-чуть побольше. Я писал каждый следующий аргумент после запятой с новой строки:

```
print("Только",
      "чуть-чуть",
      "побольше")
```

Иногда полезно так делать, чтобы код было легче читать.

Задание завершающей строки при выводе

По умолчанию функция `print()` завершает выводимый на экран текст переходом на новую строку. Это значит, что вызов `print()` в следующий раз заставит дальнейший текст отобразиться на одну строку ниже. Этого, в общем-то, мы и хотим, но не помешала бы также возможность вручную назначать заключительный символ или символы. Можно было бы, например, вместо новой строки определить пробел как завершающую строку при вызове функции `print()`. Это означает, что при вызове следующего `print()` его строка напечатается сразу после пробела вместо перевода строки. Данная функциональность используется в коде нашей программы:

```
print("Вот", end=" ")
```

```
print("оно...")
```

Текст Вот оно... печатается, таким образом, в одну строку. Это потому, что в первой команде `print()` в качестве завершающей строки выбран пробел. Компьютер печатает на экране Вот (с пробелом после «т») и не переходит на новую строку. Следующая команда `print()` начинает выводить текст оно... сразу после пробела, следующего за «т». Легко понять, что такого эффекта я достиг, сделав пробел значением параметра `end` в функции `print()`: `end=" "`. Создавая собственные выражения с `print()`, вы можете тем же способом назначать любые строки в качестве заключительных печатаемых символов: запятую, имя параметра `end`, знак равенства, собственно строку. Теперь вы можете гораздо более гибко форматировать выводимый текст.

ПОДСКАЗКА

Если вам пока неясно, что такое параметр, не беспокойтесь. О параметрах и присвоении им значений вы сможете подробно прочесть в главе 6, в разделе «Параметры и возвращаемые значения».

Создание строк в тройных кавычках

Интереснее всего в новой программе, конечно, то, что она печатает Game Over огромными псевдографическими буквами. За это ответственна в коде строка:

“““



Вот что называется *строкой в тройных кавычках*. Это строка, ограниченная парой строенных кавычек. Как и ранее, неважно, пользуетесь вы кавычками или

апострофами; надо следить лишь за тем, чтобы открывающие и закрывающие кавычки были одинаковыми.

Как видите, строка в тройных кавычках может занимать в коде несколько строк и выводится на экране точно в таком же виде, как и вводится.

НА САМОМ ДЕЛЕ

Если вам по душе составленные из отдельных символов огромные буквы, как в Game Over 2.0, то вам понравится и вся так называемая ASCII-графика. Это, собственно говоря, картинки, составляемые из символов, которые доступны на клавиатуре. ASCII расшифровывается как American Standard Code for Information Interchange (Американский стандартный код для информационного обмена). В этом коде доступно представление 128 стандартных символов. Между прочим, графика на основе букв — не новое изобретение. Она появилась задолго до компьютера: первый доказанный факт выполнения графики на пишущей машинке датируется 1898 годом.

Escape-последовательности в строках

Escape-последовательности (или экранированные последовательности) позволяют вставлять в строки специальные символы. С помощью этого инструмента вы достигнете большей мощи и гибкости в отображении текста на экране. Типичную escape-последовательность образуют два символа: обратный слеш (\) и еще какой-либо символ. Все это пока лишь таинственные слова, но достаточно увидеть своими глазами несколько escape-последовательностей, и вы сразу поймете, как же просто ими пользоваться.

Знакомство с программой «Воображаемые благодарности»

Некоторые программы, известив пользователя о том, что игра окончена, выводят «благодарности» — список всех лиц, принимавших участие в работе над проектом. «Воображаемые благодарности» — это программа, в которой с помощью escape-последовательностей можно получить некоторые эффекты, недостижимые иным образом. Результат запуска программы показан на рис. 2.3.

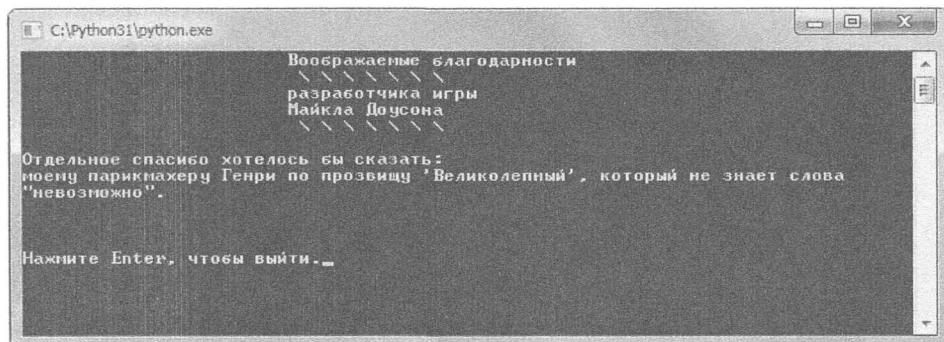


Рис. 2.3. Не надо оваций...

На первый взгляд код этой программы довольно загадочен, но вы его скоро поймете. Код можно найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 2; файл называется fancy credits.py.

Вставка табуляционного отступа

Иногда надо «отодвинуть» текст от левой кромки окна, по которой он обычно выравнивается. В текстовом редакторе можно воспользоваться для этого клавишей Tab, а в данных строкового типа применяется соответствующая escape-последовательность \t. Именно таково значение символов \t в строке кода:

```
print("\t\t\tВоображаемые благодарности")
```

Escape-последовательность табуляционного отступа встречается здесь три раза подряд, так что, выводя текст на экран, программа сначала делает три отступа, а затем печатает слова Воображаемые благодарности. Поэтому в консольном окне заголовок Воображаемые благодарности находится приблизительно по центру.

Табуляция удобна не только для создания отступов, как в этой программе, но и для оформления текста в виде нескольких столбцов.

Вывод обратного слеша

Если вы достаточно предусмотрительны, то, возможно, уже задумались о том, как же напечатать обратный слеш, если интерпретатор всегда распознает его как начальный символ escape-последовательности. Ответ прост: надо использовать два обратных слеша подряд. Каждая из следующих строк кода выводит на экран три табуляционных отступа и вслед за тем семь обратных слешей (столько же, сколько пар \\), разделенных пробелами:

```
print("\t\t\t\t \\ \\ \\ \\ \\ \\ \\ \\ \\ ")  
print("\t\t\t\t \\ \\ \\ \\ \\ \\ \\ \\ \\ ")
```

Вставка пустой строки

Едва ли не самая полезная из escape-последовательностей отвечает за переход на новую строку и выглядит как \n. Эта последовательность позволяет вставлять

в текст, где необходимо, символ пустой строки. Символ \n в начале строки может, например, выразительно отделить текст от напечатанного на экране выше. Этот эффект и достигнут в выражении:

```
print("\nОтдельное спасибо хотелось бы сказать:")
```

Компьютер видит последовательность \n, печатает пустую строку и лишь потом выводит текст Отдельное спасибо хотелось бы сказать:.

Вставка кавычек

Оказывается, вставить в строку цитату (даже с такими же ограничителями, как ограничители самой строки) очень легко. За вывод буквального апострофа отвечает escape-последовательность \' , а кавычка представляется в виде \" . Эти команды компьютер не может перепутать с концом строки. Вот почему не вызывает ошибки следующая строка кода, в которой использованы кавычки обоих типов:

```
print("моему парикмахеру Генри по прозвищу \'Великолепный\', который не знает слова \"невозможно\".")
```

В качестве ограничителей строки здесь выступают кавычки. Рассмотрим строку по частям, чтобы стало яснее.

1. \'Великолепный\' отображается на экране как 'Великолепный'.
2. Каждая из двух последовательностей \' отображается как апостроф.
3. \"невозможно\" на экране принимает вид "невозможно".
4. Обе последовательности \" печатаются как кавычки.

Звук системного динамика

При запуске этой программы вы без труда заметите ее отличительную черту: она звучит! Команда print("\a") заставляет звучать системный динамик вашего компьютера. Это непосредственно делает escape-последовательность \a — «символ системного динамика». Каждый раз при применении к нему функции print() динамик звучит. Можно «напечатать» только эту последовательность, как я и сделал, а можно поместить ее внутри строки. Ничто не мешает также инициировать звук системного динамика несколько раз — «напечатать» несколько символов \a.

Некоторые escape-последовательности работают предусмотренным образом лишь тогда, когда программа запускается из операционной системы, а не в IDLE. Таков, в частности, символ системного динамика. Пусть, например, я написал программу из одного выражения print("\a"). Если запустить ее в IDLE, на экране появится квадратик, а компьютер сохранит молчание. Но если запустить ту же самую программу из Windows, то есть дважды щелкнуть на значке файла, то системный динамик прозвучит, как и должен.

Итак, в действии escape-последовательности, кажется, не так уж и плохи: они обеспечивают большой прирост функциональности. В табл. 2.1 приведены некоторые полезные escape-последовательности.

Таблица 2.1. Escape-последовательности

Последовательность	Описание
\\\	Обратный слеш. Выводит: \
\'	Апостроф, или одиночная кавычка. Выводит: '
\"	Кавычка. Выводит: "
\a	Звук системного динамика. Заставляет звучать динамик компьютера
\n	Новая строка. Перемещает курсор в начало следующей строки
\t	Горизонтальный отступ — символ табуляции. Перемещает курсор вправо на один отступ

Сцепление и повторение строк

Итак, вы увидели, как можно вставлять в строки специальные символы. Но есть возможность манипулировать и целыми строками, например соединять две в одну или повторять какую-либо строку произвольное количество раз.

Знакомство с программой «Забавные строки»

Программа «Забавные строки» выводит на экран несколько строк. Результаты ее работы отражены на рис. 2.4. Хотя с печатью строк на экране вы уже знакомы, предлагаемый здесь способ вывода текста на экран будет для вас в новинку. Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `silly_strings.py`.

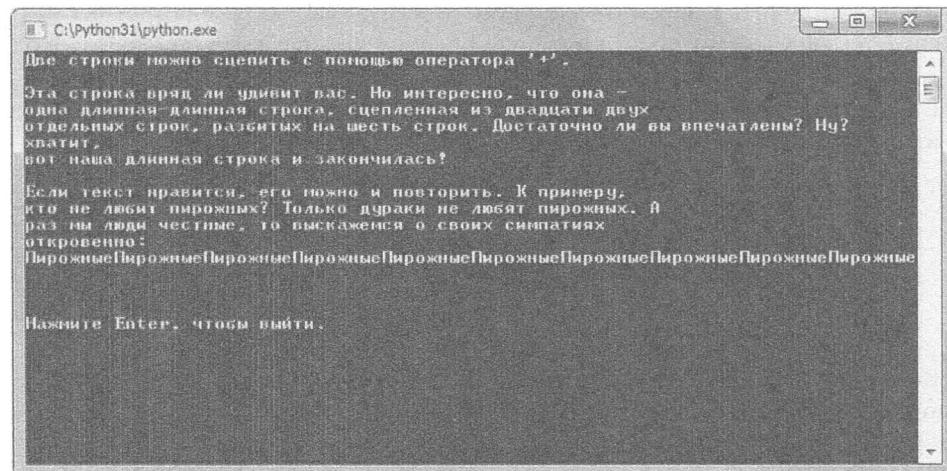


Рис. 2.4. Строки на экране выглядят иначе, чем в исходном коде программы

```
# Забавные строки
# Демонстрирует сцепление и повторение строк
print("Две строки можно " + "сцепить с помощью оператора ' +' .")
```

```

print("\nЭта строка " + "вряд ли " + "удив " + "ит вас. " \
+ "Но интересно. " + "что" + " она - \n" + "одна длинная-длин" \
+ "на " + "я" + " строка. сцепленная " \
+ "из " + "двадцати двух\n" + "отдельных строк. разбитых " \
+ "на шесть строк." + " Достаточно ли вы" + " впечатлены? " + "Ну хватит.\n" \
+ "вот " + "наша " + "длинная" + " строка и закончилась!")
print("\nЕсли текст нравится, его можно и повторить. К примеру.")
print("кто не любит пирожных? Только дураки не любят пирожных. А")
print("раз мы люди честные, то выскажемся о своих симпатиях")
print("откровенно:")
print("Пирожные" * 10)
input("\n\nНажмите Enter, чтобы выйти.")

```

Сцепление строк

Сцепление, или конкатенация, строк — это соединение их вместе так, что из двух или нескольких получается одна. Хороший пример — первая из показанных выше команд с print:

```
print("Две строки можно " + "сцепить с помощью оператора '+'.")
```

Оператор + соединяет строки "Две строки можно" и "сцепить с помощью оператора '+'." так, что получается единая новая строка. В этом нет ничего противоречащего интуиции: строки как бы складываются подобно числам, и за это отвечает тот же оператор, что и за сложение чисел. При конкатенации двух строк их содержимое «сплавляется» вместе без какого-либо символа-разделителя и без пробела. Поэтому, если вы, например, захотите сцепить строки "пирожные" и "чай", то получится "пирожныечай", а не "пирожные чай". В большинстве случаев между сцепляемыми строками нужен пробел; не забывайте его добавлять в одну из строк.

Следующая аналогичная команда показывает, что конкатенацию можно растягивать, насколько благорассудится:

```

print("\nЭта строка " + "вряд ли " + "удив " + "ит вас. " \
+ "Но интересно. " + "что" + " она - \n" + "одна длинная-длин" \
+ "на " + "я" + " строка. сцепленная " \
+ "из " + "двадцати двух\n" + "отдельных строк. разбитых " \
+ "на шесть строк." + " Достаточно ли вы" + " впечатлены? " + "Ну хватит.\n" \
+ "вот " + "наша " + "длинная" + " строка и закончилась!")

```

Компьютер печатает на экране одну длинную строку, и пользователь даже не подозревает, что она составлена из 22 отдельных строк.

Символ продолжения строки

Обычно одной строке кода соответствует одна команда. Но это не общеобязательное правило. Одно выражение можно растянуть на несколько строк кода. Для этого достаточно воспользоваться символом продолжения строки \ (обратным слешем), как показано в предшествующем коде. Поместите символ продолжения

строки там, где вы обычно используете в коде пробел (кроме тех пробелов, что внутри строки); разделенные этими символами строки интерпретатор будет рассматривать как одну длинную строку кода.

Компьютеру все равно, какой протяженности строки в коде, а вот людям — не все равно. Если вам кажется, что какую-либо строку кода для большего удобства чтения стоило бы разделить на несколько, не стесняйтесь делать это с помощью символа продолжения строки.

Повторение строк

Следующую важную идею, показанную в этой программе, иллюстрирует такой код:

```
print("Пирожные" * 10)
```

В результате работы этого кода создается и выводится на экран новая строка: "ПирожныеПирожныеПирожныеПирожныеПирожныеПирожныеПирожныеПирожныеПирожные" — десятикратное повторение строки "Пирожные".

Оператор повторения *****, как и оператор сцепления, легко понять интуитивно. Повторение строк — это ведь почти то же самое, что и умножение чисел; поэтому тот же самый оператор явился здесь закономерно. Чтобы повторить строку какое-либо количество раз, достаточно написать вслед за этой строкой оператор повторения ***** и искомое число.

Работа с числами

До сих пор вы пользовались строковыми значениями для представления текста. Но текст, очевидно, не единственный тип информации. Компьютеры хранят данные и в других формах. Одна из базовых и вместе с тем важнейших форм представления информации — числа. Почти в каждой программе выполняются операции над числами. Неважно, пишете вы трехмерную игру-шутер или пакет финансовых программ для частных клиентов, все равно ваш код должен каким-то образом представлять и обрабатывать числа. К счастью, в Python есть несколько типов числовых значений; этот набор удовлетворяет все нужды как разработчика игр, так и банковского программиста.

Знакомство с программой «Текстовые задачи»

Сюжетом для нашей следующей программы послужат кошмары из школьной алгебры — текстовые задачи. Да-да, те самые, в которых два поезда направляются навстречу друг другу из разных городов, грозя столкнуться посередине пути. Впрочем, не бойтесь. Уж вам-то не придется решать никаких текстовых задач, да и вообще вспоминать математику — за вас все сделает компьютер. Программа «Текстовые задачи» — это всего лишь занимательный (хочется надеяться) способ научить вас работе с числами. Внимание на рис. 2.5, на котором представлен пробный пуск.

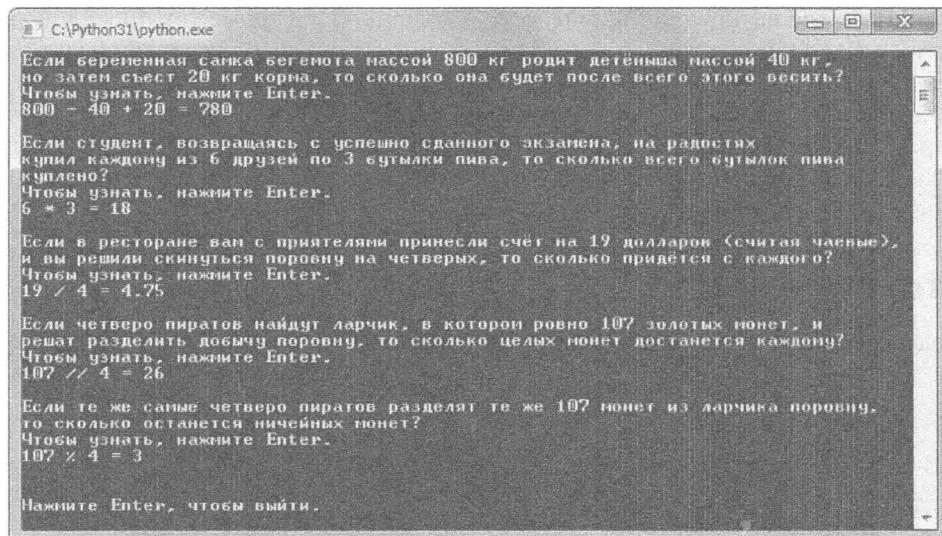


Рис. 2.5. Python позволяет не только складывать, вычитать, умножать и делить, но и следить за весом беременной самки бегемота

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `word_problems.py`.

```

# Текстовые задачи
# Демонстрирует математические операции
print("Если беременная самка бегемота массой 800 кг родит детеныша массой 40 кг,"
print("но затем съест 20 кг корма, то сколько она будет после всего этого весить?")
input("Чтобы узнать, нажмите Enter.")
print("800 - 40 + 20 =", 800 - 40 + 20)
print("\nЕсли студент, возвращаясь с успешно сданного экзамена, на радостях"
print("купил каждому из 6 друзей по 3 бутылки пива, то сколько всего бутылок пива
куплено?")
input("Чтобы узнать, нажмите Enter.")
print("6 * 3 =", 6 * 3)
print("\nЕсли в ресторане вам с приятелями принесли счет на 19 долларов (считая
чаевые).")
print("и вы решили скинуться поровну на четверых, то сколько придется с каждого?")
input("Чтобы узнать, нажмите Enter.")
print("19 / 4 =", 19 / 4)
print("\nЕсли четверо пиратов найдут ларчик, в котором ровно 107 золотых монет, и")
print("решат разделить добычу поровну, то сколько целых монет достанется каждому?")
input("Чтобы узнать, нажмите Enter.")
print("107 // 4 =", 107 // 4)
print("\nЕсли те же самые четверо пиратов разделят те же 107 монет из ларчика по-
ровну.")
print("то сколько останетсяничейных монет?")
input("Чтобы узнать, нажмите Enter.")
print("107 % 4 =", 107 % 4)
input("\n\nНажмите Enter, чтобы выйти.")

```

Числовые типы данных

Вполне очевидно, что в программе «Текстовые задачи» используются числа. Но далеко не столь очевидно, что в ней, собственно, два типа чисел. Python предоставляет в распоряжение программистов несколько числовых типов данных; те два, которыми мы пользуемся здесь, — наиболее типичные. Это *целые* (`int`) и *дробные* (`float`) числа. У целых чисел отсутствует дробная часть. Для примера, числа 1, 27, -100 и 0 — целые, а 2,376, -99,1 и 1,0 — это дробные числа.

Применение математических операторов

Математические операторы способны превратить ваш компьютер в дорогой многофункциональный калькулятор. Вся символика, которую я использовал выше, должна быть вам хорошо знакома. Например, код `800 - 40 + 20` вычитает 40 из 800 и затем добавляет 20, прежде чем вывести результат 780. Говоря технически, интерпретатор оценивает выражение `800 - 40 + 20` и находит, что оно равно 780. В данном контексте *выражение* — это всего лишь последовательность из значений, перемежающихся операторами; такую последовательность можно упростить, или «оценить». Код `6 * 3` умножает 6 на 3 и выводит результат 18; наконец, код `19 / 4` делит 19 на 4 и выводит десятично-дробный результат 4.75.

Да, все это математика, знакомая с младших классов. Но не так просто разгадать смысл следующего расчета: `107 // 4`. Применение `//` как математического оператора вам, вероятно, незнакомо. Здесь символы `//` (два правых слеша) выполняют деление нацело, то есть такое, при котором результат всегда является целым числом (дробная часть отбрасывается). В противовес ему существует обычное деление с оператором `/`, которое вы уже видели; в нем дробная часть результата учитывается. Результат операции `107 // 4` окажется равным 26.

Следующее выражение, `107 % 4`, тоже, возможно, заставит вас теряться в догадках. Здесь символ `%` означает *нахождение остатка по модулю* при делении нацело одного числа на другое. Выражение `107 % 4` оказывается равным 3.

В табл. 2.2 приводится сводка данных о некоторых математических операторах.

Таблица 2.2. Полезные математические операторы

Оператор	Описание	Пример	Результат
<code>+</code>	Сложение	<code>7 + 3</code>	10
<code>-</code>	Вычитание	<code>7 - 3</code>	4
<code>*</code>	Умножение	<code>7 * 3</code>	21
<code>/</code>	Деление (обычное)	<code>7 / 3</code>	2.3333333333333335
<code>//</code>	Деление (с остатком)	<code>7 // 3</code>	2
<code>%</code>	Остаток от деления	<code>7 % 31</code>	1

ПОДСКАЗКА

В табл. 2.2 обратите внимание на то, как работает обычное деление. Оно утверждает, что 7 разделить на 3 равно 2.3333333333333335. Это очень хорошее приближение, но не точное значение. Для большинства задач такие результаты подходят, но не забывайте об этой особенности, пользуясь десятично-дробными числами в своих программах. Поддержку точной десятичной арифметики для дробных чисел осуществляет модуль `decimal`. Подробнее о нем можно прочесть в документации Python.

Переменные

С помощью переменных можно хранить информацию и манипулировать ею. Это один из важнейших аспектов программирования. В Python переменные — способ организации данных и доступа к ним.

Знакомство с программой «Привет»

Обратите внимание на рис. 2.6, отражающий результаты работы программы «Привет».

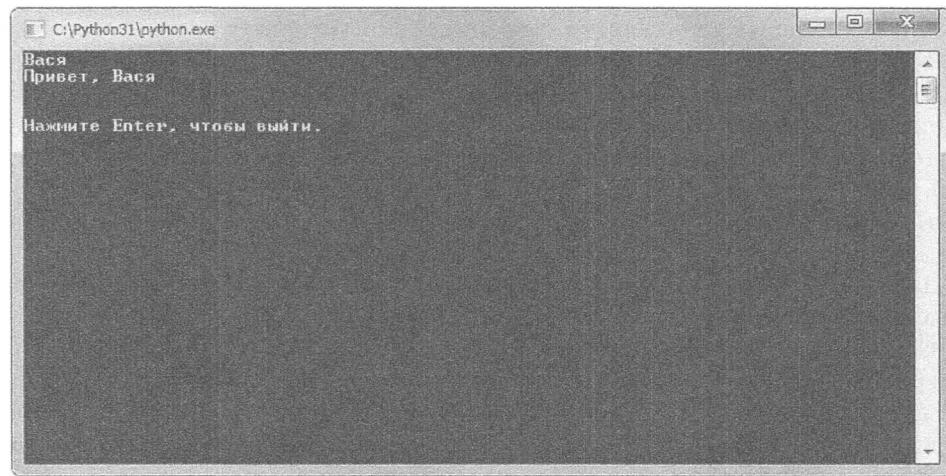


Рис. 2.6. Программа приветствует всех Василиев на свете

Судя по скриншоту на рис. 2.6, эта программа не может содержать ничего нового для вас. Но в ее коде использована такая фундаментальнейшая и новая вещь, как переменные. Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется greeter.py.

```
# Привет
# Демонстрирует использование переменных
name = "Вася"
print(name)
print("Привет.", name)
input("\n\nНажмите Enter, чтобы выйти.")
```

Создание переменных

Переменные позволяют хранить данные под «ярлыками»-именами и через них обращаться к этим данным. Вместо того чтобы давать точную ссылку на адрес, под которым в памяти компьютера хранятся какие-либо сведения, достаточно запросить об этих сведениях переменную. Это все равно что звонить другу по мобиль-

ному телефону. С точки зрения звонящего, совершенно неважно, где сейчас находится адресат звонка: несколько нажатий клавиш — и вот вы уже беседуете.

Впрочем, прежде, чем пользоваться переменной, ее надо создать, как в следующей строке кода:

```
name = "Вася"
```

Это так называемая *операция присвоения*. В ней создается переменная name, которой присваивается значение — ссылка на строку "Вася". Вообще принято говорить, что переменным присваиваются значения. Если переменная ранее не существовала, как name в данном примере, то сначала интерпретатор создаст ее, а потом присвоит ей значение¹.

ЛОВУШКА

Строго говоря, присваивающая конструкция сохраняет значение (которое спрашивается) где-либо в памяти, а переменной передает только ссылку на это значение, так что по-настоящему в переменных ничего не хранится. Ревнители языка Python говорят «переменная принимает значение», а не «переменной присваивается значение». Я буду пользоваться этими терминами как взаимозаменяемыми, судя по тому, где лучше выразиться одним словом, а где — другим.

О последствиях того, что переменные ссылаются на значения, а не хранят их, вы узнаете в разделе «Распределенные ссылки» главы 5.

Использование переменных

Созданная переменная обязательно ссылается на какое-либо значение. Удобство переменных в том, что их можно подставлять вместо соответствующих значений. Так, например, код:

```
print(name)
```

выводит на экран имя "Вася", как это сделала бы и команда print("Вася"). В свою очередь, код:

```
print("Привет.", name)
```

напечатает на экране "Привет.", затем пробел, затем имя "Вася". Опять же name взаимозаменяется с "Вася".

Имена переменных

Будучи единственным хозяином своей программы, вы вправе выбирать имена переменным. В программе-образце я решил назвать переменную name, но с тем же успехом можно было бы ее именовать person, guy или даже alpha7345690876: работа программы николько не изменилась бы. Для создания корректных имен переменных надо следовать всего нескольким правилам; о некорректном имени Python вам сообщит, выведя ошибку. Итак, важнейших правил два:

¹ Автор не совсем прав. Интерпретатор создает все нужные переменные при создании области видимости. Все переменные (слева от оператора присвоения) в локальном scope будут созданы сразу же, еще на этапе компиляции кода. — Примеч. науч. ред.

- имя переменной может состоять только из цифр, букв и знаков подчеркивания;
- имя переменной не может начинаться с цифры.

Вдобавок к этим двум абсолютным законам есть несколько негласных правил, которым следуют все опытные программисты. Вы тоже, приобретя некоторый опыт, почувствуете разницу между «просто корректными» и «хорошими» именами переменных (приведу один пример прямо сейчас: имя `alpha7345690876` — корректное, но очень, очень плохое). Перечислю основное, что надо запомнить.

- **Имя должно описывать суть.** Следует называть переменные так, чтобы другой программист, взглянув на ваш код, смог толком разобраться, что есть что. Поэтому, например, `score` лучше, чем `s`. (Иключение — кратковременно действующие переменные, которым программисты склонны присваивать короткие имена, например `x`. Но даже этот случай нельзя считать подлинным исключением, ведь, называя переменную `x`, программист дает понять, что она временная.)
- **Будьте последовательны.** Есть много разных позиций по вопросу о том, как лучше оформлять имена переменных, составленные из нескольких слов. Будет ли лучше, например, написать `high_score` или `highScore`? Я привык пользоваться подчеркиванием. Отнюдь не важно, какой системы вы придерживаетесь, лишь бы вы следовали все время только ей, хотя по стандарту оформления программ правильно использовать для переменных и функций именно подчеркивания.
- **Уважайте обычай языка.** Некоторые правила именования переменных — всего лишь дань традиции. Так, в большинстве языков программирования, в том числе и в Python, имя переменной принято начинать со строчной буквы. Другая традиция состоит в том, чтобы избегать подчеркиваний в качестве начальных символов в именах переменных (у имен с начальным подчеркиванием в Python особый смысл).
- **Следите за длиной.** Иногда эта рекомендация может вступать в противоречие с первой, требующей от имен переменных описательной силы. В самом деле, разве удобно пользоваться таким, например, именем: `personal_checking_account_balance`? Видимо, нет. Чрезсур длинные имена переменных проблематичны, в частности, тем, что загромождают код. А кроме того, чем длиннее имя, тем выше риск сделать в нем опечатку. Советую вам не создавать имена длиннее 15 символов.

ХИТРОСТЬ

Самодокументирующим называется код, из которого даже без комментариев легко понять, что происходит в программе. «Говорящие» имена переменных — немаловажный шаг на пути к созданию такого кода.

Получение пользовательского ввода

Оценив возможности программы «Привет», вы, вполне возможно, так и остались скептически настроены по отношению к переменным. Да, и без всяких переменных можно выполнить то же самое, что выполняет «Привет». Но в важнейших, фундаментальных вещах, таких как получение, сохранение и обработка значений из

пользовательского ввода, без переменных просто не обойтись. Рассмотрим следующую программу, которая умеет по-разному приветствовать всех пользователей.

Знакомство с программой «Персональный привет»

«Персональный привет» добавляет к функциональности «Привета» всего один, но ключевой элемент: пользовательский ввод. Вместо того чтобы работать с заранее определенными значениями, компьютер просит пользователя ввести имя и затем, приветствуя, обращается к человеку по имени. Работа программы отражена на рис. 2.7.

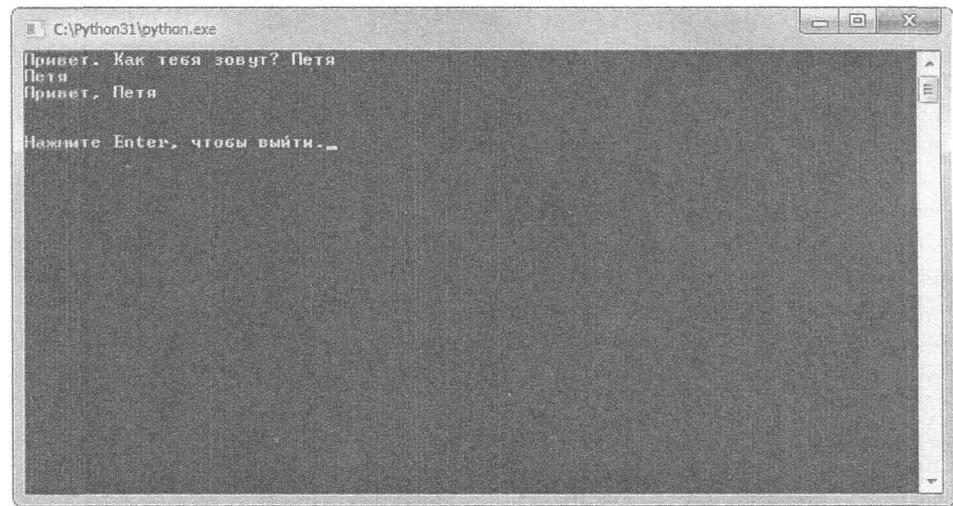


Рис. 2.7. Теперь переменной name присваивается строковое значение из пользовательского ввода. Это может быть, например, «Петя»

Не так уж и сложно принимать значения из пользовательского ввода. Почти не изменившийся код программы можно найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `personal_greeter.py`.

```
# Персональный привет
# Демонстрирует получение пользовательского ввода
name = input("Привет. Как тебя зовут? ")
print(name)
print("Привет, ", name)
input("\n\nНажмите Enter, чтобы выйти.")
```

Применение функции `input()`

Изменился только вид присваивающей конструкции:

```
name = input("Привет. Как тебя зовут? ")
```

Левая сторона этого выражения выглядит так же, как и в программе «Привет»: вновь созданной переменной name сейчас будет присвоено значение. Но на этот раз

правая сторона конструкции представляет собой вызов функции `input()`. Функция `input()` принимает текст, который пользователь вводит с клавиатуры. Ее строковый аргумент выводится на экран, прежде чем запросить пользователя о вводе. В данном случае я передал функции `input()` строку "Привет. Как тебя зовут? ". На рис. 2.7 можно видеть, что функция `input()` переадресовала мой вопрос пользователю. Она ожидает, пока пользователь что-нибудь введет. Сразу после нажатия клавиши `Enter` функция `input()` *возвратит* как строку в частности то, что ввел пользователь. Этакая строка, *возвращенная функцией*, и станет значением переменной `name`. Чтобы нагляднее представить себе то, что происходит, вообразите, что в присваивающей конструкции на месте `input()` стоит сразу та строка, которую ввел пользователь. Конечно, код программы мы уже не будем менять, но может быть полезно представлять, что на месте вызова функции на самом деле находится возвращаемое ею значение.

Если вам пока не до конца понятны все эти вызовы функций, аргументы и возвращаемые значения, прибегнем еще к одной аналогии: вызов функции `input()` сродни телефонному заказу пиццы. Сама функция `input()` — это как бы пиццерия. Позвонив в пиццерию, можно сделать заказ; вызвав функцию, можно заставить ее действовать. В телефонном разговоре с пиццерией заказчик сообщает какие-то подробности, например марку пиццы. А программист, когда вызывает функцию `input()`, передает ей аргумент — в нашем случае строку "Привет. Как тебя зовут? ". Когда вы повесите трубку, разносчик пиццы привезет вам вожделенную еду. А после того как вы вызовете функцию `input()`, она возвратит пользовательский ввод в виде строки.

Остальной код программы «Персональный привет» работает точно так же, как и в программе «Привет». Для компьютера совершенно неважно, каким образом переменная `name` приобрела свое нынешнее значение. Поэтому код:

```
print(name)
```

выводит на экран значение `name`, а код:

```
print("Привет.", name)
```

отображает слово "Привет.", пробел и затем значение `name`.

Теперь вы знаете достаточно, чтобы понять смысл той строки, которой мы уже по традиции заканчиваем консольные приложения. Цель этого кода — дождаться, пока пользователь нажмет клавишу `Enter`:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

Функция `input()` работает нужным нам образом. Поскольку в данном случае совершенно неважно, что пользователь введет (и введет ли вообще) перед нажатием `Enter`, значение, которое наша функция возвращает, никакой переменной не присваивается. Это необычно и как-то нелепо — получая значение, ничего с ним не делать, но такова моя воля. Если возвращаемое какой-либо функцией значение не сохранять в переменной, то система будет просто его игнорировать. Поэтому, как только пользователь нажмет `Enter`, вызов функции `input()` окончится, а вслед за ним остановится и программа. Консольное окно исчезнет.

Применение строковых методов

У Python богатый арсенал средств работы со строками. Одна из разновидностей этих средств — так называемые строковые методы. Они позволяют создавать новые строки на основе старых. Диапазон возможностей широк: от незатейливого приведения всех символов строки к верхнему регистру до хитроумных замен и подстановок.

Знакомство с программой «Манипуляции с цитатой»

По словам Марка Твена, «предсказывать вообще очень трудно, особенно будущее». Да, предвидеть грядущее нелегко, и именно поэтому в наши дни зачастую забавно звучат сделанные всего-то полвека назад «предсказания» судеб техники. Вот хороший пример: «Думаю, на мировом рынке можно будет продать штук пять компьютеров». Так заявил в 1943 году руководитель IBM Томас Уотсон (Thomas Watson). В моей программе «Манипуляции с цитатой» эти слова с помощью строковых методов несколько раз преподносятся по-разному. (Программу я сумел написать только потому, что в свое время успел купить компьютер № 3 из тех пяти. Шутка.) Пробный запуск показан на рис. 2.8.

The screenshot shows a Windows command-line interface window titled 'C:\Python31\python.exe'. The window displays the following text output:

```
Исходная цитата:  
Думаю, на мировом рынке можно будет продать штук пять компьютеров.  
А вот в верхнем регистре:  
ДУМАЮ, НА МИРОВОМ РЫНКЕ МОЖНО БУДЕТ ПРОДАТЬ ШТУК ПЯТЬ КОМПЬЮТЕРОВ.  
В нижнем регистре:  
думаю, на мировом рынке можно будет продать штук пять компьютеров.  
Как заголовок:  
Думаю. На Мировом Рынке Можно Будет Продать Штук Пять Компьютеров.  
С ма-а-аленькой заменой:  
Думаю, на мировом рынке можно будет продать несколько миллионов компьютеров.  
А вот опять исходная цитата:  
Думаю, на мировом рынке можно будет продать штук пять компьютеров.  
Нажмите Enter, чтобы выйти...
```

Рис. 2.8. Ошибочная догадка бизнесмена несколько раз выводится на экран, и каждый раз в новом виде благодаря строковым методам

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `quotation_manipulation.py`.

```
# Манипуляции с цитатой
# Демонстрирует строковые методы
# цитата из Томаса Уотсона, который в 1943 г. был директором IBM
quote = "Думаю, на мировом рынке можно будет продать штук пять компьютеров."
print("Исходная цитата:")
```

```

print(quote)
print("\nОна же в верхнем регистре:")
print(quote.upper())
print("\nВ нижнем регистре:")
print(quote.lower())
print("\nКак заголовок:")
print(quote.title())
print("\nС ма-а-аленькой заменой:")
print(quote.replace("штук пять", "несколько миллионов"))
print("\nА вот опять исходная цитата:")
print(quote)
input("\n\nНажмите Enter, чтобы выйти.")

```

Создание новых строк с помощью строковых методов

Здесь применяется новая для вас концепция, но код по-прежнему остался легко читаемым. Вот, например, строка:

```
print(quote.upper())
```

Глядя на нее, можно догадаться, что она выведет на экран такой вариант строки quote, который будет набран сплошь прописными буквами. Нужного эффекта добивается строковый метод `upper()`. *Строчный метод* – это своего рода «способность», которая есть у строки. К примеру, команда `quote` способна методом `upper()` создать новую строку с прежним текстом, но в верхнем регистре. Эту новую строку метод возвращает, и код, таким образом, оказывается равносителен следующему:

```
print("ДУМАЮ, НА МИРОВОМ РЫНКЕ МОЖНО БУДЕТ ПРОДАТЬ ШТУК ПЯТЬ КОМПЬЮТЕРОВ.")
```

Конечно, в действительности код выглядит иначе, но можно его представить себе таким, чтобы лучше понять работу метода.

Вы, вероятно, заметили, что вызов метода заканчивается скобками. Это должно было напомнить о функциях. В самом деле, методы схожи с функциями. Основное отличие в том, что стандартная функция, например такая, как `input()`, может быть вызвана независимо от чего, а строковый метод обязательно применяется к конкретной строке. Не было бы смысла написать, например, так:

```
print(upper())
```

Чтобы «вызвать», или «запустить», метод, надо к имени переменной, ссылающейся на строку, добавить точку, за ней вписать имя метода и, наконец, пару скобок. Эти скобки нужны не просто для красоты. Как и функциям, методам можно передавать аргументы. Правда, `upper()` не принимает ни одного аргумента, но далее мы увидим и принимающий аргументы строковый метод `replace()`.

Код:

```
print(quote.lower())
```

применяет метод `lower()` к строке `quote` и возвращает вариант этой строки, сплошь набранный строчными буквами. Новая строка печатается на экране.

Код:

```
print(quote.title())
```

выводит на экран такую версию quote, в которой, как принято в англоязычных заголовках, первые буквы всех слов прописные, а остальные — строчные. Именно таков формат строк, которые возвращает метод title().

Код:

```
print(quote.replace("штук пять", "несколько миллионов"))
```

выводит на экран новую строку, в которой по сравнению с quote все вхождения подстроки "штук пять" заменены строкой "несколько миллионов".

Метод replace() принимает по крайней мере два аргумента: «старый», заменяемый текст и «новый», приходящий ему на смену. Аргументы разделяются запятой. Можно также указать необязательный третий аргумент — целое число: это будет максимально возможное количество замен, которые методу разрешено совершить.

В заключение программа опять выводит на экран слова Томаса Уотсона. Это делает код:

```
print("\nA вот опять исходная цитата:")
print(quote)
```

На рис. 2.8 показано, что значение quote не изменилось. Запомните: строковые методы создают новые значения. Их работа никак не сказывается на исходной строке.

Строковые методы, уже знакомые вам, и еще несколько других кратко охарактеризованы в табл. 2.3.

Таблица 2.3. Полезные строковые методы

Метод	Описание
upper()	Возвращает строку, символы которой приведены к верхнему регистру
lower()	Возвращает строку, символы которой приведены к нижнему регистру
swapcase()	Возвращает новую строку, в которой регистр всех символов обращен: верхний становится нижним и наоборот
capitalize()	Возвращает новую строку, в которой первая буква прописная, а остальные — строчные
title()	Возвращает новую строку, в которой первая буква каждого слова прописная, а остальные — строчные
strip()	Возвращает строку, из которой убраны все интервалы (табуляция, пробелы, символы пустых строк) в начале и конце
replace(old, new [,max])	Возвращает строку, в которой вхождения строки old замещаются строкой new. Необязательный параметр max устанавливает наибольшее возможное количество замен

Правильно выбранный тип

До сих пор вам приходилось пользоваться тремя типами данных: строками, целыми числами и десятичными дробями. Важно не только знать эти типы по названиям, но и уметь работать с ними. Если вы не будете уметь этого, то рано или поздно столкнетесь с неожиданными результатами.

Знакомство с программой «Рантье» (версия с ошибкой)

Наша следующая программа рассчитана на тех беспечных прожигателей жизни, которым изрядный капиталец позволяет не заботиться о завтрашнем дне. Руководствуясь пользовательским вводом, программа вычисляет общую сумму ежемесячных издержек нью-йоркского рантье. Предполагается, что, узнав цену жизни не по средствам, обладатель капитала задумается о некотором упорядочении трат, чтобы ему не пришлось на старости лет искать работу. Впрочем, из заголовка вы могли понять, что программа не будет работать ожидаемым образом. Ее пробный запуск показан на рис. 2.9.

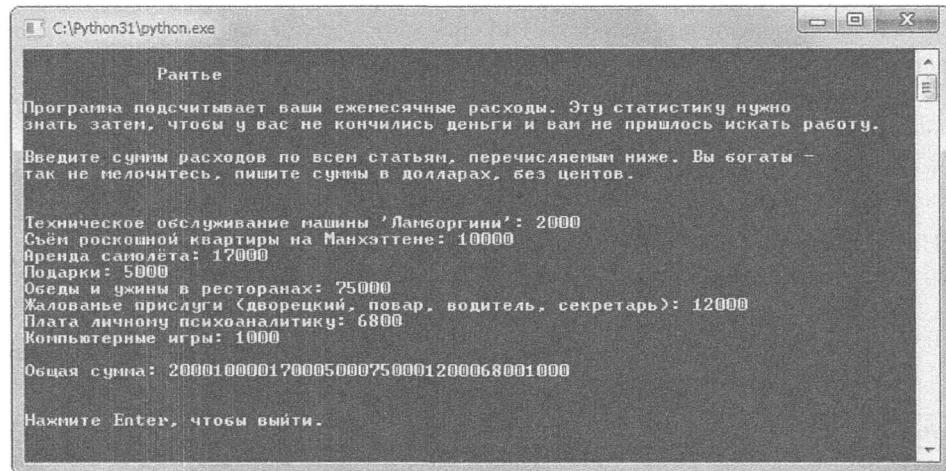


Рис. 2.9. Сумма ежемесячных трат должна быть велика, но не настолько же! В чем непорядок?

Очевидно, что в программе есть ошибка. Но ошибка не такая, от которой бы приложение вообще не запускалось. Если программа благополучно работает, но выдает неожиданные результаты, значит, она содержит логическую ошибку. Пользуясь тем, что вам уже известно, вы могли бы выяснить, взглянув на код программы, что за неполадка вкралась в ее работу. Код можно найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `trust_fund_bad.py`.

```
# Рантье (версия с ошибкой)
# Демонстрирует логическую ошибку
print(
"""


```

Рантье

Программа подсчитывает ваши ежемесечные расходы. Эту статистику нужно знать, чтобы у вас не закончились деньги и вам не пришлось искать работу.
Ведите суммы расходов по всем статьям, перечисляемым ниже. Вы богаты - так не мелочитесь, пишите суммы в долларах, без центов.

```
"""


```

```
)
car = input("Техническое обслуживание машины 'Ламборгини': ")
rent = input("Съем роскошной квартиры на Манхэттене: ")
jet = input("Аренда самолета: ")
gifts = input("Подарки: ")
food = input("Обеды и ужины в ресторанах: ")
staff = input("Жалованье прислуги (дворецкий, повар, водитель, секретарь): ")
guru = input("Плата личному психоаналитику: ")
games = input("Компьютерные игры: ")
total = car + rent + jet + gifts + food + staff + guru + games
print("\nОбщая сумма:", total)
input("\n\nНажмите Enter, чтобы выйти.")
```

Если вы пока не заметили проблему, то ничего страшного. Дам еще одну подсказку: присмотритесь хорошенько к результату на рис. 2.9. Вот длинное-длинное число, которое программа выводит в качестве суммы расходов. А вот отдельные суммы, которые по порядку вводил пользователь. Ничего не подмечаете? Ладно, поехали дальше.

Обнаружение и устранение логических ошибок

Логические ошибки нередко бывает очень трудно исправить. Поскольку программа не совершает аварийную остановку, то интерпретатор не выводит сообщение об ошибке, а значит, узнать от самого компьютера, в чем проблема, не удастся. Надо следить за поведением программы и тщательно исследовать ее код.

В данном случае решающая улика — результат, который программа выдает. Огромное число, выведенное на экран, — это явно не сумма всех чисел, которые ввел пользователь. Кропотливо сличив все числа, мы обнаружим, что так называемая «сумма» образована простым сцеплением слагаемых. Как так случилось? Если помните, функция `input()` возвращает значения строкового типа. Поэтому каждое из «чисел», которые пользователь ввел, рассматривается на самом деле как строка. Отсюда: все переменные в программе ссылаются на строковые, а не на числовые значения. Поэтому код

```
total = car + rent + jet + gifts + food + staff + guru + games
```

не суммирует числа, а сцепляет строки!

Итак, проблема найдена, но как же ее поправить? Очевидно, строки надо как-то преобразовать в числа, и после этого программа заработает желательным для нас образом. А можно ли выполнить такое преобразование? Поскольку нужно, значит, можно!

НА САМОМ ДЕЛЕ

Символ «+» обслуживает как строки, так и целые числа. Когда один и тот же оператор применяется к значениям разных типов, это называется перегрузкой оператора. Опыт программы «Рантъе» может заставить вас невзлюбить перегрузку операторов, но, в сущности, зря. Разве плохо, что соединение (сложение) строк осуществляется знаком «плюс»? Конечно, нет. Хорошо реализованная перегрузка операторов делает код яснее и изящнее.

Конвертация значений

Решение проблемы, с которой столкнулась программа «Рантье», — преобразовать строковые значения, которые возвращает функция `input()`, в числовые. Поскольку мы работаем с целыми суммами в долларах, имеет смысл конвертировать строки в целые числа.

Знакомство с программой «Рантье» (версия без ошибки)

Новая версия программы «Рантье» не содержит логической ошибки, которой страдал ее первый вариант. Результат работы обновленной программы можно увидеть на рис. 2.10.

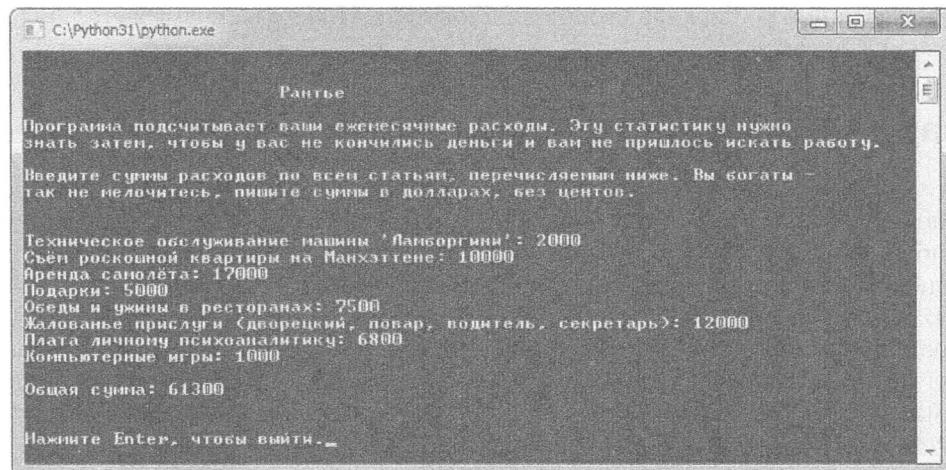


Рис. 2.10. 61 300 долларов в месяц — это гораздо больше похоже на правду

Теперь сумма издержек вычисляется верно. Код этой программы вы можете найти на сайте-помощнике в папке **Chapter 2** под названием `trust_fund_good.py`.

```
# Рантье (версия без ошибки)
# Демонстрирует type conversion
print(
    """
    Рантье
    Программа подсчитывает ваши ежемесячные расходы. Эту статистику нужно знать затем,
    чтобы у вас не кончились деньги и вам не пришлось искать работу.
    Введите суммы расходов по всем статьям, перечисляемым ниже. Вы богаты - так не мело-
    читесь, пишите суммы в долларах, без центов.
    """
)
```

```
car = input("Техническое обслуживание машины 'Ламборгини': ")
```

```

car = int(car)
rent = int(input("Съем роскошной квартиры на Манхэттене: "))
jet = int(input("Аренда самолета: "))
gifts = int(input("Подарки: "))
food = int(input("Обеды и ужины в ресторанах: "))
staff = int(input("Жалованье прислуги (дворецкий, повар, водитель, секретарь): "))
guru = int(input("Плата личному психоаналитику: "))
games = int(input("Компьютерные игры: "))
total = car + rent + jet + gifts + food + staff + guru + games
print("\nОбщая сумма:", total)
input("\n\nНажмите Enter, чтобы выйти.")

```

Преобразование строк в целые числа

Для преобразования значений из одного типа в другой существует несколько функций. В следующих строках показана та из них, которая приводит аргумент к целочисленному типу:

```

car = input("Техническое обслуживание машины 'Ламборгини': ")
car = int(car)

```

Первая из этих двух строк принимает пользовательский ввод (строку) и делает его значением переменной `car`. Во второй строке выполняется преобразование: функция `int()` принимает строку, на которую ссылается `car`, и возвращает ее преобразованной в целое число. Это новое значение присваивается той же переменной `car`.

В следующих семи строках другие издержки принимаются из пользовательского ввода и преобразуются в числа:

```

rent = int(input("Съем роскошной квартиры на Манхэттене: "))
jet = int(input("Аренда самолета: "))
gifts = int(input("Подарки: "))
food = int(input("Обеды и ужины в ресторанах: "))
staff = int(input("Жалованье прислуги (дворецкий, повар, водитель, секретарь): "))
guru = int(input("Плата личному психоаналитику: "))
games = int(input("Компьютерные игры: "))

```

Заметьте, что каждое присвоение выполняется в одну строку кода. А все потому, что вызовы функций `input()` и `int()` — вложенные. Когда говорят о **вложении** вызовов функций, имеют в виду, что один находится внутри другого. Это удобно, если значение, возвращаемое «внутренней» функцией, может выступать как аргумент для «внешней» функции. Здесь `input()` возвращает строковое значение, которое `int()` успешно преобразует в число.

В конструкции, присваивающей значение переменной `rent`, функция `input()` сначала спрашивает у пользователя, сколько тот платит за апартаменты. Пользователь что-то отвечает; это строковое значение `input()` возвратит, после чего программа применит функцию `int()` к этой строке как аргументу. В свою очередь, `int()` возвратит целое число. Это число и становится значением переменной `rent`. Точно так же присвоение выполняется и в следующих шести выражениях.

Есть и другие функции конвертации значений между типами. Некоторые из них перечислены в табл. 2.4.

Таблица 2.4. Некоторые функции, выполняющие преобразования типов

Функция	Описание	Пример	Результат
float(x)	Преобразует значение x в десятичную дробь	float("10.0")	10.0
int(x)	Преобразует значение x в целое число	int("10")	10
str(x)	Преобразует значение x в строку	str(10)	'10'

Составные операторы присвоения

Составных операторов присвоения существует множество, но в основе их всех лежит общая идея. Пусть, например, вы решили узнать, сколько наш рантье истратит в год на обеды и ужины. Чтобы рассчитать годовую сумму и сделать ее значением прежней переменной food, можно написать так:

```
food = food * 52
```

В этом коде значение food, увеличенное в 52 раза, снова присваивается переменной food. Так вот, существует более простая запись:

```
food *= 52
```

Оператор *= — один из составных операторов присвоения. Он тоже умножает значение food на 52 и вновь присваивает результат переменной food, но код с ним становится более компактным. Поскольку программисты то и дело присваивают переменным новые значения, вычисленные на основе существующих, составные операторы очень удобно «сокращают» решение типичных задач.

Некоторые из составных операторов присвоения перечислены в табл. 2.5.

Таблица 2.5. Полезные составные операторы присвоения

Оператор	Образец	Равносильная запись
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5

Вернемся к программе «Бесполезные факты»

Теперь вам известно все, что необходимо, чтобы спрограммировать приложение «Бесполезные факты» — центральный проект этой главы. Я буду демонстрировать и разбирать код не совсем обычным образом — не целиком, а небольшими порциями. Со всей программой можно ознакомиться на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 2. Файл называется `useless_trivia.py`.

Начальные комментарии

Хотя комментарии не оказывают никакого действия на работу программы, они составляют важную часть любого проекта. Вот, как обычно, несколько из них в начале программы:

```
# Бесполезные факты  
#  
# Узнает у пользователя его / ее личные данные и выдает несколько фактов  
# о нем / ней. Эти факты истинны, но совершенно бесполезны.
```

ПОДСКАЗКА

В зоне начальных комментариев опытные программисты зачастую отмечают, какие изменения вносились в код и когда. Это позволяет без труда проследить историю версий. Такая практика считается полезной при работе нескольких программистов над одним проектом.

Получение пользовательского ввода

С помощью функции `input()` программа узнает имя пользователя, его возраст и массу тела:

```
name = input("Привет. Как тебя зовут? ")  
age = input("Сколько тебе лет? ")  
age = int(age)  
weight = int(input("Хорошо, и последний вопрос. Сколько в тебе килограммов? "))
```

Запомните, что `input()` всегда возвращает строку. Поскольку значения `age` и `weight` мы будем интерпретировать как числа, их следует преобразовать. По отношению к переменной `age` процесс конвертации разбит на два этапа: сначала я присвоил переменной строковое значение, полученное от `input()`, а потом преобразовал эту строку в число и вновь присвоил той же переменной. В случае с переменной `weight` я решил обойтись одной строкой кода: вызовы функций вложены один в другой.

ПОДСКАЗКА

Я неспроста решил присваивать значения двумя разными способами: это призвано напомнить вам, что оба способа работают. Впрочем, обычно на практике последовательно пользуются одной из двух моделей присвоения.

Вывод name на экран в нижнем и верхнем регистре

Строковые методы позволяют следующему коду вывести на экран два варианта `name` — полностью строчными или прописными буквами:

```
print("\nЕсли бы поэт Каммингс адресовал тебе письмо, он бы обратился к тебе так: ",  
name.lower())  
print("А если бы это был рехнувшийся Каммингс, то так: ", name.upper())
```

Эдвард Каммингс — это, кстати, такой американский поэт-экспериментатор, который вообще не пользовался прописными буквами. Будь он жив сейчас, в письме по электронной почте он, скорее всего, не изменил бы своей привычке. Но будь он, кроме того, не вполне психически здоров, он мог бы поступить совсем наоборот, и его машинопись «закричала» бы буквами в верхнем регистре.

Пятикратный вывод имени

Далее на экран будет выведено пятикратно воспроизведенное имя пользователя:

```
called = name * 5
print("\nЕсли бы маленький ребенок решил привлечь твоё внимание",)
print("он произнес бы твоё имя так:")
print(called)
```

Переменной `called` присваивается пять раз повторенное значение переменной `name`. Потом программа выводит на экран сопроводительный текст и вслед за ним — `called`.

Подсчет количества секунд

Следующий код вычисляет и выводит на экран возраст пользователя в секундах:

```
seconds = age * 365 * 24 * 60 * 60
print("\nТвой нынешний возраст - свыше", seconds, "секунд.")
```

Поскольку в году 365 суток, в сутках 24 часа, в часе 60 минут, а в минуте 60 секунд, то оценку возраста пользователя в секундах можно получить умножением `age` на `365 * 24 * 60 * 60`. Полученное значение присваивается переменной `seconds`, которую выводит на экран очередная строка кода.

Вычисление значений `moon_weight` и `sun_weight`

Следующие четыре строки рассчитывают и выводят вес пользователя, каким он был бы на Луне и на Солнце:

```
moon_weight = weight / 6
print("\nЗнаете ли вы, что на Луне вы весили бы всего", moon_weight, "кг?")
sun_weight = weight * 27.1
print("А вот находясь на Солнце, вы бы весили", sun_weight, "кг. (Но, увы, это продолжалось бы недолго...)")
```

Известно, что сила тяготения на лунной поверхности в шесть раз меньше, чем на Земле. Поэтому переменной `moon_weight` присваивается значение, равное $1/6$ части `weight`. На поверхности Солнца гравитация примерно в 27,1 раза сильнее земной, поэтому `sun_weight` получает значение, равное `weight * 27.1`.

Ожидание выхода

Последняя строка кода ожидает нажатия `Enter`, в результате чего пользователь расщупается с программой:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

¹ Измерение веса в килограммах или, как у автора, в фунтах — неточность. От притяжения меняется не масса, а вес, то есть не устойчивый физический признак тела, а только ощущения человека. — *Примеч. пер.*

Резюме

Из этой главы вы узнали, как создавать строки в одиночных, двойных и тройных кавычках. Вы научились вставлять в строки специальные символы с помощью escape-последовательностей. Вы увидели, как склеивать и повторять строки. Вы усвоили разницу между двумя числовыми типами: целыми и десятичными дробями, а также научились работать с ними. Кроме того, вы теперь знаете, как преобразовывать значения из строковых в числовые и наоборот. Вы овладели техникой работы с переменными, познакомились с хранением информации в переменных и доступом к ней. Наконец, из этой главы вы узнали, как, принимая пользовательский ввод, сделать программу интерактивной.

Задачи

- Придумайте два списка: допустимых и недопустимых имен переменных. Объясните, почему каждое из имен допустимо или соответственно недопустимо. Затем придумайте еще два списка — «хороших» и «плохих» допустимых имен — и объясните свой выбор.
- Напишите программу, в окно которой пользователь сможет ввести названия двух своих любимых блюд. Программа должна склеять две эти строки и выводить полученную строку как название нового невиданного блюда.
- Напишите программу «Щедрый посетитель», в окно которой пользователь сможет ввести сумму счета за обед в ресторане. Программа должна выводить два значения: чаевые из расчета 15 и 20 % от указанной суммы.
- Напишите программу «Автодилер», в окно которой пользователь сможет ввести стоимость автомобиля без наценок. Программа должна прибавлять к ней несколько дополнительных сумм: налог, регистрационный сбор, агентский сбор, цену доставки машины по месту назначения. Пусть налог и регистрационный сбор вычисляются как доля от начальной стоимости, а остальные наценки будем считать фиксированными величинами. Окончательная цена автомобиля должна быть выведена на экран.

3 Ветвление, циклы с оператором `while` и псевдокод. Игра «Отгадай число»

Вплоть до настоящего времени вы писали программы с простым — последовательным — порядком исполнения: каждое выражение исполняется однократно по порядку следования. Если бы программировать можно было только так, стало бы очень трудно, если не прямо невозможно, создавать большие приложения. Из этой главы вы узнаете, как выборочно исполнять отдельные фрагменты кода и повторно запускать какие-либо части программы. Вы научитесь:

- генерировать случайные числа с помощью функций `randint()` и `randrange()`;
- пользоваться конструкциями `if` для исполнения кода при определенном условии;
- пользоваться конструкциями `else` для выбора вариантов продолжения при некотором условии;
- пользоваться конструкциями `elif`, чтобы осуществлять выбор продолжения на основе нескольких условий;
- организовывать циклы `while` для повторения частей программы;
- планировать свой труд с помощью псевдокода.

В этой главе мы создадим классическую игру на отгадывание числа. Для тех, кому не посчастливилось узнать ее в детстве, поясню: компьютер выбирает случайное число от 1 до 100, а игрок должен его отгадать за минимальное количество попыток. При каждой очередной попытке компьютер говорит игроку, как соизмеряется предложенный вариант с действительно загаданным числом: первое больше, второе больше или они совпадают. Как только игрок отгадывает число, игра заканчивается. Игровой процесс показан на рис. 3.1.

Генерирование случайных чисел

Пользователь ждет от программы стабильной и предсказуемой работы. Но по-настоящему захватывающим процесс взаимодействия человека и компьютера может сделать только элемент непредсказуемого: внезапная смена стратегии виртуального

противника, монстр, неожиданно появляющийся из соседней двери, и т. п. Этот элемент случая, (не)везения создают случайные числа, для генерации которых в Python существует простой механизм.

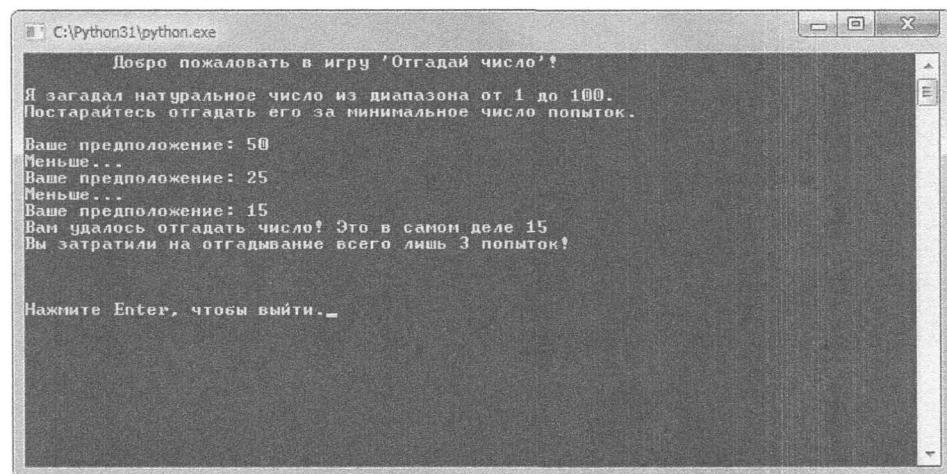


Рис. 3.1. Вот так рекорд — всего с трех попыток! Попробуйте отгадать быстрее

ЛОВУШКА

Python порождает случайные числа на основе формулы, так что они не на самом деле случайные, а, как говорят, псевдослучайные. Этот способ удобен для большинства приложений (кроме онлайновых казино). Настоящие случайные числа можно получить, например, с сайта <http://www.fourmilab.ch/hotbits/>. На этом сайте случайные числа генерируются на основе непредсказуемого естественного процесса радиоактивного распада.

Знакомство с программой «Кости»

«Кости» — симулятор броска костей в известной азартной игре. О самой игре, впрочем, совершенно ничего не надо знать, чтобы оценить работу программы. «Кости» всего лишь воспроизводят выпадение очков на двух шестигранных игральных костях: отображаются очки, выпавшие на первой, на второй кости и в сумме на обеих. Выпадающие очки определяет функция, генерирующая случайные числа. Работу программы иллюстрирует рис. 3.2.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 3; файл называется `craps_roller.py`.

```
# Кости
# Демонстрирует генерацию случайных чисел
import random
# создаем случайные числа из диапазона 1 - 6
die1 = random.randint(1, 6)
die2 = random.randrange(6) + 1
total = die1 + die2
print("При вашем броске выпало", die1, "и", die2, "очков, в сумме", total)
input("\n\nНажмите Enter, чтобы выйти.")
```

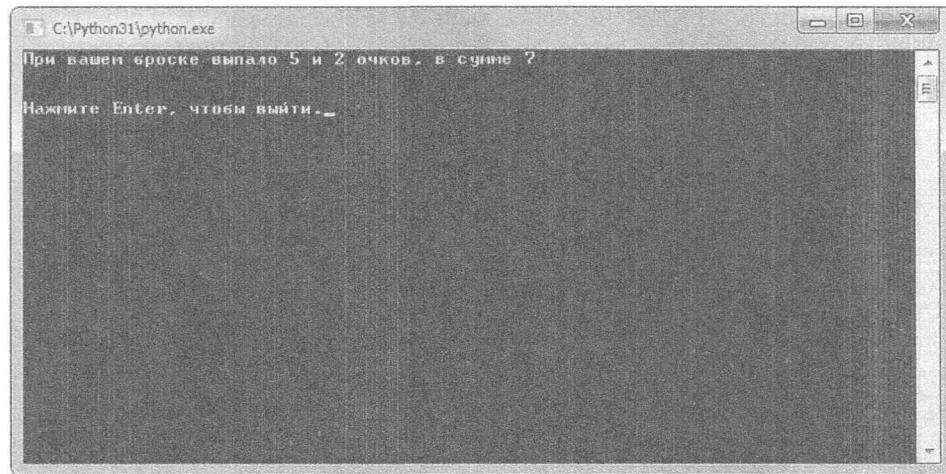


Рис. 3.2. Увы! Первыйбросок костейпринес мне 7 очков: это проигрыш

Импорт модуля random

В первой строке кода программы встречаем выражение со служебным словом `import`. Такие выражения позволяют импортировать или загружать модули, в данном случае модуль `random`:

```
import random
```

Модули – это файлы, содержащие код, пригодный для использования в других программах. Модуль обычно представляет собой совокупность функций, относящихся к одной и той же области. Так, модуль `random` содержит функции, связанные с генерацией случайных чисел и получением случайных результатов.

Если вообразить себе программу как дом или сооружение, то модули – это специальные наборы инструментов, за которыми при необходимости можноходить, чтобы воспользоваться ими в работе. Как видите, в качестве инструмента я достал модуль `random`.

Применение функции randint()

В модуле `random` есть функция `randint()`, которая возвращает случайное целое число. Программа «Кости» вызывает `randint()` следующим образом:

```
random.randint(1, 6)
```

Можно заметить, что вызов `randint()` осуществляется не напрямую, а через содержащий ее модуль `random`. Вообще функции из импортированных модулей вызываются именно так: `название_модуля.имя_функции`. Этот способ доступа к функциям называется *точечной нотацией*. Точечная нотация имеет тот же смысл, что и притяжательные конструкции в русском языке. Если мы говорим: «Машина Майка»,

мы имеем в виду, что машина принадлежит Майку. Запись `random.randint()` в точечной нотации значит, что функция `randint()` принадлежит модулю `random`. С помощью точечной нотации осуществляется доступ к разным составляющим импортированных модулей.

Функция `randint()` принимает два целочисленных аргумента и возвращает случайное целое число, лежащее в диапазоне между ними. Так, например, если передать функции аргументы 1 и 6, гарантированно будет возвращено 1, 2, 3, 4, 5 или 6. Это удовлетворительная модель броска шестигранной кости.

Применение функции `randrange()`

В модуле `random` есть также функция `randrange()`, которая возвращает случайное целое число. Есть несколько способов вызова `randrange()`, из которых самый простой состоит в передаче функции одного аргумента — положительного целого числа. При таком вызове функция возвратит случайное число из промежутка от 0 (включая) до введенного значения (не включая). Вот почему вызов `random.randrange(6)` возвратит 0, 1, 2, 3, 4 или 5. А как же 6? Заметим, что `randrange()` выбирает случайное число из шести возможных значений, первое из которых 0. Поэтому, чтобы в переменной `die2` оказалось нужное значение, я просто прибавляю 1 к результату работы функции:

```
die2 = random.randrange(6) + 1
```

В итоге `die2` получает значение 1, 2, 3, 4, 5 или 6.

ЛОВУШКА

В программе «Кости» применены обе функции: `randint()` и `randrange()`, чтобы продемонстрировать оба доступных способа генерации случайных чисел. В своей практике вы вольны выбирать ту из функций, которая лучше отвечает вашим нуждам.

Условные конструкции с if

Ветвление фундаментально важно для программирования в целом. Проще всего понимать ветвление как выбор одного из доступных путей. Конструкция `if` позволяет программе выбрать, исполнять определенный фрагмент кода или просто пропустить его, в зависимости от того, как указал программист.

Знакомство с программой «Пароль»

В программе «Пароль» конструкция `if` используется, чтобы воспроизвести процедуру входа в компьютерную систему с высокой степенью защиты. Пользователь может войти лишь в том случае, если введет верный пароль. Вид окна программы при ее запуске показан на рис. 3.3 и 3.4.

Код программы можно найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 3; файл называется `password.py`.

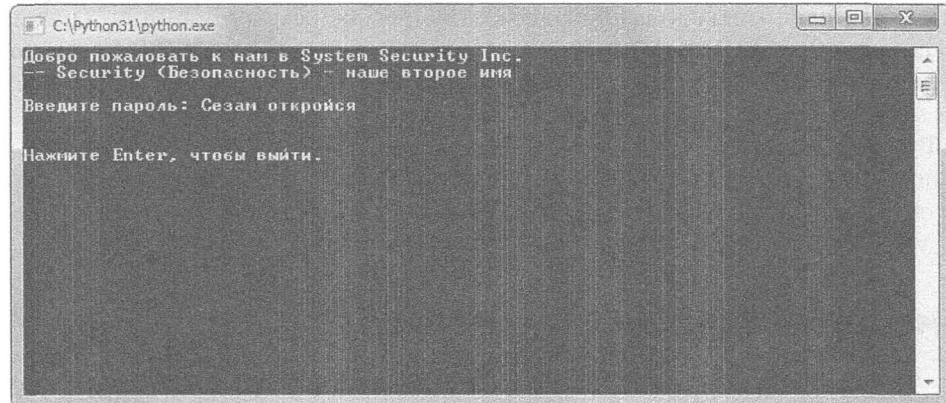


Рис. 3.3. Хе-хе, вам ни за что не взломать мою систему!

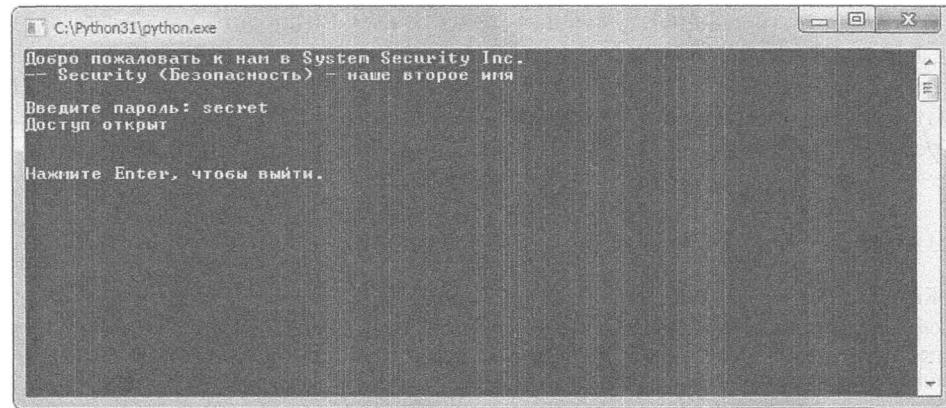


Рис. 3.4. Наверное, мне стоило выбрать пароль посложнее, чем «secret»?..

```
# Пароль
# Демонстрирует использование конструкции if
print('Добро пожаловать к нам в ООО "Системы безопасности".')
print('-- Безопасность - наше второе имя\n')
password = input("Введите пароль: ")
if password == "secret":
    print("Доступ открыт")
input("\n\nНажмите Enter, чтобы выйти.")
```

НА САМОМ ДЕЛЕ

Программа «Пароль» хороша лишь как демонстрация выполнения кода по условию, но не как образец системы безопасности. Ведь кто угодно может, заглянув в исходный код, обнаружить там пароль «secret». Для валидации пароля в настоящих системах безопасности обычно используется та или иная форма криптографии. Криптография — наука о шифровании, азы которой были созданы не одну тысячу лет назад. Она учит, как кодировать информацию, чтобы доступ к ней могли получить только определенные лица. Современная криптография — обширная область теоретической информатики. Некоторые ученые работают исключительно над криптографическими задачами.

Разбираемся, как работает конструкция if

Ключевое место в программе «Пароль» занимают следующие строки:

```
if password == "secret":  
    print("доступ открыт")
```

Эта часть кода удивительно проста. Понять ее смысл можно, читая код, как обычный английский текст. Если значение переменной `password` равно `"secret"`, на экран выводится текст `доступ открыт` и программа переходит к следующему выражению. Если же введенный пользователем пароль не равен `"secret"`, то текст не выводится и программа сразу, минуя строку, следующую за `if`, переходит далее.

Создание условий

При использовании конструкции `if` всегда задается *условие*, то есть такое выражение, которое истинно или ложно. Вы, конечно, хорошо знакомы с условиями, которых немало и в повседневной жизни: по сути, почти любое наше суждение может рассматриваться как условие. Например, сказав: «На улице сорокаградусная жара», мы создадим условие, ведь это суждение либо истинно, либо ложно.

В Python имеются встроенные значения истинности-ложности: `True` — правда, `aFalse`, как легко догадаться, — ложь. С любым условием может быть сопоставлено одно из этих двух значений. Так, в программе «Пароль» условие, содержащееся в конструкции `if`, — это `password == "secret"`. Это условие утверждает, что переменная `password` — это строка `"secret"`. Если значение `password` равно `"secret"`, то условие выполнено — `True`, иначе выражение в условии ложно — `False`.

Операторы сравнения

Часто условия создаются при сравнении величин, для чего существуют особые *операторы сравнения*. Один из таких операторов вы уже видели в коде программы «Пароль»: это оператор «равно», в символьической записи `==`.

ЛОВУШКА

Оператор «равно» состоит из двух знаков равенства. Если оставить из двух только один, система выдаст ошибку, потому что одиночный знак равенства используется в контексте присвоения значений переменным. Выражение `password = «secret»` — это конструкция, в которой переменной `password` присваивается значение `«secret»`. А вот `password == «secret»` — это условие, результат проверки которого имеет вид `True` или `False`. Несмотря на внешнее сходство, оператор присвоения и оператор «равно» ведут себя совершенно по-разному.

Кроме «равно», есть и другие операторы сравнения. Наиболее полезные из них перечислены в табл. 3.1.

Таблица 3.1. Операторы сравнения

Оператор	Значение	Пример	Истиность
<code>==</code>	Равно	<code>5 == 5</code>	<code>True</code>
<code>!=</code>	Не равно	<code>8 != 5</code>	<code>True</code>

Таблица 3.1 (продолжение)

Оператор	Значение	Пример	Истинность
>	Больше	3 > 10	False
<	Меньше	5 < 8	True
>=	Больше или равно	5 >= 10	False
<=	Меньше или равно	5 <= 5	True

ЛОВУШКА

Операторы сравнения позволяют сравнивать целые числа с целыми, десятичные дроби с десятичными дробями, а также целые числа с десятичными дробями. Более того, можно сравнивать строки; результат сравнения будет основан на алфавитном порядке. Так, например, «апельсин» < «яблоко» возвратит True, потому что по алфавиту «апельсин» стоит раньше «яблока» (ближе к началу словаря). Но в Python не всегда можно выполнить сравнение. Объекты разных типов, для которых не определено единое отношение порядка, не могут быть сравнены с помощью операторов <, <=, > и >=. К примеру, с их помощью нельзя сравнить целое число и строку. Попробуйте проверить условие 10 > «пять». Все, что вы получите, — большое-пребольшое сообщение об ошибке.

Создание блоков кода с помощью отступов

Вы могли заметить, что вторая строка конструкции if, сразу после условия, — `print("Доступ открыт")` — написана с отступом. Код с отступом превращается в **блок**. Соответственно, блоком называют одну или несколько идущих подряд строк с одинаковым отступом. Отступ организует код не только визуально, но и логически: блок — единая конструктивная единица.

Среди прочего, блоки могут использоваться в составе конструкций if. Отступами оформляются выражения (одно или несколько), исполняемые в том случае, если условие истинно. В программе «Пароль» единственный блок состоит из одной строки кода `print("Доступ открыт")`.

Поскольку в блоке может быть сколько угодно строк, вам полезно будет добавить в программу особое приветствие для тех пользователей, которые введут правильный пароль. Измененный блок станет выглядеть так:

```
if password == "secret":
    print("Доступ открыт")
    print("Добро пожаловать! Вы, наверное, очень важный господин..")
```

Правильно введя пароль, пользователь увидит на экране Доступ открыт и Добро пожаловать! Вы, наверное, очень важный господин.. Если же пользователь ошибется при вводе, то ни одного из этих двух сообщений система ему не покажет.

ПОДСКАЗКА

В обществе программирующих на Python идут жаркие дебаты по поводу того, пользоваться для отступов табуляцией или пробелами (и если пробелами, то в каком количестве). В сущности, это дело вкуса каждого из нас. Но есть две общие рекомендации. Во-первых, будьте последовательны: если вы пользуетесь двумя пробелами, то пользуйтесь ими повсеместно. Во-вторых, не комбинируйте пробелы с табуляцией. Даже такое сочетание позволяет сделать отступ, но впоследствии это может принести вам много неприятностей. Обычно отступ делается с помощью одного табуляционного символа или четырех пробелов — все на ваше усмотрение.

Создание собственных условных конструкций

Мы разобрали от начала до конца один образец условной конструкции. Подводя итоги, я хотел бы кратко проинструктировать вас относительно вашей дальнейшей самостоятельной практики. Условная конструкция — это служебное слово `if`, затем условие, двоеточие `:` и, наконец, блок из одного или нескольких выражений. Если условие истинно, то содержащийся в блоке код будет выполнен. Если же условие ложно, то программа переходит к исполнению строки кода, следующей за блоком условной конструкции.

Конструкции `if` с условием `else`

Иногда требуется, чтобы программа «выбирала», исходя из условия: если истинно — сделать одно, если ложно — сделать что-нибудь другое. Это позволяет выполнить условие `else` в конструкции `if`.

Знакомство с программой «Открыто/Закрыто»

Программа «Пароль», как помните, приветствовала пользователя, который правильно ввел пароль, а на ошибочный ввод никак не реагировала. В программе «Открыто/Закрыто» данная проблема решена с помощью условия `else`. Правильно введенный пароль, как и ранее, открывает доступ (см. рис. 3.4). Работу улучшенной версии иллюстрируют рис. 3.5.

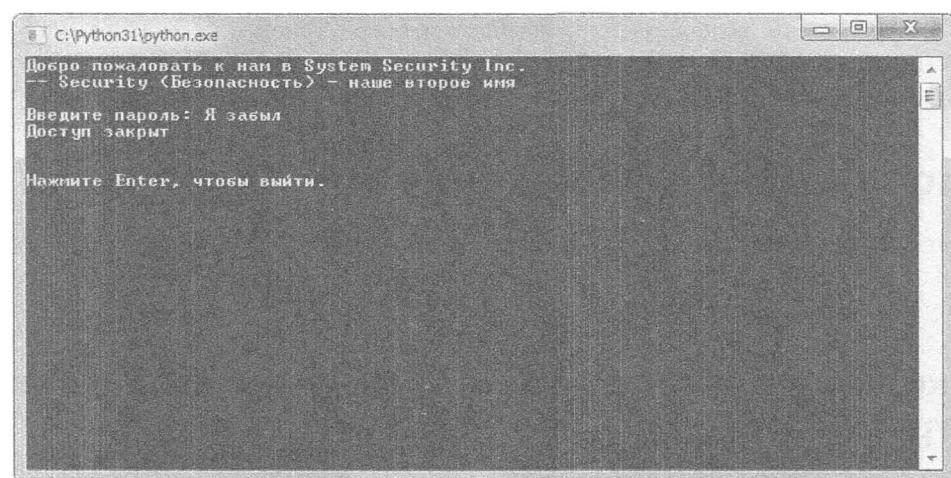


Рис. 3.5. Ошибка в пароле заставляет систему вывести леденящее душу сообщение «Доступ закрыт»

Код этой программы можно найти на сайте-помощнике в папке Chapter 3; файл называется `granted_or_denied.py`.

```
# Открыто/Закрыто
# Демонстрирует использование условия else
print('Добро пожаловать к нам в ООО "Системы безопасности".')
print('-- Безопасность - наше второе имя\n')
password = input("Введите пароль: ")
if password == "secret":
    print("Доступ открыт")
else:
    print("Доступ закрыт")
input("\n\nНажмите Enter, чтобы выйти.")
```

Разбираемся в условиях else

Здесь только одно отличие от программы «Пароль»: в конец конструкции `if` я добавил блок кода с предшествующим `else`:

```
if password == "secret":
    print("Доступ открыт")
else:
    print("Доступ закрыт")
```

Если значение переменной `password` равно `"secret"`, то программа выводит на экран сообщение `Доступ открыт`, как и ранее. В противном случае (благодаря условию `else`) на экран будет выведен текст `Доступ закрыт`.

В условной конструкции с `if` и `else` будет обязательно исполнен один из двух блоков кода: когда условие истинно — первый по порядку, непосредственно следующий за условием, а когда ложно — блок после `else`.

Условие `else` после блока с `if` создается так: служебное слово `else`, затем двоеточие и блок выражений. Условие `else` должно находиться внутри того же блока, что и предшествующая ему конструкция с `if` и условием, то есть отступ для `if` и для `else` должен быть одинаковым. Визуально они должны быть на одном уровне по вертикали.

Использование выражения elif

Ваша программа позволяет выбирать из нескольких возможностей, если дополнительно задействовать условия с `elif`. Они чрезвычайно удобны в том случае, когда одну переменную надо многократно сравнивать с разными величинами.

Знакомство с программой «Компьютерный датчик настроения»

В середине 1970-х годов — да-да, в прошлом веке! — был ажиотажный спрос на товар, именуемый «Кольцо настроения» (Mood Ring). Это было колечко, в которое был вставлен камень. Он мог менять цвет в зависимости от настроения носителя. «Компьютерный датчик настроения» — новое слово в психотехнике: теперь компьютер способен заглянуть в душу пользователю и сказать, какое у него настроение. На рис. 3.6 показано, какое у меня было настроение, когда я сочинял эту главу.

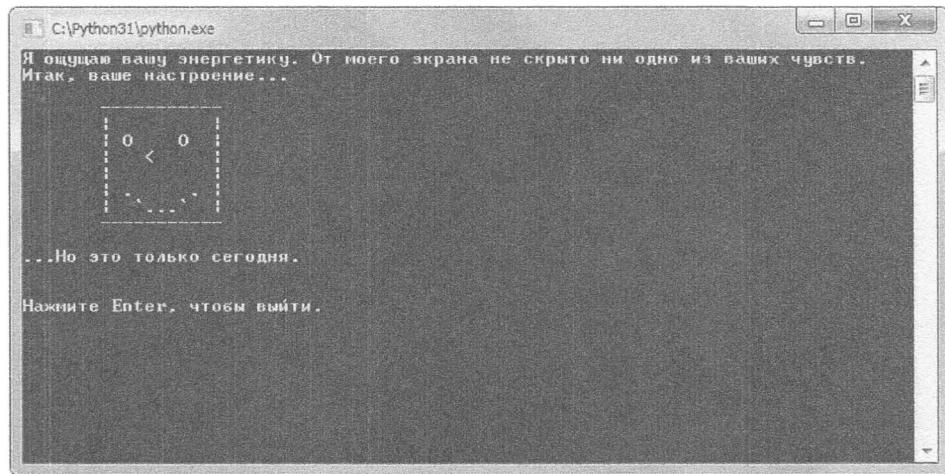


Рис. 3.6. Похоже, при разработке программы «Компьютерный датчик настроения» я был в отличном расположении духа

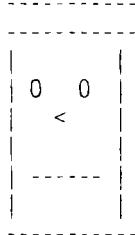
Пошутили, и хватит. Конечно, программа не проникает в глубины пользовательских эмоций с помощью электрических импульсов на коже, которые бы «ощущали» клавиатура и экран. «Компьютерный датчик настроения» всего лишь генерирует случайное число, в зависимости от значения которого конструкция `if` с условиями `elif` выбирает, какое из трех доступных псевдографических «лиц» отобразить. (Кстати, «Кольцо настроения» тоже не показывало истинное настроение. Это была игрушка на жидкких кристаллах, которые принимали тот или иной цвет под влиянием разной температуры тела.)

Код программы доступен на сайте-помощнике (www.courseptr.com/downloads) в папке **Chapter 3**; файл называется `mood_computer.py`.

```

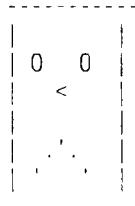
# Компьютерный датчик настроения
# Демонстрирует использование elif
import random
print("Я ощущаю вашу энергику. От моего экрана не скрыто ни одно из ваших чувств.")
print("Итак, ваше настроение...")
mood = random.randint(1, 3)
if mood == 1:
    # радостное
    print( \
"""
+---+
| 0   0 |
|       |
|     <  |
|       |
+---+
...
""")
```

```
elif mood == 2:
    # так себе
    print( \
    """
```



```
    """)
```

```
elif mood == 3:
    # прескверное
    print( \
    """
```



```
    """)
```

```
else:
```

```
    print("Не бывает такого настроения! (Должно быть, вы совершенно не в себе.)")
    print("...Но это только сегодня...")
    input("\n\nPress the enter key to exit.")
```

Разбираемся в условиях elif

Условная конструкция, использующая `elif`, может содержать последовательность условий, которые программа будет по порядку проверять. В «Датчике настроения» строк с условиями всего три:

- if mood == 0:
- elif mood == 1:
- elif mood == 2:

Заметьте, что первое условие задано с помощью `if`, а все остальные проверяются как `elif` (сокращение от `else if`). Допускается сколь угодно много условий `elif` подряд.

Отдельно выписав условия, мы лучше поймем задачу всей структуры: сравнить переменную `mood` с тремя разными значениями. Сначала программа проверяет, равно ли значение `mood` единице. Если это так, на экран выводится радостное лицо, а если нет, то программа переходит к следующему условию и проверяет, равно ли

значение mood двум. Если это так, будет выведено спокойное лицо, а если нет, то программа, наконец, проверяет, равно ли значение mood трем. При положительном результате проверки выводится печальное лицо.

ЛОВУШКА

Важно помнить, что как только в конструкции if с условиями elif очередное условие оказывается истинным, программа исполняет соответствующий блок кода и переходит к следующему выражению. Таким образом, даже если несколько условий истинны, исполнению подлежит все равно максимум один блок кода. Для «Датчика настроения» это маловажно, ведь переменная mood может принимать только одно численное значение, поэтому истинно будет лишь одно из условий. Но не следует забывать об описанной особенности в конструкциях, где может одновременно выполняться несколько условий. Тогда исполнению подлежит первый по порядку блок с истинным условием.

Если бы ни одно из условий для переменной mood не выполнялось, сработал бы заключительный блок под оператором else и на экран был бы выведен текст: Не бывает такого настроения! (Должно быть.. вы совершенно не в себе.). Поскольку значение mood всегда равно 1, 2 или 3, этого никогда не произойдет. Я создал это условие на всякий случай. Это не обязательно было делать: финальное else-выражение добавлено просто по желанию.

ЛОВУШКА

Несмотря на такую произвольность, заключительное условие else часто оказывается удобным решением: оно выполняется абсолютно во всех случаях, когда не истинно ни одно из созданных вами условий. Даже если вам кажется, что какое-либо из условий всегда истинно, заключительную конструкцию else все же можно применить для «невозможного» случая, как это показано в моем коде.

Итак, вы познакомились с тремя схожими, но неодинаково мощными конструкциями ветвления. Кратким инструктирующим напоминанием послужит вам табл. 3.2.

Таблица 3.2. Операторы ветвления — краткое резюме

Выражение	Описание
if <условие>: <блок>	Простейшая условная конструкция. Если <условие> истинно, то <блок> выполняется, в противном случае — пропускается
if <условие>: <блок 1> else: <блок 2>	Условная конструкция с условием else. Если <условие> истинно, выполняется <блок 1>, если ложно — <блок 2>
if <условие 1>: <блок 1> elif <условие 2>: <блок 2> ... elif <условие N>: <блок N> else: <блок N+1>	Условная конструкция с дополнительными условиями elif и обязательным else в конце. Будет исполнен блок после первого же условия, которое окажется истинным. Если ни одно условие не принимает значение True, будет исполнен блок после заключительного else

Создание циклов с использованием while

Циклы есть повсюду вокруг нас. Даже на бутылке с шампунем можно обнаружить циклическую инструкцию: «До тех пор пока ваши волосы остаются грязными, повторно промывайте и намыливайте их». В сущности, идея очень проста: пока истинно какое-либо условие, повторять какие-либо операции, но, несмотря на свою простоту, в программировании циклы стали чрезвычайно мощным инструментом. Удобно, например, основать на цикле компьютерный вариант телегры-викторины: программа может продолжать игровой процесс до тех пор, пока еще есть вопросы. Другой вариант — банковское приложение. Можно велеть программе запрашивать пользователя о его номере счета до тех пор, пока он не введет корректный номер. Именно такую функциональность и обеспечивают циклы с оператором `while`.

Знакомство с программой «Симулятор трехлетнего ребенка»

В современном мире приходится жить на безумной скорости, и многим из нас не хватает времени на общение с детьми. Юрист, допоздна занятый на работе, никак не может выкроить вечер на то, чтобы побеседовать с сынишкой, а коммивояжер, который все время в пути, тщетно надеется повидать наконец свою маленькую племянницу. Вот та проблема, решению которой призвана поспособствовать программа «Симулятор трехлетнего ребенка». С помощью цикла `while` она воссоздает разговор с маленьким ребенком. Как работает программа, показано на рис. 3.7.

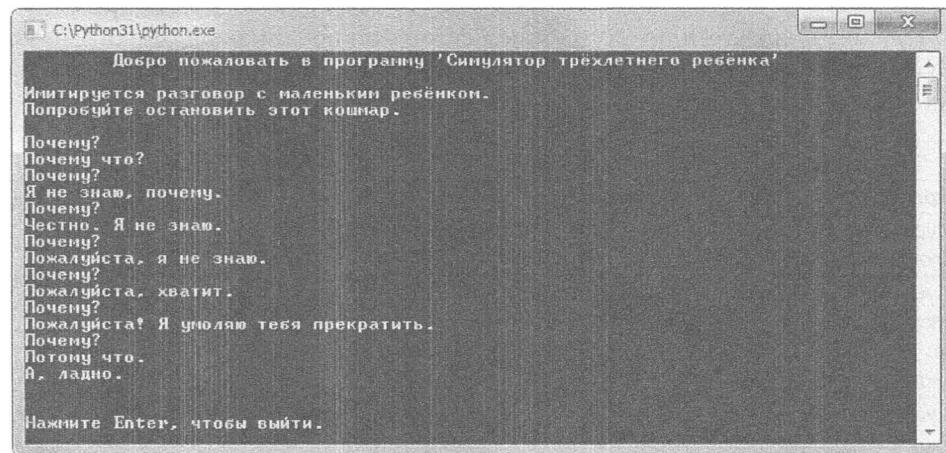


Рис. 3.7. Если вам когда-либо приходилось заботиться о трехлетнем ребенке, то воспоминания сразу оживут в вашей душе

Как видите, программа вновь и вновь спрашивает: «Почему?» — до тех пор пока пользователь не введет ответ «Потому что». Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 3. Файл называется `three_year-old.py`.

```
# Симулятор трехлетнего ребенка
# Демонстрирует работу цикла while
print("\tДобро пожаловать в программу 'Симулятор трехлетнего ребенка'\n")
print("Имитируется разговор с маленьким ребенком.")
print("Попробуйте остановить этот кошмар.\n")
response = ""
while response != "Потому что.__":
    response = input("Почему?\n")
print("А, ладно.")
input("\n\nНажмите Enter, чтобы выйти.")
```

Разбираемся в работе цикла while

Цикл в программе «Симулятор трехлетнего ребенка» состоит всего из двух строк:

```
while response != "Потому что.__":
    response = input("Почему?\n")
```

Формат цикла `while` может показаться вам знакомым, и это неспроста. Цикл удивительно похож на своего «ближайшего родственника» — конструкцию `if`. Различие только в том, что на месте `if` стоит `while`, но сходство не ограничивается лишь поверхностным подобием. Блок кода (в частности, внутри цикла — так называемое тело цикла) будет выполнен, если условие истинно, причем в конструкции `while` компьютер проверяет условие на истинность снова и снова, цикл за циклом, пока оно не окажется ложным. Отсюда сам термин «цикль».

В данном случае тело цикла представляет собой одну строку кода: `response = input("Почему?\n")`. Этот код будет раз за разом исполняться все то время, пока пользователь вводит не «Потому что.», а какой-либо иной вариант ответа. Когда наконец условие `response != "Потому что."` оказывается ложным, цикл благополучно завершается и программа переходит к следующему выражению: `print("А, ладно.")`.

Инициализация управляющей переменной

Зачастую контроль над циклом `while` осуществляется *управляющая переменная*. Это переменная, фигурирующая в условии: ее сопоставляют с некоторым значением или значениями. Можно думать об управляющей переменной как о часовом, который до поры до времени заграждает выход из кода, образующего тело цикла. В «Симуляторе трехлетнего ребенка» управляющая переменная — `response`. В условии, образующем цикл, она сопоставляется со строкой «Потому что.» каждый раз, прежде чем снова запустить блок на исполнение.

Управляющую переменную следует инициализировать. Как правило, инициализация происходит непосредственно перед началом цикла. Так поступил и я:

```
response = ""
```

ЛОВУШКА

Если управляющая переменная не будет иметь никакого значения в тот момент, когда она впервые сравнивается с образцом, то программа выдаст ошибку.

Обычно управляющей переменной сначала приписывают пустое значение какого-либо вида. Так, я приравнял response к пустой строке "". Можно было сделать начальным значением response любое слово, даже "подстаканник", и программа работала бы совершенно так же, но код содержал бы в себе ненужное усложнение.

Проверка значения управляющей переменной

Убедитесь, что условие цикла while изначально ложно. В противном случае тело цикла никогда не будет исполнено. Рассмотрим, например, немного измененный вариант цикла, с которым мы работаем:

```
response = "Потому что."
while response != "Потому что.":
    response = input("Почему?\n")
```

Поскольку переменная response равна "Потому что." как раз в момент запуска цикла, тело цикла так и не будет исполнено.

Изменение значения управляющей переменной

После того как вы создадите условие, инициализируете управляющую переменную и убедитесь, что блок кода не останется неисполненным, в вашем распоряжении окажется работающий цикл. Осталось сделать так, чтобы цикл когда-нибудь закончился.

Если цикл (без внешнего вмешательства) не остановится никогда, говорят, что это **бесконечный цикл**. Добро пожаловать в наш клуб неудачников! Рано или поздно любому программисту суждено по неосторожности создать бесконечный цикл и наблюдать за тем, как программа без устали повторяет какую-либо операцию или просто сразу «виснет».

Вот несложный пример бесконечного цикла:

```
counter = 0
while counter <= 10
    print(counter)
```

Вероятно, программист хотел бы, чтобы этот код выводил на экран числа от 0 до 10. Увы, выводится только 0, и притом неопределенно много раз. Программист забыл написать команду, которая бы изменяла значение управляющей переменной counter в теле цикла. Запомните: исполнение циклизированного кода должно менять (хотя бы в некоторых случаях) значение управляющей переменной. Иначе вы получите бесконечный цикл.

Борьба с бесконечными циклами

В бесконечных циклах одного типа значение управляющей переменной никогда не меняется, как было только что показано. Но есть и более хитроумная альтернатива. В следующей нашей программе значение управляющей переменной меняется в теле цикла, но что-то все же не в порядке: цикл бесконечен. Сумеете ли вы распознать проблему прежде, чем я объясню ее суть?

Знакомство с программой «Проигранное сражение»

«Проигранное сражение» — это история последней доблестной схватки героя с огромной армией троллей. Такой сценарий можно повсеместно найти в ролевых играх. Повествование о битве ведется последовательно, «удар за ударом»: герой уничтожает тролля, но затем сам получает повреждения и т. д. Программа должна закончиться смертью героя. Или не должна?

Код этой игры можно найти на сайте-помощнике в папке Chapter 3; файл называется `losing_battle-bad.py`.

```
# Проигранное сражение
# Демонстрирует тот самый ненавистный нам бесконечный цикл
print("Вашего героя окружила огромная армия троллей.")
print("Смрадными зелеными трупами врагов устланы все окрестные поля.")
print("Одинокий герой достает меч из ножен, готовясь к последней битве в своей жизни.\n")
health = 10
trolls = 0
damage = 3
while health != 0:
    trolls += 1
    health -= damage
    print("Взмахнув мечом, ваш герой истребляет злобного тролля, "
          "но сам получает повреждений на", damage, "очков.\n")
print("Ваш герой доблестно сражался и убил", trolls, "троллей.")
print("Но увы! Он пал на поле боя.")
input("\n\nНажмите Enter, чтобы выйти.")
```

Как работает эта программа, показано на рис. 3.8. Получился бесконечный цикл, который я вынужден остановить нажатием **Ctrl+C**.

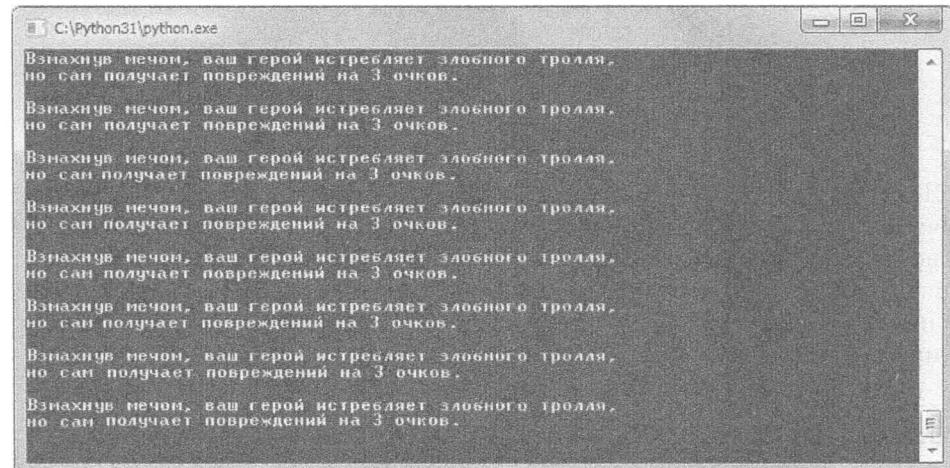


Рис. 3.8. Кажется, ваш герой бессмертен. Придется остановить программу принудительно

Что же происходит?

Трассировка программы

Похоже на то, что в коде содержится логическая ошибка. Чтобы найти ее, удобно было бы трассировать исполнение программы. При *трассировке* вы симулируете работу программы, делая все, что должна делать она: выполняются все команды и контролируются значения всех переменных. Такое «пошаговое» выполнение программы помогает выяснить, что именно происходит в каждый момент ее работы, и установить, какими оплошностями в коде вызвана ошибка.

Простейшие инструменты трассировки — карандаш и бумага. Я разделил страницу на четыре колонки, по одной для каждой переменной и каждого условия. Сначала вид страницы был таким:

```
health    trolls    damage    health != 0
```

Первая проверка условия цикла `while` позволяет вписать первый ряд значений:

health	trolls	damage	health != 0
10	0	3	True

Поскольку условие истинно (`True`), то после первой проверки условия тело цикла будет выполнено. Вслед за тем, в момент второй проверки условия, трасса будет выглядеть уже так:

health	trolls	damage	health != 0
10	0	3	True
7	1	3	True

Еще несколько итераций — и трасса примет такой вид:

health	trolls	damage	health != 0
10	0	3	True
7	1	3	True
4	2	3	True
1	3	3	True
-2	4	3	True
-5	5	3	True
-7	6	3	True

Я остановил трассировку, заподозрив здесь бесконечный цикл. В самом деле, значение переменной `health` в последних трех строках отрицательно (то есть не равно 0), а условие по-прежнему выполняется. Значит, проблема в том, что переменная `health` с каждым последующим исполнением тела цикла убывает, но никогда не обращается в ноль, вследствие чего организующее цикл условие все время истинно. Это и делает цикл бесконечным.

Условия, которые могут становиться ложными

Итак, надо позаботиться не только о том, чтобы условие цикла `while` хоть когда-нибудь выполнялось, но и о том, чтобы оно в некоторых случаях не выполнялось. Иначе шансы создать бесконечный цикл останутся высокими. Впрочем, программу

«Проигранное сражение» легко исправить. Стока с условием должна принять такой вид:

```
while health > 0:
```

Код обновленной программы можно найти на сайте-помощнике в папке Chapter 3; файл называется `losing_battle-good.py`. Теперь, как только переменная `health` примет отрицательное значение, условие перестанет выполняться и цикл будет завершен. Убедимся в этом, выполнив трассировку программы с новым условием:

health	trolls	damage	health > 0
10	0	3	True
7	1	3	True
4	2	3	True
1	3	3	True
-2	4	3	False

Вот теперь все работает надлежащим образом. Запуск отлаженной программы показан на рис. 3.9.

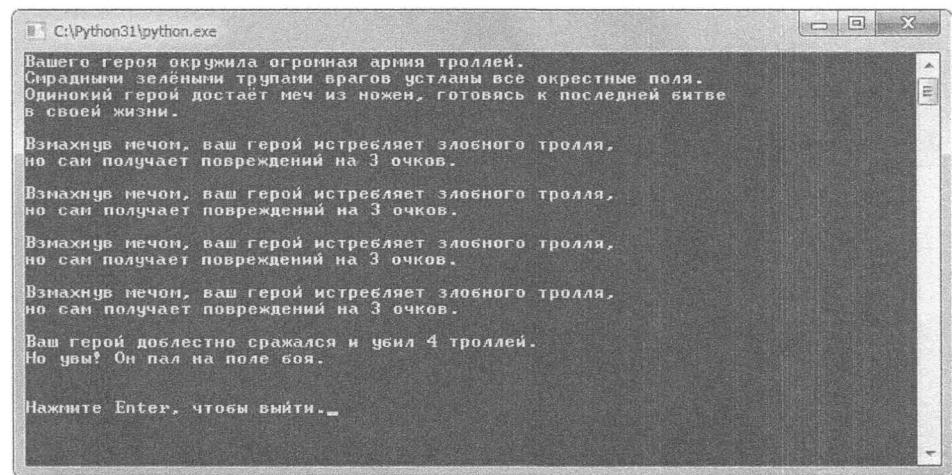


Рис. 3.9. Работа программы корректна, бесконечный цикл устранен. Судьба героя, однако, оказывается не столь радужной, как прежде

Значения как условия

Если бы я попросил вас оценить значение выражения $35 + 2$, вы бы не задумываясь ответили: 37. Но как оценить, истинно 37 или ложно? В Python разрешено рассматривать любое значение как истинное или ложное. Да, совершенно любое значение любого типа подлежит рассмотрению в таком ключе. И 2749, и 8.6, и "подстаканник", и 0, и "" можно интерпретировать как истину (`True`) или ложь (`False`). В этом логика языка может показаться вам странной, но по сути все довольно несложно: выбор `True` или `False` регламентирован простыми правилами. Важно, что логическая интерпретация значений позволяет сделать условные конструкции элегантнее.

Знакомство с программой «Метрдотель»

Если вас еще никогда не выпроваживали из дорогого французского ресторана, то наша следующая программа даст вам такой опыт. «Метрдотель» пригласит вас отужинать и затем поинтересуется, сколько вы можете пожертвовать ему за труды. Если вы готовы раскошелиться всего лишь на 0 долларов, вас закономерно оставят без внимания. Если же вы внесете какую-либо другую сумму, то столик в ресторане будет ожидать вас. Работу программы иллюстрируют рис. 3.10 и 3.11.

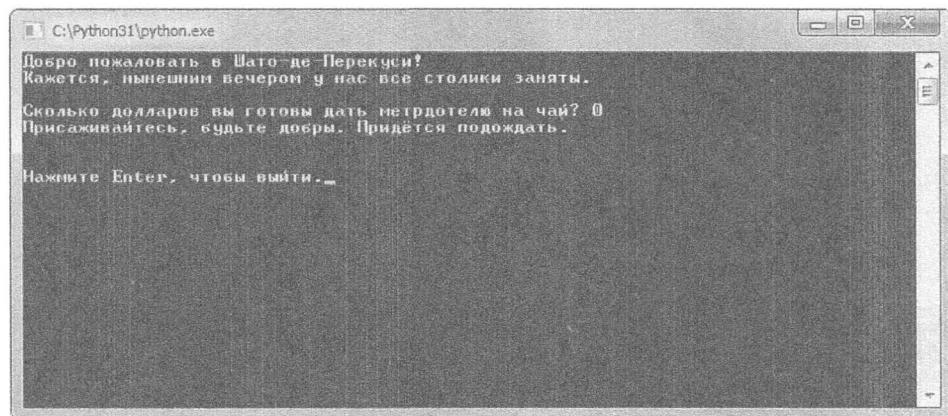


Рис. 3.10. Если оставить метрдотеля без чаевых, то в ресторане не найдется для вас свободного столика

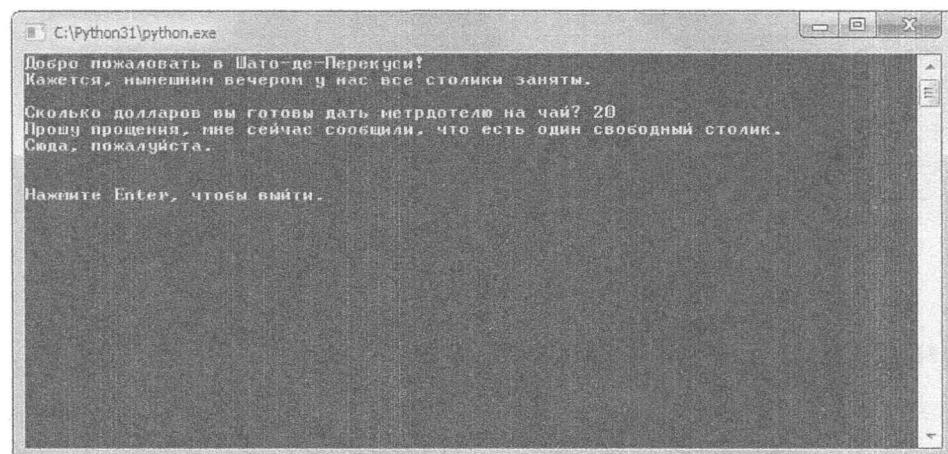


Рис. 3.11. «Излечить» метрдотеля от забывчивости оказалось удивительно просто — всего одной купюрой!

Наблюдая за работой программы, вы, скорее всего, не впечатлитесь: ведь что-то похожее вам уже приходилось выполнять. Но разница в том, что сейчас мы пользуемся отнюдь не оператором сравнения. В качестве условия программа «Метрдо-

тель» рассматривает саму внесенную сумму (значение — количество денег). Код программы можно найти на сайте-помощнике (www.courseptr.com/downloads) в папке **Chapter 3**; файл называется `maitre_d.py`.

```
# Метрдотель
# Демонстрирует условную интерпретацию значений
print("Добро пожаловать в Шато-де-Перекуси!")
print("Кажется, нынешним вечером у нас все столики заняты.\n")
money = int(input("Сколько долларов вы готовы дать метрдотелю на чай? "))
if money:
    print("Прошу прощения, мне сейчас сообщили, что есть один свободный столик."
          "Сюда, пожалуйста.")
else:
    print("Присаживайтесь, будьте добры. Придется подождать.")
input("\n\nНажмите Enter, чтобы выйти.")
```

Истинные и ложные значения

Новая для вас идея вводится в следующей строке кода:

```
if money:
```

Заметьте, что `money` не сравнивается ни с каким значением: сама переменная `money` выступает как условие. Когда число проверяется в качестве условия, 0 полагают ложным значением, а любое другое число — истинным. Значит, мы могли бы написать иначе:

```
if money != 0:
```

Но первая версия проще, элегантнее и интуитивно яснее. Она естественнее читается. «Если есть деньги» — так можно перевести этот код на русский язык.

Правила, по которым определяется логическая истинность-ложность значения, просты. Общий принцип: переменная с пустым или нулевым значением ложна, а любая другая — истинна. Поэтому, например, 0 трактуется как невыполненоное условие, а любое иное число — как выполненное условие. Пустая строка "" ложна, а любая иная строка — истинна. Как видите, большинство значений истинны и лишь пустые/нулевые — ложны. Вам еще предстоит убедиться, что проверка на пустоту — распространенная операция, вследствие чего концепция значений как условий работает в множестве программ.

Осталось заметить, что, введя отрицательное количество долларов, вы все же заслужите благосклонность метрдотеля. Запомните: из чисел только 0 логически должен, а все отрицательные числа, как и положительные, — истинны.

Намеренное создание бесконечных циклов

После раздела «Борьба с бесконечными циклами» вас должен немало озадачить раздел о бесконечных циклах, вводимых *намеренно*. Разве не всегда ошибочен цикл, количество итераций которого неопределенно велико?

На этот вопрос следует ответить так. «Действительно бесконечный» цикл, который в самом деле никогда не заканчивается, — это, бесспорно, логическая ошибка. Напротив, «намеренно бесконечный» цикл несет условие выхода в себе самом. Понять работу такого цикла лучше всего из примера.

Знакомство с программой «Привередливая считалка»

В программе «Привередливая считалка» выполняется счет от 1 до 10 с помощью «намеренно бесконечного» цикла. Программа названа привередливой потому, что она очень не любит число 5 и при счете пропускает его. Как работает программа, продемонстрировано на рис. 3.12.

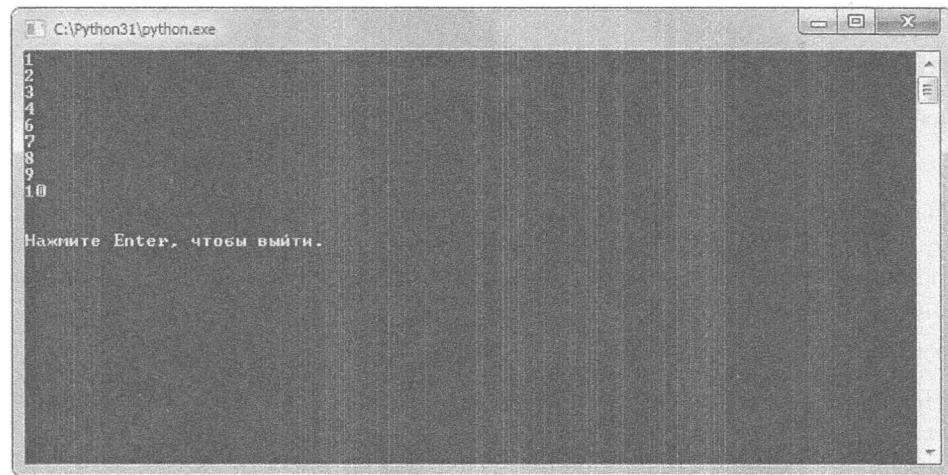


Рис. 3.12. Число 5 программа пропускает благодаря команде `continue`, а выход из цикла осуществляется командой `break`

Код этой программы можно найти на сайте-помощнике в папке Chapter 3; файл называется `finicky_counter.py`.

```
# Привередливая считалка
# Демонстрирует работу команд break и continue
count = 0
while True:
    count += 1
    # завершить цикл, если count принимает значение больше 10
    if count > 10:
        break
    # пропустить 5
    if count == 5:
        continue
    print(count)
input("\n\nНажмите Enter, чтобы выйти.")
```

Выход из цикла с помощью команды `break`

В моем примере цикл организуется условием:

`while True:`

Это значит, что цикл будет продолжаться до тех пор, пока команда в самом теле цикла не завершит его при каком-либо условии. В моей программе и условие и команда присутствуют:

```
# завершить цикл, если count принимает значение больше 10
if count > 10:
    break
```

Поскольку каждый раз при исполнении тела цикла переменная `count` сразу же увеличивается на 1, рано или поздно ее значение достигнет 11. Тогда-то команда `break` («прервать», то есть остановить цикл) и положит конец счету.

Команда `continue` и возврат к началу цикла

Непосредственно перед выводом значения `count` на экран в коде идут строки:

```
# пропустить 5
if count == 5:
    continue
```

Команда `continue` означает, что нужно вернуться к началу цикла. При возврате в очередной раз будет проверено условие цикла, и, если оно окажется истинным, тело цикла снова будет выполняться. Поэтому, когда переменная `count` принимает значение 5, программа, не доходя до выражения `print(count)`, возвращается в начало цикла. Пятерка так никогда и не выводится на экран.

Как пользоваться командами `break` и `continue`

Операторы `break` и `continue` применимы в любом цикле — не только в «намеренно бесконечном». Но ими следует пользоваться разумно. Для некоторых, в том числе и для вас, будет непростой задачей следить за работой цикла, который перегружен командами `break` и `continue`; станет гораздо труднее понять, при каких условиях он завершится. Да и жизненной необходимости в `break` и `continue`, в общем-то, нет: любой цикл, в котором используются эти операторы, может быть переписан и без них.

Однако иногда в Python «намеренно бесконечный» цикл выглядит яснее обычного цикла. В таких случаях, то есть когда условие традиционного типа существенно менее удобно, многие программисты и пользуются «намеренно бесконечными» циклами.

Составные условия

Вплоть до настоящего времени мы пользовались только конструкциями, в которых сравниваются ровно два значения. Это так называемые «простые» условия.

Для большего удобства вы можете сочетать простые условия, пользуясь *логическими операторами*. Такие сочетания принято называть «*составными*» условиями. Составное условие позволяет программе принять решение, руководствуясь несколькими значениями.

Знакомство с программой «Эксклюзивная сеть»

Человеку с улицы закрытый клуб не сулит ничего хорошего. Я создал программу «Эксклюзивная сеть» — симулятор элитной компьютерной сети, в которой зарегистрированы лишь избранные. Это избранное меньшинство состоит из меня самого и нескольких крупнейших разработчиков игр (недурная компания собралась, правда?).

Как и в настоящих компьютерных системах, в этой программе каждый пользователь должен ввести свое имя (логин) и пароль. Зарегистрированный пользователь может войти в сеть, только введя корректную пару «логин — пароль». Если вход в сеть выполнен успешно, то система персонально приветствует пользователя.

Кроме того, как и в настоящих компьютерных системах, имеется несколько уровней защиты. Не будучи убежденным снобом, я позволяю заходить в сеть и гостям, но уровень защиты у них будет самым низким. Работу программы иллюстрируют рис. 3.13–3.15.

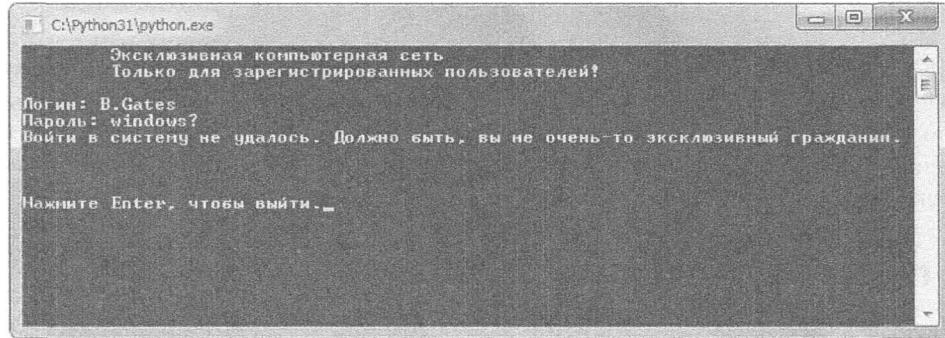


Рис. 3.13. Если вы не зарегистрированный пользователь и не гость, в систему вы не попадете

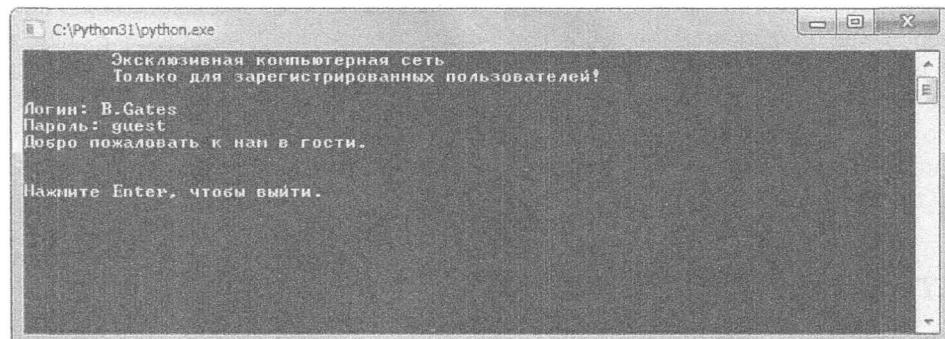


Рис. 3.14. Гостям позволено входить, но уровень их защиты чрезвычайно низок

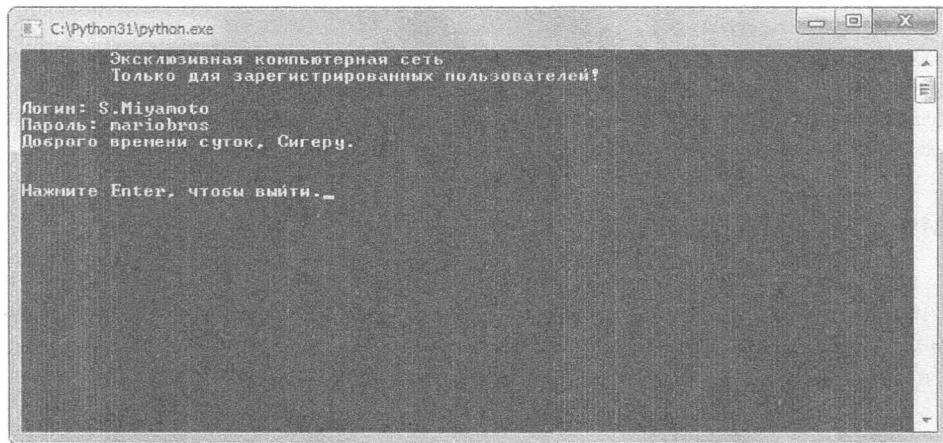


Рис. 3.15. Похоже на то, что зашел один из полноправных пользователей

Код программы можно найти на сайте-помощнике в папке Chapter 3; файл называется exclusive_network.py.

```
# Эксклюзивная сеть
# Демонстрирует составные условия и логические операторы
print("\tЭксклюзивная компьютерная сеть")
print("\tТолько для зарегистрированных пользователей!\n")
security = 0
username = ""
while not username:
    username = input("Логин: ")
password = ""
while not password:
    password = input("Пароль: ")
if username == "M.Dawson" and password == "secret":
    print("Привет, Майк.")
    security = 5
elif username == "S.Meier" and password == "civilization":
    print("Здравствуй, Сид.")
    security = 3
elif username == "S.Miyamoto" and password == "mariobros":
    print("Доброго времени суток, Сигеру.")
    security = 3
elif username == "W.Wright" and password == "thesims":
    print("Как дела, Уилл?")
    security = 3
elif username == "guest" or password == "guest":
    print("Добро пожаловать к нам в гости.")
    security = 1
else:
    print("Войти в систему не удалось. Должно быть, вы не очень-то эксклюзивный гражданин.\n")
input("\n\nНажмите Enter, чтобы выйти.")
```

НА САМОМ ДЕЛЕ

Чтобы реализовать настоящую частную сеть, ни в коем случае нельзя указывать логины и пароли прямо в коде. Следует воспользоваться какой-либо системой управления базами данных (СУБД). СУБД — это средство организаций и редактирования массивов тематически близкой информации, а также доступа к ним. Будучи достаточно мощными, СУБД могут быстро и безопасно работать с тысячами и даже миллионами пар «логин — пароль».

Логический оператор not

Я хотел сделать так, чтобы пользователь обязательно что-то вводил в качестве логина и пароля, чтобы простое нажатие `Enter` (то есть создание пустой строки) не помогало. Нужен был цикл, который запрашивал бы пользователя о его логине до тех пор, пока пользователь не введет какие-нибудь символы:

```
username = ""
while not username:
    username = input("Логин: ")
```

В условии, организующем этот цикл `while`, я применил логический оператор `not`. Он значит то же, что и английское слово `not` («не»). Если перед английским выражением поставить `not`, то полученная фраза будет по смыслу противоположна исходной. В языке Python поместить `not` перед условием — значит создать новое условие, проверка которого на истинность будет давать противоположный результат по сравнению с прежним. Таким образом, `not username` истинно тогда и только тогда, когда `username` ложно, а `not username` ложно тогда и только тогда, когда `username` истинно. Вот табличное представление этого:

<code>username</code>	<code>not username</code>
<code>True</code>	<code>False</code>
<code>False</code>	<code>True</code>

Начальное значение переменной `username` в программе — пустая строка, что логически интерпретируется как «ложь». Тогда `not username` окажется истинным и тело цикла будет выполнено в первый раз. Новое значение `username` программа принимает из пользовательского ввода. Если пользователь просто нажимает `Enter`, то значением `username` остается, как и ранее, пустая строка. Вследствие этого `not username` сохраняет истинность и цикл повторяется. Значит, пока пользователь просто нажимает `Enter`, цикл не устанет запрашивать логин. Но как только пользователь что-нибудь введет, `username` перестанет быть равным пустой строке и превратится в истинное значение, а `not username`, следовательно, — в ложное. Как и было задумано, на этом цикл прекратится. То же самое программа проделывает с переменной `password`.

Логический оператор and

Если зарегистрированный пользователь хочет войти в эксклюзивную сеть, он должен ввести такие логин и пароль, которые вместе образуют пару, известную системе. Так, Сиду Мейеру, чтобы войти, надо было бы ввести `S. Meier` в качестве логина и `civilization` в качестве пароля. Разрушив эту пару, Сид не смог бы войти. К при-

меру, не сработают пары S.Meier и mariobros, M.Dawson и civilization и даже civilization и S.Meier. За тем, чтобы S.Meier было введено в качестве имени пользователя, а civilization — в качестве пароля, программа следит с помощью следующего кода:

```
elif username == "S.Meier" and password == "civilization":
```

Составное условие в этой строке образовано из двух простых. Простые условия — это, конечно, username == "S.Meier" и password == "civilization". Они подобны тем условиям, которые вам уже приходилось видеть; отличие в том, что между ними указан логический оператор and. Полученное составное условие вдвое длиннее, но оно отнюдь не перестает быть условием — истинным либо ложным суждением.

Когда же истинно условие username == "S.Meier" and password == "civilization", а когда ложно? В английском языке союз and может предполагать, в частности, что верны оба высказывания, связанные им. Так и здесь: условие истинно тогда и только тогда, когда истинны оба составляющих его условия username == "S.Meier" и password == "civilization". В противном случае наше составное условие ложно. Вот табличное представление:

```
username == "S.Meier" password == "civilization" username == "S.Meier" and password == "civilization"
```

True	True	True
True	False	False
False	True	False
False	False	False

ПОДСКАЗКА

Пользуйтесь логическим оператором and как связкой между двумя условиями, когда хотите сформировать из них составное условие, истинное лишь в том случае, если обе его части истинны.

Итак, если Сид вводит как логин S.Meier, а как пароль — civilization, то составное условие оказывается истинным. Тогда система приветствует Сида и вводит его на определенный уровень защиты. Аналогичная процедура выполняется и для всех остальных зарегистрированных пользователей. В конструкции if–elif–else перебираются четыре различные пары «логин — пароль». Если пользователь ввел пару, известную системе, то компьютер приветствует его лично и обеспечит защиту заранее предусмотренного уровня. Если же пользователь или гость не сумеет войти в систему надлежащим образом, то компьютер скажет ему, что попытка не удалась, и обидно пошутит.

Логический оператор or

Гости тоже допускаются в сеть, хотя и с низкой степенью защиты. Чтобы гостю ничто не мешало опробовать функциональность сети, ему достаточно ввести guest в качестве логина или пароля. Вот строки кода, отвечающие за вход гостя в систему:

```
elif username == "guest" or password == "guest":
    print("Добро пожаловать к нам в гости.")
    security = 1
```

Проверяемое в этой конструкции `elif` условие `username == "guest" or password == "guest"` похоже на все предыдущие, которые относятся к зарегистрированным пользователям. Но большая разница состоит в том, что используется логический оператор `or`.

Составное условие с `or` истинно в том случае, если истинно по крайней мере одно из простых условий, входящих в его состав. Логический оператор `or` также схож по значению с соответствующим английским словом. Союз `or` означает «любой из»; если истинно любое из простых условий, то истинно и составное. В рассматриваемом случае истинность `username == "guest"`, или `password == "guest"`, или обоих этих условий делает условие `username == "guest" or password == "guest"` истинным. В противном случае оно не выполняется. Вот табличное представление:

<code>username == "guest"</code>	<code>password == "guest"</code>	<code>username == "guest" or password == "guest"</code>
<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>

Планирование программ

До сего момента все наши программы были крайне простыми. Схематически «планировать» такую программу на листке бумаги — то же самое, что стрелять из пушки по воробьям. Но идея планирования даже небольших программ, в сущности, хороша, потому что позволяет сэкономить время (а зачастую и нервы) в процессе разработки.

В этой главе я уже сравнивал программирование со строительством. Вообразите-ка себе подрядчика, который берется построить вам дом без единого чертежа! Может получиться монстр без окон, с 12 ванными комнатами и входной дверью на втором этаже. Кроме того, затраты на строительство в добрых десять раз превысят смету.

То же самое происходит и в программировании. Не имея плана, вы будете тратить много времени на борьбу с то и дело возникающими трудностями; может даже случиться так, что вы в конце концов не заставите программу работать. Вот почему так важно планирование. Ему посвящена особая область разработки ПО. Извлечь для себя пользу из некоторых простых приемов планирования работы может и начинающий программист.

Алгоритмы на псевдокоде

Алгоритмом называется последовательность ясных, легко выполнимых инструкций, направленных на достижение какой-либо цели. Алгоритм — это как бы краткое содержание будущей программы, некая квинтэссенция, которая придумана до начала работы над кодом и руководит процессом разработки.

Алгоритм состоит не только из цели. Это список шагов, которые надо пройти по порядку. Так, к примеру, «Стать миллионером» — это не алгоритм в собственном смысле, а лишь цель, впрочем, достойная цель.

Я написал «Алгоритм зарабатывания миллиона долларов». Вот он:

если вы способны придумать новый полезный товар

выпустите его в свет

иначе

выпустите существующий товар под своей маркой

создайте рекламный ролик своего товара

покажите этот ролик по телевидению

назначьте цену \$100 за единицу товара

продажте 10000 единиц товара

Вот что такое алгоритм: выписанная в явном виде последовательность конечных шагов, следуя которым можно достичь поставленной цели.

Обычно алгоритмы пишутся на так называемом *псевдокоде*, и мой алгоритм не исключение. Псевдокод — некая промежуточная форма между естественным языком (чаще английским) и языком программирования. Каждый, кто понимает обычный язык, поймет и мой алгоритм. В то же время алгоритм отчасти схож и с Python: первые четыре строки напоминают конструкцию *if* с условием и *else*, что вовсе не случайно.

Пошаговая доработка алгоритма

Как любой план или схема, ваш алгоритм, скорее всего, не ограничится первым наброском. Часто требуется добавить несколько «приближений», прежде чем воплотить алгоритм в коде. Процесс дописывания алгоритмов, в результате которого они становятся программно реализуемыми, называют *пошаговой доработкой*. Дорабатывать — это прежде всего делать более развернутым. Когда каждый шаг алгоритма разбивается на последовательность еще более простых шагов, от этого алгоритм становится ближе к коду. При пошаговой доработке элементы алгоритма дробятся на части до тех пор, пока вы не почувствуете, что весь алгоритм можно с легкостью «транслировать» в программу. Как пример рассмотрим один из шагов «Алгоритма зарабатывания миллиона долларов»:

создайте рекламный ролик своего товара

Это чересчур расплывчатая задача. Как именно создать рекламу? Ответить на этот вопрос поможет пошаговая доработка, путем которой мы разобьем один этап на несколько более мелких. Получится следующее:

напишите сценарий рекламы своего товара

арендуйте телестудию на один день

найдите съемочную группу

обеспечьте присутствие благосклонных зрителей

отснимите ролик

Если перечисленные пять шагов кажутся вам ясно сформулированными и выполнимыми, то доработка этой части алгоритма прошла успешно. Если же вам

что-нибудь непонятно, стоит глубже доработать проблематичный шаг алгоритма. Действуя таким образом, вы в конце концов получите подробный алгоритм, а вслед за ним и миллион долларов.

Вернемся к игре «Отгадай число»

Игра «Отгадай число» совмещает в себе многие из тех новых понятий, которые вы освоили при чтении этой главы. Это, по сути, первая несложная игра на Python, с которой вы можете познакомить своих родственников, друзей и/или подруг.

План программы

Задумав игру, я написал немного псевдокода:

```
выбрать случайное число  
до тех пор пока игрок не отгадает число  
    предоставить игроку возможность отгадывать  
    поздравить игрока
```

Как первое приближение — неплохо, но здесь недостает нескольких важных элементов. Во-первых, программа должна сообщать игроку, как соотносится предложенный им вариант с загаданным числом: оно больше или меньше. Во-вторых, система должна следить за тем, сколько попыток отгадывания совершил игрок, и сообщать ему в конце игры окончательное число.

ПОДСКАЗКА

Если первый составленный вами план программы неполон — ничего страшного. Всегда начинайте планирование с **ключевых** идей, а потом добавляйте недостающие звенья до тех пор, пока план не окажется достаточно подробным.

Вот доработанный вид моего алгоритма:

```
поприветствовать игрока и объяснить ему правила игры  
    выбрать случайное число в диапазоне от 1 до 100  
    предложить игроку отгадать число  
    приравнять порядковый номер попытки к 1  
    если число, предложенное игроком, не совпадает с загаданным  
        если оно больше загаданного  
            предложить игроку выбрать число поменьше  
        иначе  
            предложить игроку выбрать число побольше  
            вновь предложить игроку отгадать число  
            увеличить порядковый номер попытки на 1  
    поздравить игрока с победой  
    сообщить игроку, сколько попыток ему понадобилось на отгадывание числа
```

Теперь я готов кодировать. Просмотрев следующие несколько разделов, вы увидите, насколько бесхитростно псевдокод транслируется в выражения Python. Код программы можно найти на сайте-помощнике в папке Chapter 3; файл называется `guess_my_number.py`.

Начальный блок комментариев

Как и все хорошие программы, наша игра начинается с блока комментариев:

```
# Отгадай число
#
# Компьютер выбирает случайное число в диапазоне от 1 до 100
# Игрок пытается отгадать это число, и компьютер говорит,
# предположение больше/меньше, чем загаданное число.
# или попало в точку
```

Импорт модуля `random`

Вся соль программы в генерации случайных чисел. Для этого импортируется модуль `random`:

```
import random
```

Объяснение правил

Игра очень проста, но не помешает все же кратко объяснить пользователю ее правила:

```
print("\tДобро пожаловать в игру 'Отгадай число'!")
print("\nЯ загадал натуральное число из диапазона от 1 до 100.")
print("Постарайтесь отгадать его за минимальное число попыток.\n")
```

Установка начальных значений

Следующим шагом присвоим всем переменным их начальные значения:

```
# начальные значения
the_number = random.randint(1, 100)
guess = int(input("Ваше предположение: "))
tries = 1
```

Переменная `the_number` содержит число, которое игрок должен отгадать. Ей присвоено случайное целочисленное значение из интервала 1.. 100, порожденное функцией `random.randint()`. Вслед за тем функция `input()` получает из пользовательского ввода первое предположение игрока, а `int()` превращает введенную строку в число. Это число становится значением переменной `guess`. Наконец, переменной `tries`, которая содержит число уже сделанных попыток, я присвоил здесь значение 1.

Цикл отгадывания

Это основная часть программы. Цикл будет исполняться до тех пор, пока игрок не введет число, загаданное компьютером. В теле цикла предположение игрока (`guess`) сравнивается с действительно загаданным числом (`the_number`). Если первое больше, то на экране появится слово Меньше, а если наоборот — слово Больше. Затем из пользовательского ввода принимается очередное предположение игрока и количество попыток увеличивается на единицу.

```
# цикл отгадывания
while guess != the_number:
    if guess > the_number:
        print("Меньше...")
    else:
        print("Больше...")
    guess = int(input("Ваше предположение: "))
    tries += 1
```

Поздравления победителю

Когда игрок наконец отгадает число, `guess` станет равным `the_number`, то есть условие `guess != the_number`, организующее цикл, перестанет выполняться, и цикл прекратится. Теперь осталось только поздравить игрока:

```
print("Вам удалось отгадать число! Это в самом деле", the_number)
print("Вы затратили на отгадывание всего лишь ", tries, " попыток!\n")
```

Здесь компьютер сообщает игроку, какое число загадано и с которой по счету попытки игрок его назвал.

Ожидание выхода

Как обычно, последняя строка кода предлагает пользователю нажать `Enter`:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

Из этой главы вы узнали, как управлять исполнением программы. Вы усвоили, что нелинейное выполнение кода становится возможным за счет проверки условий, и приобрели навыки создания простых и составных условий. Вы научились пользоваться конструкцией `if` (с ее вариантами `if-else` и `if-elif-else`), благодаря которой возможно ветвление программы. Вы познакомились с циклами `while` — полезным инструментом повторения отдельных блоков кода. Вы получили представление о том, как важно планировать работу над программой, и узнали, что план программы может иметь вид алгоритма на псевдокоде. Кроме того, вы научились генерировать случайные числа, способные добавить в ваши программы элемент непредсказуемости.

- Напишите программу — симулятор пирожка с «сюрпризом», — которая бы при запуске отображала один из пяти различных «сюрпризов», выбранный случайным образом.
- Напишите программу, которая бы «подбрасывала» условную монету 100 раз и сообщала, сколько раз выпал орел, а сколько — решка.
- Измените программу «Отгадай число» таким образом, чтобы у игрока было ограниченное количество попыток. Если игрок не укладывается в заданное число (и проигрывает), то программа должна выводить сколь возможно суровый текст.
- А вот задача посложнее. Напишите на псевдокоде алгоритм игры, в которой случайное число от 1 до 100 загадывает человек, а отгадывает компьютер. Прежде чем приступить к решению, задумайтесь над тем, какой должна быть оптимальная стратегия отгадывания. Если алгоритм на псевдокоде будет удачным, попробуйте реализовать игру на Python.

4

Циклы с оператором `for`, строки и кортежи. Игра «Анаграммы»

Итак, вы теперь знаете, насколько удобны переменные для доступа к данным. Но по мере того как ваши программы будут удлиняться и усложняться, в них вырастет и количество переменных, следить за которыми может быть очень трудоемко. Вот почему вам пора познакомиться с понятием последовательности и новым типом данных — *кортежем*, который позволяет организовать информацию в виде упорядоченной группы единиц и работать с содержимым этой группы. Предстоит выяснить, что уже известный вам строковый тип — частный случай последовательности. Вы узнаете и об одной разновидности циклов, которая специально предназначена для работы с последовательностями.

Вот конкретные умения, которые вам предстоит приобрести:

- создавать циклы `for` для перебора элементов последовательности;
- с помощью функции `range()` создавать последовательность чисел;
- работать со строками как с последовательностями;
- использовать мощную функциональность кортежей;
- применять функции и операторы работы с последовательностями;
- индексировать последовательности и создавать срезы.

Знакомство с игрой «Анаграммы»

В игре «Анаграммы», окно которой показано на рис. 4.1, применяются многие из тех новых понятий, которые вы освоите при чтении этой главы.

Эта игра — воссоздание типичной головоломки на перестановку букв, одной из тех, какие в прошлом веке можно было встретить в воскресных газетах (тех самых, которые публика читала, пока не появился Интернет). Компьютер случайным образом выбирает из группы слов одно, переставляет его буквы тоже в случайном порядке и предъявляет игроку. Задача человека — восстановить исходное слово.

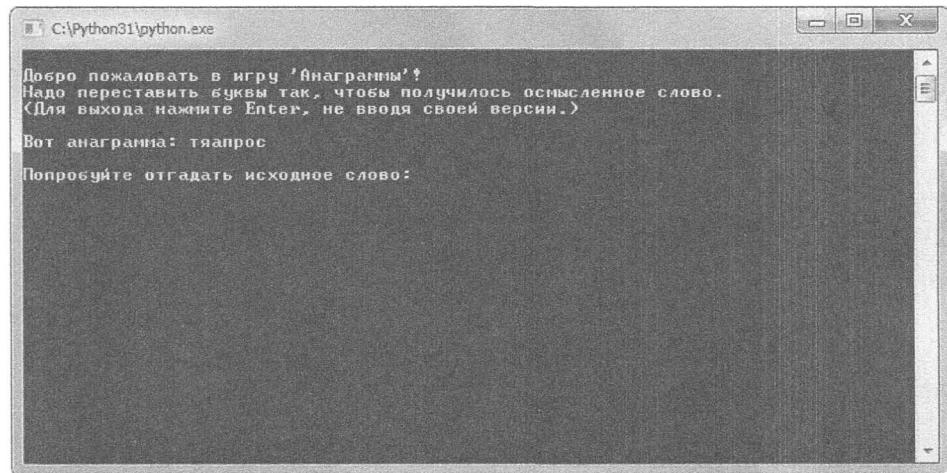


Рис. 4.1. Игра «Анаграммы». Непросто будет переставить буквы так, чтобы получилось слово

Применение циклов for

В предыдущей главе вы познакомились с одной разновидностью циклов — с циклами `while`. В них отдельный фрагмент кода исполняется повторно до тех пор, пока истинно какое-либо условие. В циклах с оператором `for` тоже циклизовано исполнение кода, но не на основе условия. В фундаменте цикла `for` лежит последовательность — упорядоченное множество объектов. Каждый раз, когда вам приходилось составлять, например, список дел на ближайшие дни, вы создавали последовательность.

Цикл `for` перебирает все *элементы* последовательности по порядку и для каждого из них исполняет фрагмент кода, заключенный в теле цикла. После достижения конца последовательности цикл завершается. Опять возьмем для примера ваш список дел. Можно написать цикл `for`, который по одному брал бы пункты этого списка и выводил их на экран. Чтобы лучше все это понять, обратимся к практике.

Знакомство с программой «Слово по буквам»

Эта программа принимает из пользовательского ввода слово и выводит его буквы по порядку на отдельных строках. Обратите внимание на рис. 4.2.

Эта несложная программа хорошо иллюстрирует суть цикла `for`. Ее код вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `loopy_string.py`.

```
# Слово по буквам
# Демонстрирует применение цикла for к строке
word = input("Введите слово: ")
print("\nВот все буквы вашего слова по порядку:")
for letter in word:
    print(letter)
input("\n\nНажмите Enter, чтобы выйти.")
```

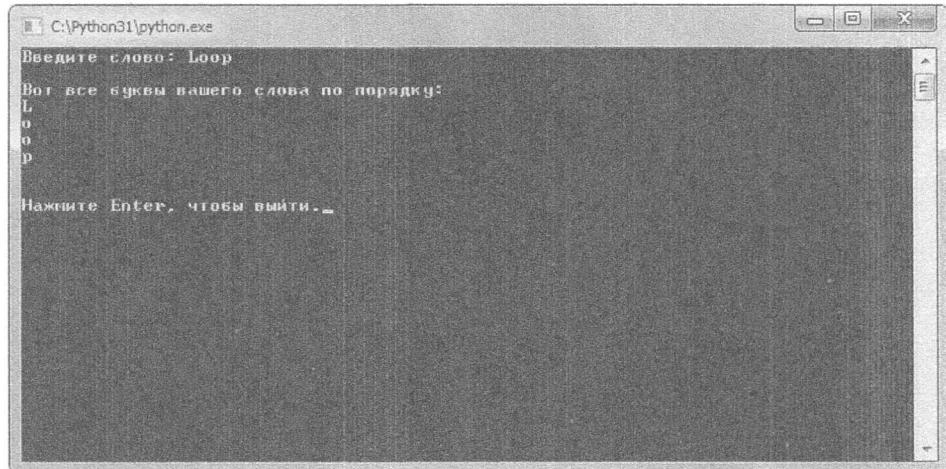


Рис. 4.2. Цикл с оператором for последовательно, по одной перебирает буквы слова, которое ввел пользователь

Разбираемся в работе цикла for

Новый для вас код — это цикл с оператором `for`, который состоит всего из двух коротких строк кода:

```
for letter in word:  
    print(letter)
```

Даже человеку, который совершенно не знаком с циклами `for`, этот код должен быть вполне понятен. Но я все же объясню точный механизм его работы.

Любая последовательность состоит из элементов. Стока — это последовательность, элементами которой являются отдельные символы. В окне программы я ввел строку "Цикл". Ее первый элемент как последовательности — символ "Ц", второй — "и" и т. д.

Цикл с оператором `for` перебирает элементы последовательности по одному. В данном случае итерациям подвергаются буквы в строке "Цикл". В цикле `for` используется переменная, значением которой становится каждый очередной элемент последовательности. Такова в моем цикле переменная `letter`; в нее попадают буквы слова "Цикл" по порядку. В теле цикла можно производить манипуляции с элементами последовательности. Моя программа просто выводит буквы слова на экран.

Заметьте, что переменная, которую один за другим попадают элементы последовательности, ничем не отличается от прочих переменных. Перед началом цикла она не существовала, значит, она будет создана. Итак, когда начинается цикл, система создает переменную `letter` и сохраняет в ней первый символ строки, в данном случае "Ц". Затем команда `print` в теле цикла печатает букву `L` на экране. Тело цикла завершилось, интерпретатор возвращается к началу цикла и сохраняет в переменной `letter` очередную букву слова — "и". На экране отображается `o`, и все происходит сначала до тех пор, пока не закончится строка "Цикл".

НА САМОМ ДЕЛЕ

В большинстве современных языков программирования есть та или иная разновидность цикла for. Однако возможности таких циклов, как правило, более ограничены. Обычно используется переменная-счетчик с целочисленным значением. При каждой итерации цикла значение счетчика меняется на одну и ту же величину. В отличие от этой традиционной модели, в языке Python цикл for обладает большей гибкостью, потому что позволяет организовать перебор любой последовательности.

Создание цикла for

Чтобы создать цикл с оператором for, вы можете следовать образцу, предложенному в моей программе. Сначала for, потом имя переменной, в которой предполагается сохранять значения элементов, потом in, потом имя перебираемой последовательности, двоеточие и, наконец, тело цикла. Вот и все.

Счет с помощью цикла for

При написании программ вы можете столкнуться с необходимостью последовательно перебирать числа. Цикл for и стандартная функция range() предоставляют такую возможность и довольно много разнообразия в ее реализации.

Знакомство с программой «Считалка»

Программа «Считалка» не представляет собой ничего особенного. Она нужна только для того, чтобы показать вам, как пользоваться функцией range() в сочетании с циклом for для перебора чисел по возрастанию, по убыванию и даже с пропусками. Чтобы увидеть результаты работы программы, взгляните на рис. 4.3.

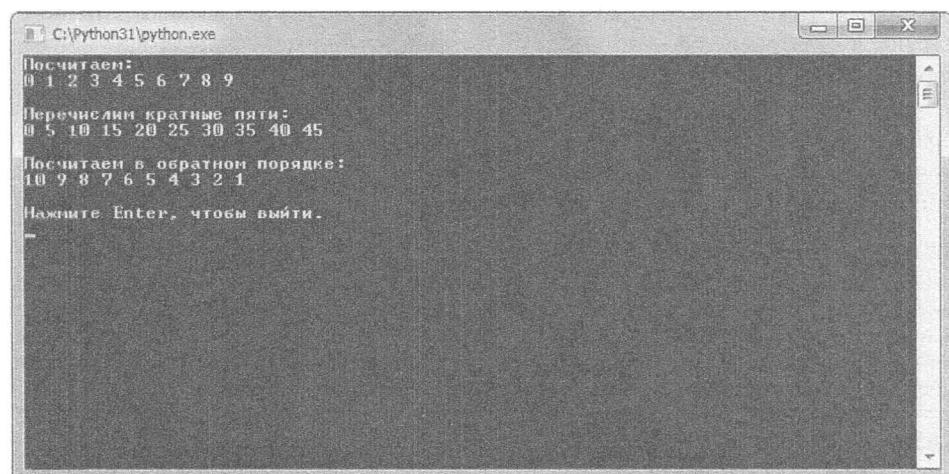


Рис. 4.3. Функция range() и цикл с оператором for позволяют перебирать числа вперед, назад и с пропусками

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется counter.py.

```
# Считалка
# Демонстрирует использование функции range()
print("Посчитаем:")
for i in range(10):
    print(i, end=" ")
print("\n\nПеречислим кратные пяти:")
for i in range(0, 50, 5):
    print(i, end=" ")
print("\n\nПосчитаем в обратном порядке:")
for i in range(10, 0, -1):
    print(i, end=" ")
input("\n\nНажмите Enter, чтобы выйти.\n")
```

НА САМОМ ДЕЛЕ

Обобщенные переменные-счетчики для перебора чисел в циклах принято обозначать *i*, *j*, *k*. Обычно имена переменных выбираются так, чтобы из них была ясна суть обозначаемых данных. Но можете мне поверить, любой опытный программист, просматривая ваш код, сразу поймет, что переменная *i*, *j* или *k* — счетчик цикла.

Счет по возрастанию

Первый цикл в этой программе организует перебор чисел по возрастанию:

```
for i in range(10):
    print(i, end=" ")
```

Этот цикл *for* работает подобно тому, который вы уже видели в программе «Слово по буквам». Но на сей раз последовательность, элементы которой перебирает цикл, не задана целиком. Значения, составляющие ее, по порядку возвращает функция *range()*. Удобно представить себе, что функция *range()* возвращает последовательность целых чисел. Если передать *range()* в качестве аргумента положительное число, то последовательность будет охватывать числа от 0 до переданного аргумента (не включая его). Поэтому, например, *range(10)* возвращает последовательность [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Так должно стать яснее, что происходит в цикле. При первой итерации счетчик *i* принимает значение 0, и оно выводится на экран; затем *i* становится равным 1, что тоже печатается на экране, и т. д. до последней итерации, в которой *i* становится равным 9, это значение выводится и цикл прекращает работу.

ЛОВУШКА

Хотя и полезно вообразить результат работы функции *range()* как последовательность целых чисел, в действительности это не совсем верно. На самом деле функция возвращает некий объект, который, в свою очередь, порождает каждое очередное число последовательности. Впрочем, для наших целей достаточно рассмотреть просто последовательность. Если вы хотите детальнее разобраться в работе функции *range()*, найдите документацию о ней на сайте <http://www.python.org>.

ПОДСКАЗКА

Собственные последовательности значений, называемые списками, вы тоже можете создавать. Для этого нужно перечислить значения через запятую и заключить всю конструкцию в квадратные скобки. Но не спешите пока создавать множество списков. Все, что надо знать о них, я сообщу вам в главе 5.

Счет по числам, кратным пяти

Следующий цикл организует перебор чисел, кратных пяти:

```
for i in range(0, 50, 5):
    print(i, end=" ")
```

Если передать функции `range()` три значения, она будет рассматривать их как начало, конец счета и интервал. Начало счета всегда выступает как первый элемент нашей воображаемой последовательности чисел, а конечное значение в нее не попадает. Таким образом, здесь получается как бы последовательность [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]. Надо еще раз отметить, что последовательность заканчивается числом 45, ведь 50 — конец счета (и поэтому не может быть включено). Если бы пришлось заканчивать перебор на 50-ти, можно было бы отодвинуть конечное значение немного вперед, например, так: `range(0, 51, 5)`.

Счет по убыванию

Последний цикл отвечает за перебор в обратном порядке:

```
for i in range(10, 0, -1):
    print(i, end=" ")
```

Здесь в вызове функции `range()` заключительный, третий аргумент равен `-1`. Как следствие, функция движется от начала к концу счета, все время прибавляя по `-1`, а ведь это то же самое, что вычитать 1. Воображаемая последовательность, которую на этот раз порождает `range()`, имеет вид [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Таким образом, счет ведется от 10 по убыванию вплоть до 1, но не включая 0.

ХИТРОСТЬ

Никто не заставляет обязательно использовать переменную цикла `for` внутри самого этого цикла. Если, например, вам угодно повторить какую-либо команду нужное количество раз, создайте цикл `for`, в теле которого не фигурировала бы переменная-счетчик. Допустим, я решил десять раз вывести на экран слово «Привет!». Чтобы сделать это, достаточно двух строк кода:

```
for _ in range(10):
    print("Привет!")
```

Операторы и функции для работы с последовательностями: применение к строкам

Как вы только что узнали, строки — один из типов последовательностей: их элементами являются отдельные символы. Для работы с последовательностями любого рода, в том числе и строками, в Python есть несколько полезных функций и операторов. Они дают доступ к элементарным, но от того не менее важным фактам, позволяя узнать, к примеру, какова длина последовательности, содержится ли в ней определенный элемент и др.

Знакомство с программой «Анализатор текста»

Наша следующая программа умеет анализировать текст, который вводит пользователь. Она выясняет, какова длина сообщения и содержится ли в нем самая частая согласная буква — «т». Этого позволяют добиться новые для вас функция и оператор работы с последовательностями.

Окно программы показано на рис. 4.4.

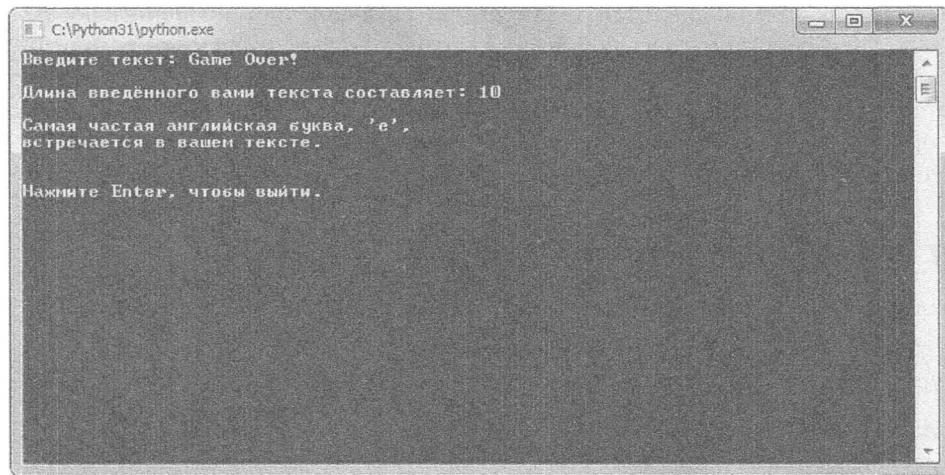


Рис. 4.4. Функция len() и оператор in помогут установить кое-какие факты о вашем тексте

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется message_analyzer.py.

```
# Анализатор текста
# Демонстрирует работу функции len() и оператора in
message = input("Введите текст: ")
print("\nДлина введенного вами текста составляет:", len(message))
print("\nСамая частая согласная. 'т'.")
if "т" in message:
    print("встречается в вашем тексте.")
else:
    print("не встречается в вашем тексте.")
input("\n\nНажмите Enter, чтобы выйти.")
```

Применение функции len()

После того как программа принимает пользовательский ввод, она отображает количество символов в нем. Это делает следующая строка кода:

```
print("\nДлина введенного вами текста составляет:", len(message))
```

Если передать функции len() какую-либо последовательность, то функция возвратит ее длину. Длиной последовательности называется количество элементов в ней.

Поскольку текст Вот он. текст!, который я ввел, состоит из десяти символов (символами мы считаем в том числе пробел и восклицательный знак), его длина равна 10.

Применение оператора `in`

Буква «т» считается самой частой согласной русского языка. Следующие строки кода в нашей программе позволяют выяснить, содержится ли эта буква в пользовательском вводе:

```
if "t" in message:  
    print("встречается в вашем тексте.")  
else:  
    print("не встречается в вашем тексте.")
```

Условие конструкции `if` выглядит так: "`t`" `in` `message`. Если строка, хранящаяся в переменной `message`, содержит символ "`t`", условие истинно, а если не содержит — ложно. В тексте Вот он. текст!, введенном при пробном запуске, имеется буква «т». Как следствие, условие "`t`" `in` `message` истинно и компьютер выводит на экран "...встречается в вашем тексте". Если бы условие оказалось ложно, например, при предъявлении компьютеру строки Игра окончена!, в которой нет буквы «т», то на экране появился бы текст "...не встречается в вашем тексте".

Содержащийся в последовательности элемент называют *членом* этой последовательности. Оператор `in` вы можете применять везде, где понадобится узнать, является ли членом последовательности какой-либо элемент. Достаточно написать имя переменной, содержащей этот элемент, затем `in` и имя последовательности — и получится условие. Если элемент — член последовательности, то условие будет истинно, если нет, то ложно.

Индексация строк

С помощью цикла `for` можно перебирать по одному символы строки. Это так называемый *последовательный доступ* к элементам. Последовательный доступ можно сравнить с извлечением очень тяжелого ящика из штабеля. Если, допустим, всего в штабеле пять ящиков, а вам нужно добраться до самого нижнего, то придется сначала снять верхний, потом второй сверху, третий, четвертый — и лишь тогда нужный ящик окажется в вашем распоряжении. Разве не удобнее расположить ящики так, чтобы извлечь один можно было, не заботясь о положении остальных? Такой вид доступа называется *произвольным* или *прямым доступом*. Он реализован для последовательностей и по отношению к ним называется *индексацией*. Достаточно указать порядковый номер элемента в последовательности (его индекс), чтобы извлечь его. Это как будто ящики лежат в ряд и достаточно выбрать пятый по счету из них.

Знакомство с программой «Случайные буквы»

В программе «Случайные буквы» индексация строки используется для непосредственного доступа к одному из ее символов, выбранному случайно. Программа

выбирает какую-либо порядковую позицию в строке "индекс", печатает номер позиции и соответствующую ей букву, а потом повторяет все это еще девять раз. Таким способом достигается хороший разброс случайных значений. Работу программы иллюстрирует рис. 4.5.

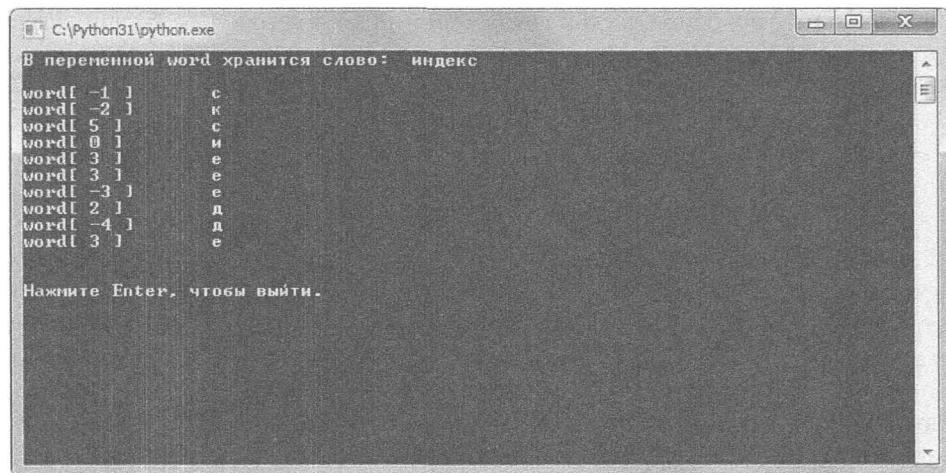


Рис. 4.5. Индексация позволяет непосредственно извлекать любой символ из строки

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `random_access.py`.

```
# Случайные буквы
# Демонстрирует индексацию строк
import random
word = "индекс"
print("В переменной word хранится слово: ", word, "\n")
high = len(word)
low = -len(word)
for i in range(10):
    position = random.randrange(low, high)
    print("word[", position, "]\t", word[position])
input("\n\nНажмите Enter, чтобы выйти.")
```

Позиции с положительными номерами

Чуть ли не первым делом в этой программе я приписываю переменной строковое значение:

```
word = "индекс"
```

В этом для вас нет ничего нового. Но следует иметь в виду, что тем самым создается последовательность (как и каждый раз, когда создается строка), а в ней позиция каждого символа описывается целым числом. Первая буква — «и» — стоит в позиции номер 0 (вообще в большинстве языков программирования индек-

сация начинается с 0). Вторая буква — «н» — стоит в позиции 1; третья — «д» — в позиции 2 и т. д.

Получить доступ к отдельному символу строки очень легко. Букву, которая находится в переменной `word` в позиции 0, обозначают попросту `word[0]`. Это верно и для любой другой позиции: достаточно заменить 0 соответствующим числом. Лучше понять и запомнить вам поможет следующий фрагмент интерактивной сессии:

```
>>> word = "индекс"
>>> print(word[0])
и
>>> print(word[1])
н
>>> print(word[2])
д
>>> print(word[3])
е
>>> print(word[4])
к
>>> print(word[5])
с
```

ЛОВУШКА

Поскольку строка «индекс» содержит шесть букв, вы могли бы подумать, что последняя буква — «с» — находится в позиции 6. Но это не так. Позиции 6 в данной строке вообще нет, ведь компьютер начинает отсчет с 0. Корректными номерами позиций являются целые числа от 0 до 5, а попытка доступа к позиции 6 вызовет ошибку. Это подтверждает интерактивная сессия:

```
>>> word = "индекс"
>>> print(word[6])
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print word[6]
IndexError: string index out of range
```

Тем самым компьютер не без злорадства заявляет, что позиции 6 в строке нет. Запомните: индекс заключительного элемента последовательности равен длине этой последовательности минус 1.

Позиции с отрицательными номерами

Если не считать того факта, что первый символ строки имеет индекс 0, в остальном неотрицательные индексы ведут себя вполне предсказуемо и просто. Однако есть и другой способ доступа к членам последовательности: позиции с отрицательными номерами. Если 0 — начало отсчета, то положительные индексы идут вправо от него. Так, в частности, происходит в строке, где начало отсчета — первый символ. Отрицательные индексы, наоборот, начинают отсчитываться от конца строки влево. Чтобы лучше понять, как это происходит, рассмотрим пример — еще одну интерактивную сессию со строкой «индекс»:

```
>>> word = "индекс"
>>> print(word[-1])
с
```

```
>>> print(word[-2])
к
>>> print(word[-3])
е
>>> print(word[-4])
д
>>> print(word[-5])
н
>>> print(word[-6])
и
```

Отсюда видно, что `word[-1]` — последняя буква слова «индекс», то есть «с». В системе отрицательных номеров позиций `-1` отсылает к последнему элементу, `-2` — ко второму с конца, `-3` — к третьему с конца и т. д. Иногда удобнее сделать началом отсчета индексов конец последовательности; для таких случаев и предназначены отрицательные номера позиций.

На рис. 4.6 показаны неотрицательные и отрицательные номера позиций, которые соответствуют шести буквам строки "индекс".

0	1	2	3	4	5
-6	-5	-4	-3	-2	-1
и	н	д	е	к	с

Рис. 4.6. Доступ к произвольной букве в слове «индекс» через номер позиции

Случайный элемент строки

Вернемся к программе «Случайные буквы». Чтобы выбрать из строки "индекс" какой-либо заранее не заданный элемент, нужно случайное число. Вот почему в первой же строке кода программы импортируется модуль `random`:

```
import random
```

Теперь надо придумать, как может быть случайно выбрана любая позиция с корректным номером, неотрицательным или отрицательным. Я решил сделать так: программа генерирует случайное число из диапазона от `-6` до `5` включительно (это все значения позиций, корректные для переменной `word`). Известно, что функция `random.randrange()` может принимать два числовых аргумента и возвращать случайное значение из интервала между ними. Пользуясь этим, я создал два граничных значения:

```
high = len(word)
low = -len(word)
```

В переменной `high` окажется число `6`, потому что слово «индекс» состоит из шести букв. В переменной `low` окажется то же значение со знаком «минус», то есть `-6`. Таковы края диапазона, из которого программа выберет случайное число.

Я предполагал, что это должно быть число не меньше -6 и не больше 5. Функция `random.randrange()` соответствует моим намерениям. Если передать ей два аргумента, она выберет случайное значение из диапазона от меньшего числа (включая его) до большего числа (не включая его). При пробном запуске, таким образом, команда:

```
position = random.randrange(low, high)
```

присвоит переменной `position` значение -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4 или 5. Этого мы и добивались, потому что именно таковы все номера позиций, которым соответствует хотя бы один из символов в строке "индекс".

Наконец, создадим цикл, который будет исполняться десять раз. В теле цикла выбирается случайная позиция, программа выводит на экран номер этой позиции и соответствующий ей символ:

```
for i in range(10):
    position = random.randrange(low, high)
    print("word[". position. "]\\t". word[position])
```

Что такое неизменяемость строк

Программисты говорят, что все последовательности делятся на две группы: изменяемые и неизменяемые. Если последовательность *изменяемая*, то ее элементы можно как-либо модифицировать, если *неизменяемая* — соответственно нельзя. Строки — неизменяемые последовательности. Их нельзя менять в том смысле, что, например, строка "Game Over!" всегда останется самой собой, преобразовать ее невозможно. Это относится к любой созданной вами строке. Ваш опыт работы со строками, вероятно, подсказывает, что я не прав. Более того, вы могли бы даже запустить интерактивную сессию и продемонстрировать что-нибудь такого рода:

```
>>> name = "Вася"
>>> print(name)
Вася
>>> name = "Петя"
>>> print(name)
Петя
```

В этом можно видеть доказательство изменяемости строк. В самом деле, строковое значение было равным "Вася", а стало равным "Петя". Но во время этой интерактивной сессии вы, собственно, и не пытались менять одну и ту же строку, а только создали две отдельные строки. Сначала вы создали строку "Вася" и сделали ее значением переменной `name`. Потом вы создали другую строку — "Петя", и она, в свою очередь, тоже стала значением `name`. И Вася и Петя — замечательные имена, но это разные имена и, значит, разные строки. То, что произошло во время интерактивной сессии, наглядно показано на рис. 4.7.

Удобно представлять себе строковые значения как записи чернилами на листках бумаги. Листок со строкой можно выбросить и заменить другим, содержащим новую строку, но изменить слова, которые уже однажды написаны, у вас не получится.

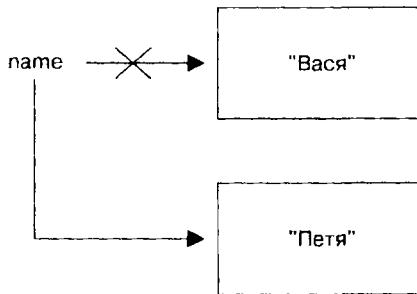


Рис. 4.7. Сначала name принимает строковое значение «Вася», а потом — строковое значение «Петя». Сами значения не меняются

Не стоит думать, будто я устраиваю много шума из ничего. У неизменяемости строк есть важные последствия. Так, в частности, нельзя присвоить новое значение элементу строки, доступному по его индексу. Чтобы понять, что я имею в виду, достаточно рассмотреть такую интерактивную сессию:

```

>>> word="игра"
>>> word[1]="к"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    word[1]="к"
TypeError: 'str' object does not support item assignment

```

Здесь, как видите, я решил превратить строку "игра" в строку "икра" (должно быть, в тот момент мне очень хотелось есть). Для этого требовалось всего лишь изменить «г» на «к». Поэтому я попробовал присвоить значение "к" второму элементу строки — word[1]. Это не увенчалось успехом: система выдает подробное сообщение об ошибке. Среди прочего интерпретатор пишет, что строковые объекты не поддерживают поэлементное присвоение (то есть отдельному символу в строке нельзя назначить новое значение взамен старого). Однако, хотя строки и невозможны менять, можно создавать новые строки из существующих.

Конструируем новую строку

Как соединить две и более строки с помощью оператора +, вы уже знаете. Иногда строить новую строку надо не из многосимвольных строк, а по одному символу. Поскольку строки неизменяемы, это будет, в сущности, та же самая конкатенация, которая при каждом добавлении символа создает новую строку.

Знакомство с программой «Только согласные»

Наша следующая программа под названием «Только согласные» принимает из пользовательского ввода текст и печатает его за вычетом всех гласных букв. Как работает эта программа, показано на рис. 4.8.

```
C:\Python31\python.exe
Введите текст: Он ненавидит гласные!
Создана новая строка: и
Создана новая строка: и
Создана новая строка: и н
Создана новая строка: и нн
Создана новая строка: и ннв
Создана новая строка: и ннвд
Создана новая строка: и ннвдт
Создана новая строка: и ннвдт г
Создана новая строка: и ннвдт гл
Создана новая строка: и ннвдт глс
Создана новая строка: и ннвдт глсн
Создана новая строка: и ннвдт глсн!

Вот ваш текст с изъятыми гласными буквами: н ннвдт глсн!

Нажмите Enter, чтобы выйти.
```

Рис. 4.8. Цикл `for` перебирает символы исходной строки, чтобы конструировать из них новые строки. Если символ — гласная буква, то программа его игнорирует

Исходная строка превращается в строку без гласных. Чтобы добиться этого, программа, по сути дела, последовательно создает сколько-то новых строк. Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `no_vowels.py`.

```
# Только согласные
# Демонстрирует, как создавать новые строки из исходных с помощью цикла for
message = input("Введите текст: ")
new_message = ""
VOWELS = "aeiouаеёиоуыэю"
print()
for letter in message:
    if letter.lower() not in VOWELS:
        new_message += letter
    print("Создана новая строка:", new_message)
print("\nВот ваш текст с изъятыми гласными буквами:", new_message)
input("\n\nНажмите Enter, чтобы выйти.")
```

Создание констант

После того как программа принимает пользовательский ввод и создает пустую переменную `new_message` для текста ответного сообщения, мы задаем строку:

```
VOWELS = "aeiouаеёиоуыэю"
```

Переменная `VOWELS` представляет собой строку всех гласных. Вы, должно быть, заметили, что имя переменной набрано прописными буквами вопреки известной вам договоренности, по которой имена переменных набирают в нижнем регистре. Впрочем, здесь я не отступил от традиции. Переменные со сплошь прописными буквами в именах обладают особым значением. Они называются *константами*

и ссылаются на значения, которые не предполагается менять (то есть на постоянные величины).

Константы ценные для программиста по двум причинам. Во-первых, они делают код понятнее. Так, в этой программе везде, где мне нужно обратиться к последовательности гласных букв, я пишу не "аеiouaeiouyзю", а VOWELS. Если вместо строки пользоваться именем переменной, ясность программы возрастет, ведь, видя переменную, вы вспоминаете, что она значит, а какая-нибудь странная строка самим своим видом может смутить. Во-вторых, константы экономят время набора и в какой-то мере уберегают от опечаток, свойственных повторному набору. Константы особенно полезны в тех случаях, когда вам приходится иметь дело с «длинными» значениями, например с очень большими числами и объемными строками. Если одно и то же неизменное значение встречается в коде в нескольких местах, это повод создать константу.

ЛОВУШКА

Будьте осторожны, создавая переменную, имя которой набрано прописными буквами. Несмотря на то что вы решились (и условились с другими программистами) ни в коем случае не менять ее значение, Python, в принципе, разрешает совершить этот варварский акт. Практика именования в данном случае сугубо условна. Поэтому, если уж вы создали переменную с именем из прописных букв, следите тщательно за тем, чтобы она оставалась константой.

НА САМОМ ДЕЛЕ

В некоторых языках программирования объявленные так константы уже не позволяют программисту менять их значения. Создавать и использовать константы этим способом безопаснее всего. В Python, однако, не существует столь прямолинейного способа создавать настоящие пользовательские константы.

Создание новых строк из существующих

Основная работа в этой программе происходит в теле цикла. По мере того как цикл перебирает символы, создается ответное сообщение без гласных букв. При каждой итерации компьютер рассматривает очередной символ введенной пользователем строки. Если это не гласная буква, то символ будет добавлен к строке-ответу. Если же это гласная, то программа перейдет к следующей букве. Как вы знаете, программа не может в собственном смысле «добавлять» символ к строке. Поэтому скажем точнее: сталкиваясь с символом, который не является гласной буквой, программа соединяет уже имеющуюся ответную строку с этим символом, так что получается новая строка. Вот код, на «плечах» которого все это лежит:

```
for letter in message:
    if letter.lower() not in VOWELS:
        new_message += letter
    print("Создана новая строка:", new_message)
```

Чтобы понять этот цикл, надо усвоить две новые мысли, которые я сейчас подробно изложу.

Во-первых, Python очень скрупулезен в работе со строками и символами. Интерпретатор считает, что "A" — не то же самое, что "a". Поскольку константе VOWELS присвоено строковое значение, состоящее сплошь из букв в нижнем регистре, сле-

дует убедиться, что оператор `in` сопоставляет с образцом только символы в нижнем регистре. Для этого и нужна запись `letter.lower()`.

ХИТРОСТЬ

Часто при сравнении двух строк неважно, совпадает ли регистр, существенно лишь, чтобы сами буквы были одинаковы. Если, например, спросить игрока, желает ли он(а) продолжить игру, ответы «Да» и «да» будут значить одно и то же. В таких случаях достаточно привести обе строки, образец и пользовательский ввод к какому-либо единому регистру (неважно, верхнему или нижнему), прежде чем сравнивать их.

Вот иллюстрация. Допустим, я хочу сопоставить два строковых значения: `name` и `winner`, чтобы увидеть, совпадают ли они; регистр неважен. Тогда я создам такое условие:

```
name.lower() == winner.lower()
```

Это условие истинно во всех случаях, когда `name` и `winner` содержат одну и ту же последовательность букв, хотя бы и набранных в разных регистрах. Так, строки "Вася" и "вася" с этой точки зрения совпадают. То же касается строк "ВАСЯ" и "вася", даже "ВаСя" и "вАСЯ".

Во-вторых, вы, скорее всего, заметили, что здесь к строкам применяется оператор `+=` — присвоения с увеличением. Вам приходилось видеть его работу с числами, но он может обслуживать и строки. Код:

```
new_message += letter
```

значит совершенно то же самое, что и:

```
new_message = new_message + letter
```

Срезы строк

Такой полезный механизм, как индексирование, дает возможность выбирать не обязательно ровно один элемент из последовательности. Можно также создавать копии непрерывных подпоследовательностей. Такие копии принято называть *срезами*. Срез может состоять из одного элемента, как при выборе по индексу, или из нескольких, например каких-нибудь трех из середины. Допустимо даже создать *срез-копию* всей исходной последовательности. Применительно к строкам это означает, что выбрать из строки можно любой символ и любое множество подряд расположенных символов — подстроку или целую строку.

Знакомство с программой «Резчик пиццы»

«Резчик пиццы» позволяет вам нарезать строку "пицца" любым способом. Это ни с чем не сравнимый способ в интерактивном режиме освоить технику создания срезов. От вас требуется лишь ввести начальную и конечную позиции, а программа позаботится о том, чтобы вывести соответствующий срез. Ее рабочее окно показано на рис. 4.9.

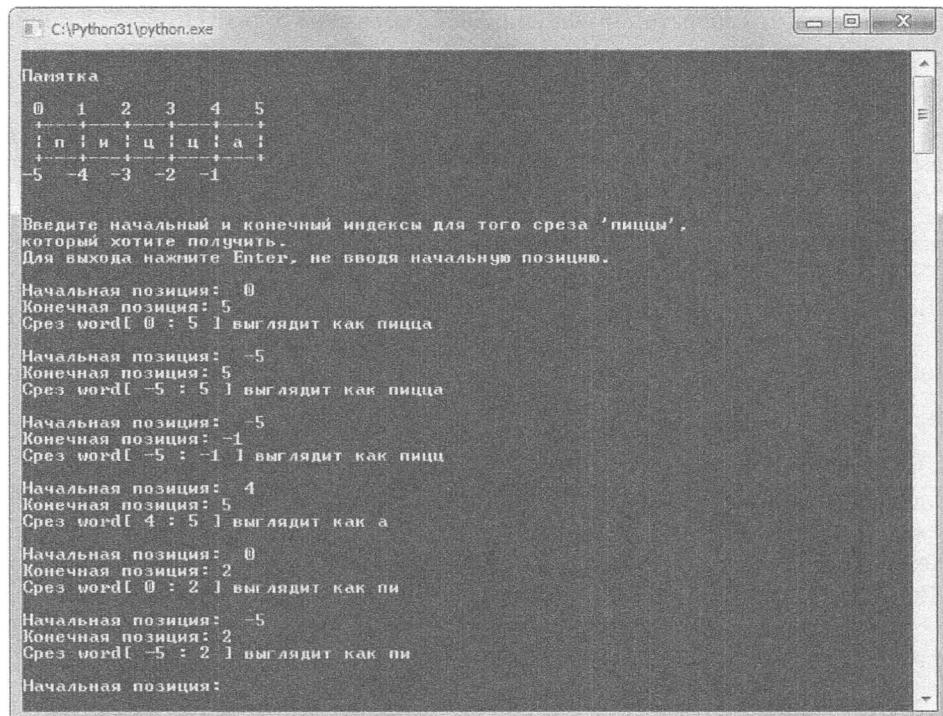


Рис. 4.9. Свежая горячая «пицца» будет нарезана в полном соответствии с вашим вкусом

В программе также содержится «памятка», или «шпаргалка», которая будет визуально напоминать вам, как выполняется срез.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `pizza_slicer.py`.

```
# Резчик пиццы
# Демонстрирует срезы строк
word = "пицца"
print(
"""
Памятка
0 1 2 3 4 5
+---+---+---+---+
| п | и | ц | ц | а |
+---+---+---+---+
-5 -4 -3 -2 -1
"""
)
print("Введите начальный и конечный индексы для того среза 'пиццы', который хотите получить.")
print("Для выхода нажмите Enter, не вводя начальную позицию.")
start = None
while start != "":
```

```

start = (input("\nНачальная позиция: "))
if start:
    start = int(start)
    finish = int(input("Конечная позиция: "))
    print("Срез word[", start, ":", finish, "] выглядит как", end=" ")
    print(word[start:finish])
input("\n\nНажмите Enter, чтобы выйти.")

```

Значение `None`

Прежде чем приступить к разбору той части кода, которая посвящена собственно срезам, обратим внимание на строку, в которой вводится новое ключевое слово:

```
start = None
```

Мы приписываем переменной `start` специальное значение `None`. С ее помощью в Python принято представлять пустое значение, заместитель еще не присвоенного. В условной интерпретации `None` рассматривается как ложное (`False`). Здесь я воспользовался этим значением, чтобы инициализировать `start` для указания в условии цикла `while`.

Разбираемся в срезах

Создание среза напоминает выбор элемента по индексу с той лишь разницей, что берется не один-единственный номер позиции элемента, а два числа: начальная и конечная позиции. В состав среза войдут все элементы, позиции которых заключены между этими двумя. На рис. 4.10 показано, как можно представить себе номера начальных и конечных позиций среза в строке "пицца". Заметьте, что эта нумерация несколько отличается от системы индексов, показанной на рис. 4.6.

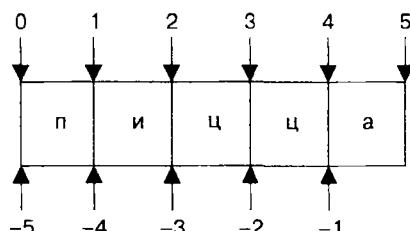


Рис. 4.10. Система границ срезов в строке «пицца». Любое сочетание неотрицательных и/или отрицательных номеров позиций дает корректный срез

Чтобы задать границы среза, надо записать номера начальной и конечной позиций в квадратных скобках через двоеточие. Понять, что я имею в виду, вам поможет такая интерактивная сессия:

```

>>> word = "пицца"
>>> print(word[0:5])
пицца
>>> print(word[1:3])
иц

```

```
>>> print(word[-4:-2])
иц
>>> print(word[-4:3])
иц
```

Срез `word[0:5]` возвращает целую строку, потому что все ее символы заключены между этими двумя ограничивающими номерами позиций. Срез `word[1:3]` возвращает строку "иц", так как только эти два символа лежат внутри границ. Как и при индексировании, можно пользоваться отрицательными номерами. Например, срез `word[-4:-2]` возвращает ту же самую строку "иц", что и `word[1:3]`, так как соответствующие отрицательные и положительные номера позиций задают одни и те же границы. Можно даже сочетать отрицательную нумерацию с положительной. Такой срез, как и любой другой, будет содержать все символы внутри пары границ. К примеру, `word[-4:3]` возвращает ту же самую строку "иц".

ЛОВУШКА

Если вызвать «невозможный» срез, в котором начальная позиция имеет больший номер, чем конечная, — наподобие `word[2:1]`, то интерпретатор Python не выведет ошибку, а просто возвратит пустую последовательность. По отношению к строкам это значит, что будет возвращена пустая строка. Будьте осторожны, ведь это, очевидно, не тот результат, которого вы ждете.

Создание срезов

Внутри цикла в программе «Резчик пиццы» следующая строка кода печатает на экране внешний вид записи, создающей срез с начальной и конечной позициями, которые ввел пользователь:

```
print("Срез word[". start. ":". finish. "] выглядит как", end=" ")
```

После этого программа выводит сам срез, вычисленный на основе значений переменных `start` и `finish`:

```
print(word[start:finish])
```

Сокращения в записи срезов

Все возможные срезы задаются какими-либо парами чисел. Но в некоторых случаях можно упрощать запись, пользуясь сокращениями. Так, разрешается опускать начальную позицию среза; тогда в ее качестве будет рассматриваться самый первый элемент последовательности. Так, например, если в переменной `word` хранится строка "пицца", то срез `word[:4]` будет значить то же самое, что `word[0:4]`. Кроме того, разрешается опускать конечную позицию, чтобы срез заканчивался вместе с последовательностью. В нашем примере `word[2:]` — сокращенная запись, по смыслу тождественная `word[2:5]`. Вы можете пропустить и оба числа, чтобы получить срез, совпадающий с целой последовательностью. В нашем примере `word[:]` — то же самое, что `word[0:5]`.

Вот интерактивная сессия в доказательство моих выкладок:

```
>>> word = "пицца"
>>> print(word[0:4])
пиц
```

```
>>>print(word[:4])
пицц
>>>print(word[2:5])
ца
>>>print(word[2:])
ца
>>>print(word[0:5])
пицца
>>> print(word[:])
пицца
```

ХИТРОСТЬ

Что вам надо обязательно запомнить из сокращений в записи срезов, так это `[:]` — способ получить точную копию последовательности. При программировании вам, возможно, время от времени придется создавать копии последовательностей. Срез от начала до конца — быстрый и эффективный способ добиться желаемого.

Создание кортежей

Кортежи, как и строки, представляют собой один из типов последовательностей. Но в отличие от строк, которые состоят только из символов, кортежи способны содержать элементы любой природы. В кортеже можно хранить, например, несколько рекордных результатов игроков или несколько фамилий сотрудников. При этом элементы кортежа не обязательно должны относиться все к одному и тому же типу. При желании ничто не мешает создать кортеж, который бы содержал и строковые, и числовые значения. Более того, строками и числами все не исчерпывается. Можно создать кортеж — последовательность картинок, звуковых файлов или даже инопланетных пришельцев (в одной из следующих глав рассказано, как это делается). Если значение можно хранить в переменной, то, значит, несколько таких значений можно сгруппировать и хранить в кортеже.

Знакомство с программой «Арсенал героя»

Программа «Арсенал героя» — попытка воссоздать набор боевых принадлежностей персонажа типичной RPG. Как и в большинстве таких игр, наш герой родился в захудалой деревушке. Его отца, разумеется, послал на смерть жестокий полководец (что же за сюжет будет у игры, если не смерть отца?). И вот теперь, когда герой повзросел, настало время мстить.

В нашей программе арсенал представлен в виде кортежа. Этот кортеж содержит строки — по одной на каждую боевую принадлежность, которой обладает герой. Сначала у героя нет ничего, но потом я наделяю его кое-какими припасами. Скромный первый шаг воинского пути показан на рис. 4.11.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `hero's_inventory.py`.

```
# Арсенал героя
# Демонстрирует создание кортежа
# создадим пустой кортеж
```

```

inventory = ()
# рассмотрим его как условие
if not inventory:
    print("Вы безоружны.")
input("\nНажмите Enter, чтобы продолжить.")
# теперь создадим кортеж с несколькими элементами
inventory = ("меч",
             "кольчуга",
             "щит",
             "целебное снадобье")
# выведем этот кортеж на экран
print("\nСодержимое кортежа:")
print(inventory)
# выведем все элементы последовательно
print("\nИтак, в вашем арсенале:")
for item in inventory:
    print(item)
input("\n\nНажмите Enter, чтобы выйти.")

```

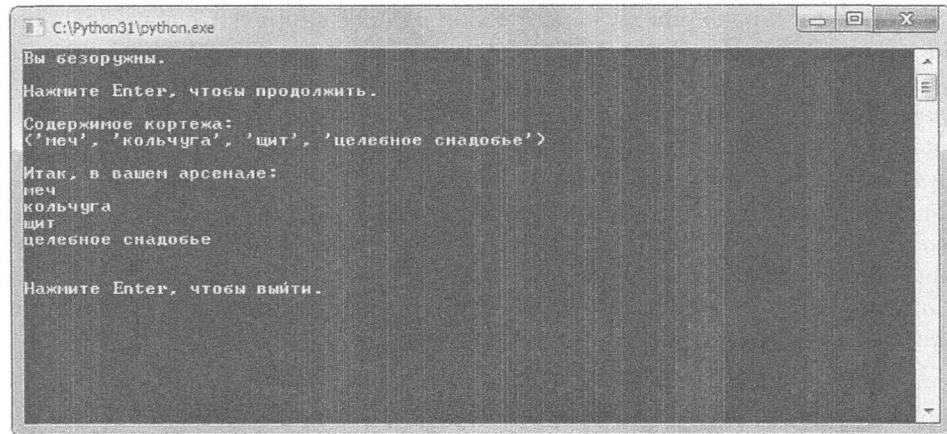


Рис. 4.11. Поначалу арсенал героя пуст, а потом, когда программа создает кортеж со строковыми элементами, герой вооружается

Создание пустого кортежа

Чтобы создать кортеж, достаточно заключить в круглые скобки последовательность значений, которые разделены запятыми. Даже просто пару скобок интерпретатор рассматривает как кортеж — правильно заданный, но пустой. В первой части программы я именно этим способом создал пустой кортеж, который показывает, что у героя ничего нет:

```
inventory = ()
```

Все очень просто: в этой строке кода переменной `inventory` присваивается значение пустого кортежа.

Кортеж как условие

Не так давно, читая об условиях, вы узнали, что в Python любое значение можно рассматривать как условие. Это относится и к кортежам. Условная конструкция реализована в следующих строках кода:

```
if not inventory:  
    print("Вы безоружны.")
```

Пустой кортеж интерпретируется как ложное условие (`False`), а кортеж, содержащий по крайней мере один элемент, — как истинное (`True`). Поскольку кортеж в переменной `inventory` пуст, он соответствует логическому значению `False`; соответственно, условие `not inventory` оказывается истинным. Как и задумано, компьютер выводит на экран строку "Вы безоружны.".

Создание непустого кортежа

С голыми руками наш герой далеко не уйдет. Вот почему я создал новый кортеж с элементами-строками, которые соответствуют разным полезным вещам в арсенале героя. Этот новый кортеж я вновь сделал значением переменной `inventory`:

```
inventory = ("меч",  
             "кольчуга",  
             "щит",  
             "целебное снадобье")
```

Элементы кортежа разделены запятыми. Иными словами, его первый элемент — строка "меч", второй — "кольчуга", третий — "щит", а заключительный, четвертый, — "целебное снадобье". В этом кортеже каждая строка — отдельный элемент.

Заметьте, кроме того, что запись кортежа охватывает несколько строк. Кортеж разрешено записывать как на одной строке, так и на нескольких (что я и сделал); во втором случае можно разрывать строку только после запятой. Это один из немногочисленных в Python видов команд, которые допускается располагать на нескольких строках.

ХИТРОСТЬ

Кортеж, разбитый на несколько строк, сделает ваш код более легким для чтения. Располагать ровно один элемент в строке кода, впрочем, не обязательно: можно поместить и несколько. Достаточно помнить, что каждая из неконечных строк должна завершаться разделителем — запятой, а на следующей строке нужно писать закрывающую скобку. Это спасает от просто элемента в скобках и от забывания дописывать запятую при добавлении элементов в конец. Соответственно, если в одноэлементном кортеже не поставить запятую после последнего и единственного элемента, это даже не будет кортеж — это будет какой-то элемент сам по себе в скобках.

Вывод элементов кортежа на экран

Несмотря на то что в кортеже может содержаться очень много элементов, для вывода их всех на экран, как и для печати на экране какого-либо элемента в отдельности, достаточно одной команды. Это и делается в следующем коде:

```
print("\nСодержимое кортежа:")
print(inventory)
```

Компьютер выводит на экран все элементы кортежа, заключив их в скобки.

Перебор элементов кортежа

В заключение я написал цикл for, который, перебирая элементы кортежа inventory, печатает их на экране по одному:

```
print("\nИтак, в вашем арсенале:")
for item in inventory:
    print(item)
```

Этот цикл выводит каждый содержащийся в inventory элемент на новую строку. Подобные циклы вам уже знакомы; в сущности, с их помощью можно организовывать перебор элементов любой последовательности.

Хотя в этом образце я создал кортеж, все элементы которого относятся к одному и тому же типу (строковому), вообще кортежи не обязательно заполнять однотипными значениями. Внутри кортежа отлично уживаются, например, строки, целые числа и десятичные дроби.

ЛОВУШКА

В других языках программирования есть структуры, подобные кортежам. Обычно они именуются массивами или векторами. В этих структурах зачастую действует ограничение, в силу которого все элементы должны быть одного типа. Может оказаться невозможным, например, хранить строковые и числовые значения в одном массиве. Имейте в виду, что подобные структуры, как правило, не столь гибки, как последовательности в Python.

Использование кортежей

Кортежи — один из типов последовательностей. Поэтому все, что вы уже знаете о последовательностях на примере строк, верно и по отношению к кортежам. Можно найти длину кортежа, вывести все его элементы на экран с помощью цикла for, а также, пользуясь оператором in, проверить кортеж на вхождение какого-либо элемента. Кортежи можно сцеплять, индексировать, создавать на них срезы.

Знакомство с программой «Арсенал героя 2.0»

Путь нашего героя продолжается. В этой программе его арсенал будет подсчитан и испытан; мы научимся извлекать отдельные принадлежности по их индексам и с помощью срезов. Герою посчастливится найти ларец, содержимое которого мы тоже представим кортежем. После соединения кортежей арсенал героя будет дополнен драгоценностями из ларца. Тестовый запуск программы показан на рис. 4.12.

Эта программа несколько длинновата, поэтому я продемонстрирую не весь ее код сразу, а по одному фрагменту на раздел. Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется hero's_inventory2.py.

```

C:\Python31\python.exe

Итак, в вашем арсенале:
меч
кольчуга
щит
целебное снадобье

Нажмите Enter, чтобы продолжить.
Сейчас в вашем распоряжении 4 предмета/-ов.

Нажмите Enter, чтобы продолжить.
Вы еще поживите и побоитесь.

Введите индекс одного из предметов арсенала: 1
Под индексом 1 в арсенале находится кольчуга

Введите начальный индекс среза: 2
Введите конечный индекс среза: 4
Срез inventory[2 : 4] — это ('щит', 'целебное снадобье')

Нажмите Enter, чтобы продолжить.
Вы нашли ларец. Вот что в нем есть:
('золото', 'драгоценные камни')
Вы приобрели содержимое ларца к своему арсеналу.
Теперь в вашем распоряжении:
('меч', 'кольчуга', 'щит', 'целебное снадобье', 'золото', 'драгоценные камни')

Нажмите Enter, чтобы выйти.

```

Рис. 4.12. Поскольку арсенал героя — кортеж, его элементы можно считать, индексировать, создавать срезы, а также объединять кортеж с другим кортежем

Настройка программы

Первая часть программы функционирует точно так же, как и в предшествующей версии «Арсенала героя». В следующем коде создается кортеж, все элементы которого выводятся на экран:

```

# Арсенал героя 2.0
# Демонстрирует работу с кортежами
# создадим кортеж с несколькими элементами и выведем его с помощью цикла for
inventory = ("меч",
             "кольчуга",
             "щит",
             "целебное снадобье")
print("\nИтак, в вашем арсенале:")
for item in inventory:
    print(item)
input("\nНажмите Enter, чтобы продолжить.")

```

Применение функции len() к кортежам

Функция `len()` работает с кортежами точно так же, как и со строками: чтобы узнать длину кортежа, поместите его имя внутрь скобок. Функция возвратит количество элементов в кортеже. Пустые кортежи, как и всякие пустые последовательности, имеют длину 0. В следующем коде функция `len()` применяется к кортежу:

```

# найдем длину кортежа
print("Сейчас в вашем распоряжении", len(inventory), "предмета/-ов.")
input("\nНажмите Enter, чтобы продолжить.")

```

Поскольку в этом кортеже четыре строки ("меч", "кольчуга", "щит" и "целебное снадобье"), на экране появится текст Сейчас в вашем распоряжении 4 предмета/-ов.

ЛОВУШКА

Заметьте, что строка «целебное снадобье» в кортеже `inventory` считается одним элементом, хотя и представляет собой два слова, разделенных пробелом.

Применение оператора in к кортежам

Является ли элемент членом кортежа (как и строки), вам поможет установить оператор `in`. Как вы уже знаете, он обычно используется в составе условий. Вот, например, какое условие создал я:

```
# проверка на принадлежность кортежу с помощью in
if "целебное снадобье" in inventory:
    print("Вы еще поживете и повоюете.")
```

Условие "целебное снадобье" `in inventory` проверяет, входит ли строка "целебное снадобье" как один элемент в состав `inventory`. Поскольку это действительно так, на экране отображается текст Вы еще поживете и повоюете.

Индексация кортежей

Индексация кортежей во всем подобна индексации строк. Чтобы получить доступ к какому-либо элементу, надо указать номер его позиции в квадратных скобках. В следующих строках кода пользователю предоставляется возможность выбрать номер позиции (индекс), а компьютер возвращает соответствующий предмет из арсенала:

```
# вывод одного элемента с определенным индексом
index = int(input("\nВведите индекс одного из предметов арсенала: "))
print("Под индексом", index, "в арсенале находится", inventory[index])
```

Наш кортеж с индексами показан на рис. 4.13.

0	1	2	3
-4	-3	-2	-1
"меч"	"кольчуга"	"щит"	"целебное снадобье"

Рис. 4.13. Каждая строка — отдельный элемент кортежа

Срезы кортежей

Этот механизм работает аналогично срезам строк. Нужны начальная и конечная позиции. Срез, который получается в итоге, — кортеж, содержащий все элементы между этими позициями.

Как и в программе «Резчик пиццы», рассмотренной выше в этой главе, здесь я предоставляю пользователю самостоятельно выбрать номера начальной и конечной позиций. Программа выводит на экран срез:

```
# отобразим срез
start = int(input("\nВведите начальный индекс среза: "))
finish = int(input("Введите конечный индекс среза: "))
print("Срез inventory[", start, ":", finish, "] - это", end=" ")
print(inventory[start:finish])
input("\nНажмите Enter, чтобы продолжить...")
```

На примере этого кортежа рис. 4.14 наглядно иллюстрирует, что представляют собой срезы кортежей.

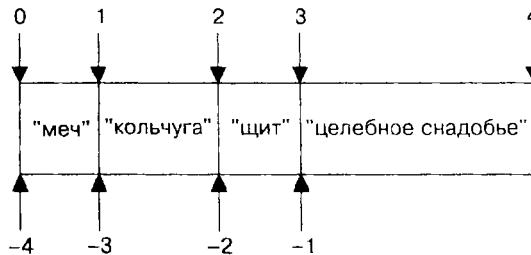


Рис. 4.14. В кортеже, как и в строке, позиции начала и конца среза расположены между элементами

Неизменяемость кортежей

Подобно строкам, кортежи неизменяемы. В доказательство приведу интерактивную сессию:

```
>>> inventory = ("меч", "кольчуга", "щит", "целебное снадобье")
>>> print(inventory)
('меч', 'кольчуга', 'щит', 'целебное снадобье')
>>> inventory[0] = "боевой топор"
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    inventory[0] = "боевой топор"
TypeError: object doesn't support item assignment
```

Хотя кортежи нельзя менять, из них можно создавать новые кортежи.

Сцепление кортежей

Кортежи соединяются тем же способом, что и строки. Достаточно поместить между именами двух кортежей оператор конкатенации +:

```
# соединим два кортежа
chest = ("золото", "драгоценные камни")
print("Вы нашли ларец. Вот что в нем есть:")
print(chest)
```

```
print("Вы приобщили содержимое ларца к своему арсеналу .")
inventory += chest
print("Теперь в вашем распоряжении:")
print(inventory)
input("\n\nНажмите Enter, чтобы выйти.")
```

Первое, что я решил сделать, — создать новый кортеж `chest`, содержащий два строковых элемента: "золото" и "драгоценные камни". Затем я вывел `chest` на экран, чтобы показать его элементы пользователю. После этого с помощью оператора `+=` я склеил кортежи `inventory` и `chest`, а результат этой операции — новый кортеж — присвоил уже существующей переменной `inventory`. Исходный кортеж, который ранее хранился в `inventory`, не был изменен, потому что его, как и любой кортеж, изменить невозможно. Вместо этого я создал совершенно новый кортеж, поместил в него элементы `inventory` и `chest` и сохранил его как `inventory`.

Вернемся к игре «Анаграммы»

Несколько новых понятий, которые вы усвоили из этой главы, применяются в игре «Анаграммы». Ее код можно изменять, создавая свой собственный список слов для отгадывания. Код программы находится на сайте-помощнике (courseptr.com/downloads) в папке Chapter 4. Файл называется `word_jumble.py`.

Настройка программы

После нескольких начальных комментариев загружается модуль `random`:

```
# Анаграммы (Word Jumble)
#
# Компьютер выбирает какое-либо слово и хаотически переставляет его буквы
# Задача игрока - восстановить исходное слово
import random
```

После этого я создаю кортеж, который будет содержать список слов. Заметьте, что название переменной `WORDS` набрано прописными буквами: далее в программе она рассматривается как константа.

```
# создадим последовательность слов, из которых компьютер будет выбирать
WORDS = ("питон", "анаграмма", "простая", "сложная", "ответ", "подстаканник")
```

Вслед за тем, чтобы выбрать из `WORDS` случайное слово, я применяю функцию `random.choice()`:

```
# случайным образом выберем из последовательности одно слово
word = random.choice(WORDS)
```

Это новая для вас функция, но по сути она очень проста. Интерпретатор берет какую-либо последовательность и извлекает из нее случайный элемент.

После того как слово выбрано, оно помещается в переменную `word`. Это и будет слово, которое игроку придется отгадать. Наконец, значение `word` копируется в переменную `correct`. С ней мы позже сравним версию игрока, чтобы увидеть, прав он или нет:

```
# создадим переменную, с которой будет затем сопоставлена версия игрока
correct = word
```

Как сформировать анаграмму

Следующий фрагмент кода — наиболее интересная часть программы как по замыслу, так и по технической реализации. Здесь мы будем переставлять буквы в исходном, случайно выбранном слове.

Прежде чем начать кодировать, я продумал эту часть программы и записал на псевдокоде (да, я пользуюсь псевдокодом на практике, а не только вам про него рассказываю). Вот в первом приближении тот алгоритм, который создает анаграмму из исходного слова:

```
создать пустую анаграмму
до тех пор пока исходное слово содержит хотя бы одну букву:
    изъять из слова случайную букву
    присоединить эту букву к анаграмме
```

По замыслу здесь все хорошо, но следует тщательнее поработать над семантикой. Поскольку строки неизменяемы, «изъять случайную букву» из слова, строго говоря, нельзя. Зато можно создать новую строку, в которой не будет этой буквы. Точно так же нельзя «присоединить букву» к строке-анаграмме, но можно создать новую строку, которая будет представлять собой конкатенацию анаграммы (в ее текущем виде) и «изъятой» буквы.

Создание пустой строки для анаграммы

Начало алгоритма реализуется очень просто:

```
# создадим анаграмму выбранного слова, в которой буквы будут расставлены хаотично
jumble = ""
```

Программа создает пустую строку и делает ее значением переменной `jumble`. Эта переменная в конце концов, после всех преобразований будет ссылаться на анаграмму.

Настройка цикла

За создание анаграммы отвечает цикл `while`. Его условие, как видите, весьма три-виально:

```
while word:
```

Я организовал цикл таким образом, чтобы исполнение тела цикла продолжалось до тех пор, пока переменная `word` не окажется приравненной к пустой строке. Все верно: при каждом следующем исполнении цикла компьютер создает очередную версию исходной строки — за вычетом одной из букв — и делает ее значением переменной `word`. В конце концов `word` окажется пустой строкой и цикл прервется.

Выбор случайной позиции в слове

Первая строка в теле цикла генерирует случайную позицию в слове, исходя из его длины:

```
position = random.randrange(len(word))
```

Буква `word[position]` и будет той буквой, которую мы «изымем» из строки `word` и «присоединим» к строке `jumble`.

Новая версия `jumble`

Следующая строка кода призвана создать обновленный вариант строки `jumble`. К прежней строке присоединяется буква `word[position]`.

```
jumble += word[position]
```

Новая версия `word`

Далее в коде цикла:

```
word = word[:position] + word[(position + 1):]
```

создан обновленный вариант строки `word`, в которой теперь уже не будет содержаться буква с индексом `position`. С помощью срезов компьютер извлекает из `word` две подстроки. Первый срез — `word[:position]` — это все буквы с начала слова до `word[position]`, не включая ее. Второй срез — `word[(position + 1):]` — это все буквы от `word[position]`, также не включая, и до конца слова. Конкатенацию этих двух строк мы делаем значением переменной `word`, которая теперь равна самой себе за вычетом одной буквы — `word[position]`.

Программа приветствует игрока

После того как буквы в слове будут переставлены, надо поприветствовать участника игры, объяснить ему правила и показать анаграмму, исходный вид которой он будет восстанавливать:

```
# начало игры
print(
    """
```

Добро пожаловать в игру 'Анаграммы'!

Надо переставить буквы так, чтобы получилось осмысленное слово.
(Для выхода нажмите Enter, не вводя своей версии.)

```
    )
print("Вот анаграмма:", jumble)
```

Получение пользовательского ввода

Теперь компьютер узнает ответ игрока. Программа будет просить ввести версию до тех пор, пока игрок не укажет правильное слово или не нажмет `Enter`:

```
guess = input("\nПопробуйте отгадать исходное слово: ")
while guess != correct and guess != "":
    print("К сожалению, вы неправы.")
    guess = input("Попробуйте отгадать исходное слово: ")
```

Поздравление с правильно отгаданным словом

К этому моменту в исполнении кода программы игрок или правильно отгадал слово, или уже вышел из игры. В первом случае компьютер поздравит его с успехом:

```
if guess == correct:
    print("Да, именно так! Вы отгадали!\n")
```

Конец игры

Программе осталось только поблагодарить игрока за участие и завершить работу:

```
print("Спасибо за игру.")
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

В этой главе вы познакомились с понятием последовательности. Вы узнали, как создавать последовательность чисел с помощью функции `range()`, и поняли, что строки представляют собой всего лишь последовательности символов. Кроме того, вы овладели техникой работы с кортежами, которые позволяют создавать последовательность элементов любого типа. Вы научились перебирать элементы последовательности с помощью цикла `for`, находить длину последовательности и проверять, является ли ее членом какой-либо элемент. Теперь вы умеете копировать части последовательности с помощью индексов и срезов. Вы осведомлены о том, что такое неизменяемость, какие ограничения она создает и как, несмотря на них, создавать новые последовательности из существующих с помощью конкатенации. В заключение вы, пользуясь всеми этими новыми знаниями, создали увлекательную игру «Анаграммы».

ЗАДАЧИ

- Напишите программу, которая бы считала по просьбе пользователя. Надо позволить пользователю ввести начало и конец счета, а также интервал между называемыми целыми числами.
- Напишите программу, которая принимала бы текст из пользовательского ввода и печатала этот текст на экране наоборот.
- Доработайте игру «Анаграммы» так, чтобы к каждому слову полагалась подсказка. Игрок должен получать право на подсказку в том случае, если у него нет никаких предположений. Разработайте систему начисления очков, по которой бы игроки, отгадавшие слово без подсказки, получали больше тех, кто запросил подсказку.
- Создайте игру, в которой компьютер выбирает какое-либо слово, а игрок должен его отгадать. Компьютер сообщает игроку, сколько букв в слове, и дает пять попыток узнать, есть ли какая-либо буква в слове, причем программа может отвечать только «Да» и «Нет». Вслед за тем игрок должен попробовать отгадать слово.

5 Списки и словари. Игра «Виселица»

Кортежи — хороший способ манипулировать элементами разных типов в составе одной последовательности. Но из-за того, что кортеж неизменяем, иногда возникают неудобства. К счастью, есть последовательности другого вида, так называемые *списки*, которые умеют все то же самое, что и кортежи, и даже больше — просто потому, что список изменяем. Его элементы можно удалять, а также добавлять новые. Можно даже подвергать список сортировке.

Я познакомлю вас еще с одной разновидностью последовательностей — *словарями*. Если список организован как набор значений, то словарь — как набор пар значений. Подобно своему тезке с книжной полки, словарь позволяет находить то значение, которое соответствует какому-либо другому.

Говоря более детально, в этой главе вы научитесь делать следующее:

- создавать списки, индексировать их и делать срезы;
- добавлять и удалять элементы списка;
- применять списочные методы, которые позволяют сортировать список и добавлять значение в конец;
- применять вложенные последовательности, которыми можно представить данные любой сложности;
- пользоваться словарями для работы с парами значений;
- добавлять и удалять элементы словаря.

Знакомство с игрой «Виселица»

В центре внимания этой главы — проект, посвященный игре «Виселица». Втайне от пользователя компьютер выбирает какое-либо слово, и игрок должен попробовать отгадать его, высказывая свои предположения побуквенно. Каждый раз, когда игрок ошибается, компьютер дорисовывает на экране изображение фигурки под виселицей. Если в отведенное количество попыток игроку не удается отгадать слово, то «повешенный» гибнет. На рис. 5.1–5.3 показан игровой процесс во всем его страшном великолепии.

Игра интересна не только сама по себе. Замечательно также и то, что к концу изучения этой главы вы узнаете, как создать собственную версию игры. Вы сможете, например, обзавестись своим личным списком «секретных» слов или заменить мои примитивные рисунки более внушительными.

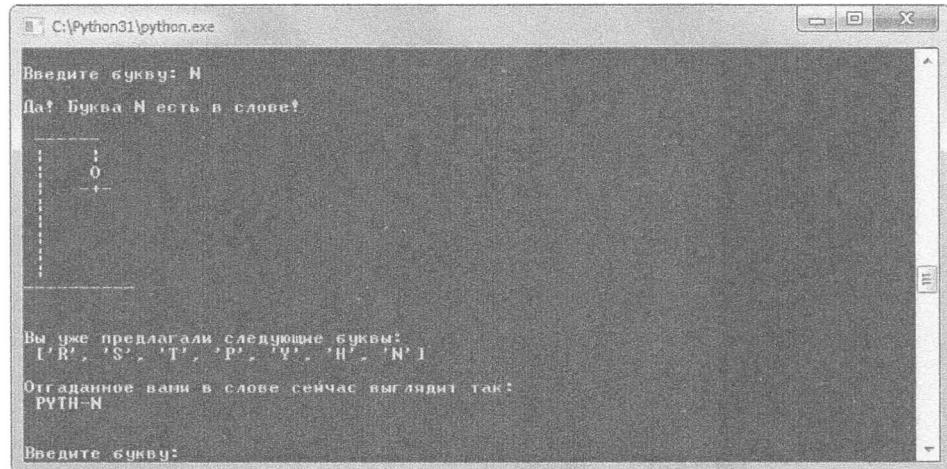


Рис. 5.1. Играем в «Виселицу». Гм... Что же за слово он загадал?

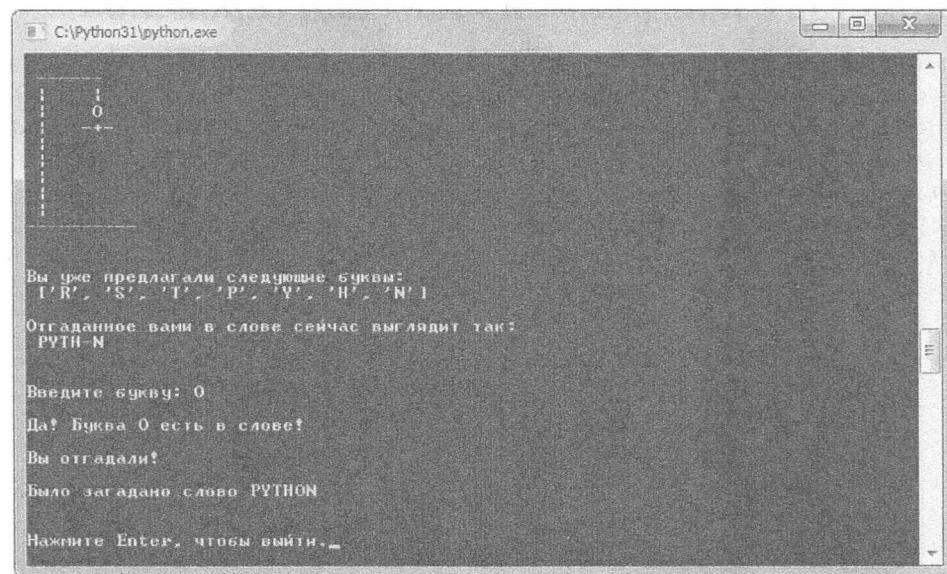


Рис. 5.2. В этой игре я победил!

Использование списков

Списки, как и кортежи, — это последовательности. Но списки изменяемы. Их элементы поддаются модификации. Вот почему списки обладают той же функциональностью, что и кортежи, и могут кое-что еще. То, что вы уже знаете о работе с кортежами, применимо и к спискам, а значит, научиться ими пользоваться для вас не составит труда.

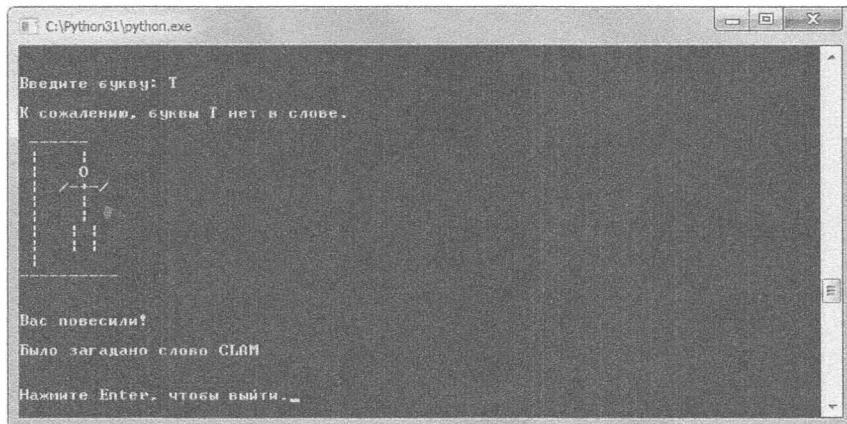


Рис. 5.3. А эта игра закончилась не лучшим образом, особенно для псевдографического человечка

Знакомство с программой «Арсенал героя 3.0»

Эта программа основана на более ранней игре «Арсенал героя 2.0», с которой мы познакомились в главе 4. Здесь для хранения данных об арсенале героя использованы не кортежи, а списки. Начальная часть «Арсенала героя 3.0» выдает такие же результаты, как и более ранняя версия. Да и код в ней практически тот же самый; разница лишь в том, что вместо кортежей везде использованы списки. На рис. 5.4 показано окно программы после выполнения этой первой части. Следующий за ней код демонстрирует полезные следствия изменяемости списков и некоторые новые приемы работы с последовательностями. Выполнение этой части программы отражено на рис. 5.5.

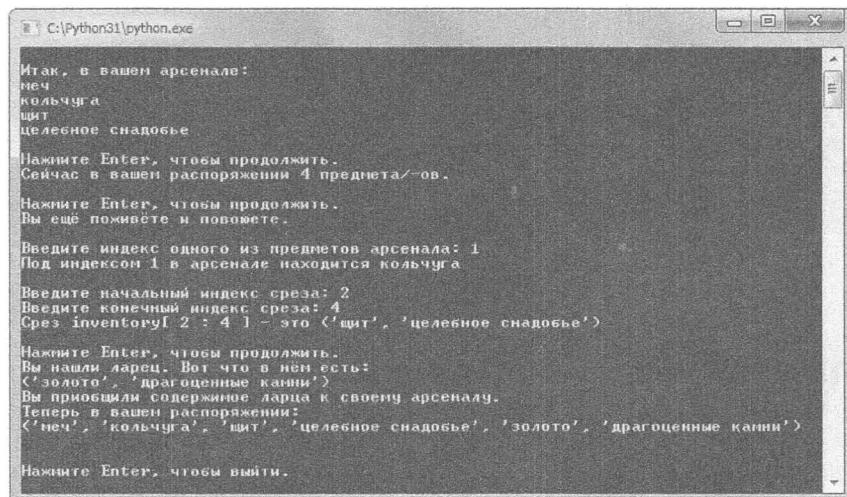


Рис. 5.4. Теперь арсенал героя представлен списком. В окне программы все выглядит почти так же, как выглядело в версии 2.0, где использовался кортеж

```
C:\Python31\python.exe

Нажмите Enter, чтобы продолжить.
Вы обненили меч на арбалет.
Теперь ваш арсенал содержит следующие предметы:
['арбалет', 'кольчуга', 'щит', 'целебное снадобье', 'золото', 'драгоценные
камни']

Нажмите Enter, чтобы продолжить.
За золото и драгоценные камни вы купили магический кристалл, способный
предсказывать будущее.
Теперь в вашем распоряжении:
['арбалет', 'кольчуга', 'щит', 'целебное снадобье', 'магический кристалл']

Нажмите Enter, чтобы продолжить.
В тяжёлом бом был раздроблен ваш щит.
Вот что осталось в арсенале:
['арбалет', 'кольчуга', 'целебное снадобье', 'магический кристалл']

Нажмите Enter, чтобы продолжить.
Воры лишили вас арбалета и кольчуги.
В арсенале теперь только:
['целебное снадобье', 'магический кристалл']

Нажмите Enter, чтобы выйти.
```

Рис. 5.5. Поскольку арсенал героя — это список, то его элементы можно добавлять, изменять и удалять

Создание списка

В первых строках кода создается новый список, который становится значением переменной `inventory`; система выводит элементы списка на экран. Единственное отличие от «Арсенала героя 2.0» состоит в том, что я заключил элементы списка в квадратные, а не в круглые, скобки и получил список, а не кортеж. Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 5. Файл называется `hero's_inventory3.py`.

```
# Арсенал героя 3.0
# Демонстрирует работу со списками
# создадим список с несколькими элементами и выведем его с помощью цикла for
inventory = ["меч", "кольчуга", "щит", "целебное снадобье"]
print("\nИтак, в вашем арсенале:")
for item in inventory:
    print(item)
input("\nНажмите Enter, чтобы продолжить.")
```

Применение функции `len()` к спискам

Следующий код побуквенно совпадает с аналогичным кодом в «Арсенале героя 2.0». Функция `len()` работает со списками так же, как и с кортежами.

```
# найдем длину списка
print("Сейчас в вашем распоряжении", len(inventory), "предмета/-ов.")
input("\nНажмите Enter, чтобы продолжить.")
```

Применение оператора `in` к спискам

Этот код тоже без изменений позаимствован из предыдущей версии. Оператор `in` не видит разницы между кортежем и списком.

```
# проверка на принадлежность списку с помощью in
if "целебное снадобье" in inventory:
    print("Вы еще поживете и побоюетесь.")
```

Индексация списков

Опять ничего нового по сравнению с версией 2.0. Для доступа к элементу списка, как прежде, надо заключить в квадратные скобки номер позиции, в которой находится элемент.

```
# вывод одного элемента с определенным индексом
index = int(input("\nВведите индекс одного из предметов арсенала: "))
print("Под индексом", index, "в арсенале находится", inventory[index])
```

Срезы списков

Готовы ли вы поверить в то, что срезы списков представляют собой то же самое, что и срезы кортежей? Опять же указываем начальную и конечную позиции через двоеточие в квадратных скобках:

```
# отобразим срез
start = int(input("\nВведите начальный индекс среза: "))
finish = int(input("Введите конечный индекс среза: "))
print("Срез inventory[", start, ":" , finish, "] - это", end=" ")
print(inventory[start:finish])
input("\nНажмите Enter, чтобы продолжить.")
```

Сцепление списков

Сцепление списков ничем не отличается от сцепления кортежей. Разница здесь будет только в том, что присвоить переменной `chest` я решил список, а не кортеж. Это небольшое, но существенное отличие, потому что конкатенации можно подвергнуть лишь однотипные последовательности.

```
# соединим два списка
chest = ["золото", "драгоценные камни"]
print("Вы нашли ларец. Вот что в нем есть:")
print(chest)
print("Вы приобщили содержимое ларца к своему арсеналу.")
inventory += chest
print("Теперь в вашем распоряжении:")
print(inventory)
input("\n\nНажмите Enter, чтобы продолжить.")
```

Изменяемость списков

Надо полагать, вас уже утомили бесчисленные фразы, общий смысл которых заключается в том, что список функционирует так же, как кортеж. До сих пор, если не считать замены круглых скобок квадратными, списки не проявили себя ничем

новым по сравнению с кортежами. Но разница все же есть, и она огромна. Повторю: списки изменяемы. Отсюда следует, что со списками можно проделывать множество трюков, для которых кортежи непригодны.

Присвоение нового значения элементу, выбранному по индексу

Поскольку список изменяем, то любому его элементу можно присвоить новое значение:

```
# присваиваем значение элементу по индексу
print("Вы обменяли меч на арбалет.")
inventory[0] = "арбалет"
print("Теперь ваш арсенал содержит следующие предметы:")
print(inventory)
input("\nНажмите Enter, чтобы продолжить.")
```

В этом коде элементу списка `inventory`, имеющему индекс 0, присваивается значение "арбалет":

```
inventory[0] = "арбалет"
```

Это новое строковое значение замещает прежнее ("меч"). Как стал выглядеть список, показывает функция `print`, с помощью которой на экран выводится новая версия `inventory`.

ЛОВУШКА

Можно присвоить новое значение существующему элементу списка, выбрав его по индексу, но создать этим способом новый элемент нельзя. Попытка присвоить какое-либо значение элементу, которого ранее не существовало, вызовет ошибку.

Присвоение новых значений срезу списка

Разрешается присвоить новое значение не только отдельному элементу списка, но и срезу. Я присвоил срезу `inventory[4:6]` значение списка из одного элемента ["магический кристалл"]:

```
# приписываем значение элементам по срезу индексов
print("За золото и драгоценные камни вы купили магический кристалл, способный пред-
сказывать будущее.")
inventory[4:6] = ["магический кристалл"]
print("Теперь в вашем распоряжении:")
print(inventory)
input("\nНажмите Enter, чтобы продолжить.")
```

Двум элементам списка — `inventory[4]` и `inventory[5]` — присваивается единое строковое значение "магический кристалл". Поскольку одноэлементный список стал значением двухэлементного среза, длина последовательности `inventory` уменьшилась на единицу.

Удаление элемента списка

Элемент списка можно удалить командой `del`, после которой этот выбранный элемент и указывается:

```
# удаляем элемент
print("В тяжелом бою был раздроблен ваш щит.")
del inventory[2]
print("Вот что осталось в арсенале:")
print(inventory)
input("\n\nНажмите Enter, чтобы продолжить.")
```

После исполнения этого кода элемент, ранее находившийся в позиции 2, то есть "щит", исчезнет из списка `inventory`. Удалить элемент — не значит создать пропуск в последовательности: длина списка уменьшится на единицу, то же произойдет с индексами элементов, которые следуют за удаленным. Иными словами, элемент в позиции 2 по-прежнему найдется в списке: это будет бывший заполнитель позиции 3.

Удаление среза списка

Из списка можно удалить и срез элементов:

```
# удаляем срез
print("Воры лишили вас арбалета и кольчуги.")
del inventory[:2]
print("В арсенале теперь только:")
print(inventory)
input("\n\nНажмите Enter, чтобы выйти.")
```

Срез `inventory[:2]`, выглядящий как `["арбалет", "кольчуга"]`, удаляется из списка `inventory` следующей командой:

```
del inventory[:2]
```

Как и при удалении элемента, длина списка сокращается. Оставшиеся элементы образуют непрерывно индексированный список, номера позиций в котором по-прежнему начинаются с 0.

Применение списочных методов

Списки располагают методами, с помощью которых ими можно управлять. Списочные методы позволяют добавить элемент, удалить элемент, выбранный по его значению, сортировать список и даже обратить порядок его элементов.

Знакомство с программой «Рекорды»

В программе «Рекорды» на основе списочных методов создается и действует подборка вымышленных лучших результатов игроков в какую-либо компьютерную игру. У программы простой интерфейс — текстовое меню. Пользователю предо-

ставляется на выбор несколько альтернатив: добавить новый рекорд, удалить один из рекордов, упорядочить список или выйти. Работа программы проиллюстрирована на рис. 5.6.

```

Рекорды
0 - Выйти
1 - Показать рекорды
2 - Добавить рекорд
3 - Удалить рекорд
4 - Сортировать список

Ваш выбор: 2

Впишите свой рекорд: 1000

Рекорды
0 - Выйти
1 - Показать рекорды
2 - Добавить рекорд
3 - Удалить рекорд
4 - Сортировать список

Ваш выбор: 1

Рекорды
1000

Рекорды
0 - Выйти
1 - Показать рекорды
2 - Добавить рекорд
3 - Удалить рекорд
4 - Сортировать список

Ваш выбор:
  
```

Рис. 5.6. Из меню пользователь может перейти к одной из функций работы со списком рекордов. Все дальнейшие процедуры незаметно для посторонних глаз проделывают списочные методы

Настройка программы

Наши начальные допущения очень просты. Вслед за комментариями, открывающими код, я создал две переменные. Первая из них — `scores` — представляет собой список, в котором будут храниться рекорды. Ее начальное значение — пустой список. Переменная `choice` содержит пункт меню, выбранный пользователем. Она инициализируется со значением `None`. Код программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 5. Файл называется `high_scores.py`.

```

# Рекорды
# Демонстрирует списочные методы
scores = []
choice = None
  
```

Отображение меню

Основа программы — цикл `while`. Его работа будет продолжаться до тех пор, пока пользователь не введет 0. Следующий код отвечает за вывод меню на экран и получение пользовательского ввода:

```

while choice != "0":
    print(
    """
Рекорды
0 - Выйти
1 - Показать рекорды
2 - Добавить рекорд
3 - Удалить рекорд
4 - Сортировать список
"""
)
choice = input("Ваш выбор: ")
print()

```

Выход из программы

Сначала я проверю, не желает ли пользователь покинуть программу. Если введен 0, компьютер пишет: До свидания.

```

# выход
if choice == "0":
    print("До свидания.")

```

После того как пользователь введет 0, условие цикла `while` при следующей проверке окажется ложным. Прервется работа цикла, а вместе с ним и всей программы.

Отображение списка рекордов

Если введено 1, будет исполнен блок с `elif`, который показывает рекорды:

```

# вывод лучших результатов на экран
elif choice == "1":
    print("Рекорды")
    for score in scores:
        print(score)

```

Добавление рекорда

Если введено 2, то компьютер спросит пользователя о значении рекорда, который тот хотел бы добавить, и сохранит это значение в переменной `score`. В последней строке следующего фрагмента кода используется списочный метод `append()`, с помощью которого `score` добавляется в конец списка `scores`. Список становится на один элемент длиннее.

```

# добавление рекорда
elif choice == "2":
    score = int(input("Впишите свой рекорд: "))
    scores.append(score)

```

Удаление рекорда

Если введено 3, то компьютер спросит пользователя о значении рекорда, который тот хотел бы удалить. Если в списке рекордов обнаружится такое значение, то первый по порядку элемент, равный ему, будет удален. Если нет, то программа уведомит об этом.

```
# удаление рекорда
elif choice == "3":
    score = int(input("Какой из рекордов удалить?: "))
    if score in scores:
        scores.remove(score)
    else:
        print("Результат", score, "не содержится в списке рекордов.")
```

Этот код сначала проверяет, является ли `score` членом последовательности `scores`. Если да, то будет вызван списочный метод `remove()`. Этот метод последовательно, начиная с позиции 0, перебирает элементы списка и сравнивает с заданным (в нашем случае `score`). Первое по порядку значение, совпадающее со значением `score`, удаляется из списка. Если искомое значение содержится в списке несколько раз, будет удалено только первое его вхождение. Это верно как вообще, так и в частности по отношению к `score`. После того как метод `remove()` благополучно удалит один элемент, список станет на единицу короче.

Заметьте, чем отличается `remove()` от `del`. Метод `remove()` удаляет элемент, заданный не номером позиции, а значением.

ЛОВУШКА

Будьте аккуратны при использовании метода `remove()`. Если попробовать удалить из списка значение, которого в нем нет, интерпретатор выдаст ошибку. Безопасная форма записи — именно такая, как в нашем коде:

```
if score in scores:
    scores.remove(score)
```

Сортировка списка рекордов

Порядок рекордов в списке совпадает с тем порядком, в котором их ввел пользователь. Для сортировки следует ввести 4:

```
# сортировка рекордов
elif choice == "4":
    scores.sort(reverse=True)
```

Метод `sort()` упорядочивает элементы списка. Это отличный инструмент, но по умолчанию сортировка проводится в возрастающем порядке, от меньших значений к большим. Вопреки этому, я хотел бы, чтобы более выдающиеся рекорды отображались выше в списке. Оказывается, метод `sort()` можно заставить упорядочивать элементы списка и по убыванию. Для этого надо передать параметру `reverse` значение `True`. Именно это и сделано здесь. Как следствие, в отсортированном списке наибольшие числа будут наверху.

ПОДСКАЗКА

Чтобы упорядочить список от меньшего к большему, вызовите метод, не передавая никаких значений параметрам. Если бы, например, я хотел упорядочить по возрастанию список numbers, я бы написал так:

```
numbers.sort()
```

Обработка ошибочного выбора

Случай, когда пользователь вводит число, которого нет в меню, обрабатывает заключительное условие else. Программа извещает пользователя о том, что его выбор непонятен системе.

```
# непонятный пользовательский ввод
else:
    print("Извините, в меню нет пункта", choice)
```

Ожидаем пользователя

После того как пользователь, желая покинуть программу, введет 0, цикл завершится, но, как и всегда, программа должна дождаться от пользователя окончательного подтверждения:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

Итак, вы увидели в действии несколько полезных списочных методов. Краткое резюме по этим и некоторым другим методам содержится в табл. 5.1.

Таблица 5.1. Избранные списочные методы

Метод	Описание
append(значение)	Добавляет значение в конец списка
sort()	Сортирует элементы по возрастанию. Необязательный параметр reverse принимает логическое значение. Если передать ему True, список будет отсортирован по убыванию
reverse()	Обращает порядок элементов списка
count(значение)	Сообщает, сколько раз данное значение входит в список
index(значение)	Возвращает номер первой из позиций списка, которые заполнены данным значением
insert(i, значение)	Вставляет значение в позицию номер i
pop([i])	Возвращает значение в позиции номер i и удаляет этот элемент из списка. Указывать число i не обязательно. Если этот аргумент не передан, возвращается и удаляется последний элемент списка
remove(значение)	Удаляет первое вхождение данного значения в список

Когда использовать кортежи, а когда — списки

Не исключаю, что сейчас вы спрашиваете себя: «К чему вообще эти кортежи?» В самом деле, списки обладают той же функциональностью, что и кортежи, а также

некоторыми дополнительными возможностями. Но не спешите отказываться от кортежей. В мире Python у них есть свое традиционное место под солнцем, и в некоторых случаях создать кортеж целесообразнее, чем список.

- Кортежи быстрее работают. Системе известно, что кортеж не изменится. Поэтому его можно сохранить таким образом, что операции с его элементами будут выполняться быстрее, чем с элементами списка. В небольших программах эта разница в скорости никак не проявит себя, но в более сложных проектах, где последовательности данных очень длины, она может оказаться значительной.
- Неизменяемость кортежей делает их идеальным средством создания констант, от которых мы и не ждем изменяемости. Константы, заданные кортежами, сделают код понятнее и безопаснее.
- Кортежи применимы в отдельных структурах данных, от которых Python требует неизменяемых значений. Ни одной из таких структур вы пока не видели на примере, но, поговорив, сама ситуация довольно распространенная. Мы столкнемся с ней уже в этой главе, в разделе «Использование словарей». Словарь должен состоять из данных неизменяемого типа, поэтому в основе некоторых разновидностей словарей лежат кортежи.

Однако, конечно, нельзя отрицать, что гибкость списков гораздо выше. Страйтесь в основном пользоваться ими, а не кортежами.

Вложенные последовательности

Выше было сказано, что списки и кортежи допускают элементы любого типа. Если это верно, то должны существовать такие списки/кортежи, которые состоят из других списков/кортежей. Они на самом деле есть — это так называемые *вложенные последовательности*. Вложенной называется последовательность, которая содержится внутри другой последовательности. Вложение — прекрасный способ организации сложных наборов данных.

Сам термин пришел из математики и прижился в жаргоне программистов. Я готов утверждать, что вложенные последовательности уже знакомы вам из жизненного опыта. Пусть, например, перед праздником вы решили купить подарки родственникам и друзьям. Очевидно, вы выпишете на листке несколько имен и рядом с каждым перечислите возможные варианты подарка. Вот вы и создали вложенную последовательность: в списке имен за каждым именем стоит список подарков.

Знакомство с программой «Рекорды 2.0»

В нашей предыдущей программе «Рекорды» были доступны только сами рекордные счета. Но, как правило, в списках игровых достижений результат сопровождается именем игрока. Это реализовано в новой версии программы, которая улучшена и в другом: теперь список рекордов сортируется автоматически, а на экране отображаются только пять наивысших достижений. Рабочее окно программы показано на рис. 5.7.

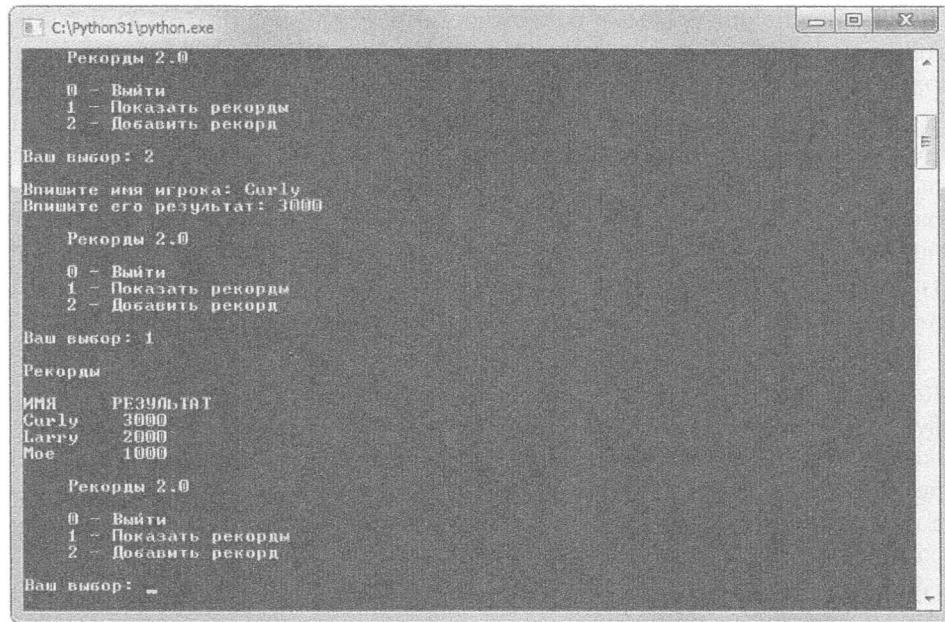


Рис. 5.7. В новой, улучшенной версии «Рекордов» имена хранятся вместе с рекордами во вложенных последовательностях

Создаем вложенные последовательности

Список или вложенный кортеж можно создать обычным образом: набрать все элементы, разделить их запятыми и окружить скобками, квадратными или круглыми. Разница в том, что среди элементов будут вложенные последовательности — целые списки или кортежи. Вот пример:

```
>>> nested = ["раз", ("два", "три"), ["четыре", "пять", "шесть"]]
>>> print(nested)
['раз', ('два', 'три'), ['четыре', 'пять', 'шесть']]
```

Здесь шесть строк, но список `nested` состоит всего из трех элементов. Первый из них — строка "раз", второй — кортеж ("два", "три"), а третий — список ["четыре", "пять", "шесть"].

Можно создать список или кортеж, содержащий любое количество других списков или кортежей. Но на практике применяются лишь некоторые единообразные формы вложения. Обратите внимание на следующий пример:

```
>>> scores = [("Маша", 1000), ("Вася", 1500), ("Петя", 3000)]
>>> print(scores)
[('Маша', 1000), ('Вася', 1500), ('Петя', 3000)]
```

Список `scores` содержит три элемента. Каждый из его элементов — кортеж, который, в свою очередь, содержит два элемента: строку и число. Эта последовательность представляет собой список рекордов с именами и результатами (как настоящий!).

В данном случае известно, что Маша набрала 1000 очков, Вася — 1500, а рекордсмен Петя достиг результата 3000.

ЛОВУШКА

Хотя можно создавать последовательности с большой глубиной вложения, обычно так не делают. Такие списки, как в следующем примере, считаются неудачными:

```
nested = ("глубоко", ("глубже", ("еще", "и еще глубже")))
```

Вложение провоцирует ошибки. Вот почему даже опытные программисты почти не пользуются последовательностями с глубиной вложенности больше двух. Для большинства ваших программ будет достаточно одного уровня вложенности, как в показанном только что списке рекордов scores.

Доступ к вложенным элементам

К вложенной последовательности, как и ко всякой другой последовательности, применимо индексирование:

```
>>> scores = [("Маша", 1000), ("Вася", 1500), ("Петя", 3000)]
>>> print(scores[0])
('Маша', 1000)
>>> print(scores[1])
('Вася', 1500)
>>> print(scores[2])
('Петя', 3000)
```

Здесь каждый элемент — кортеж. Его мы и получаем, вызвав элемент с определенным индексом. Но что, если нам надо получить доступ к одному из элементов одного из кортежей? Один способ заключается в том, чтобы создать для этого кортежа переменную и извлечь из нее элемент по индексу, например, так:

```
>>> a_score = scores[2]
>>> print(a_score)
('Петя', 3000)
>>> print(a_score[0])
Петя
```

Но извлечь строку "Петя" из списка scores можно и более прямым путем:

```
>>> print(scores[2][0])
Петя
```

Два индекса в записи scores[2][0] говорят компьютеру, что мы хотим взять элемент списка scores в позиции номер 2 (это ("Петя", 3000)), а из него, в свою очередь, взять элемент в позиции номер 0 (это "Петя"). Множественную индексацию такого типа можно использовать в списках с вложением, чтобы напрямую обращаться к вложенным элементам.

Распаковка последовательности

Если известно, сколько элементов есть в последовательности, то можно создать для каждого из них особую переменную, не затратив больше одной строки кода:

```
>>> name, score = ("Иван Иванович", 175)
>>> print(name)
Иван Иванович
>>> print(score)
175
```

Это так называемая *распаковка* — операция, которая применима к последовательностям любого вида. Помните, что надо подставить столько же самостоятельных переменных, сколько элементов в последовательности. Иначе интерпретатор сообщит об ошибке.

Настройка программы

Как и в первой версии программы «Рекорды», назначим начальные значения переменных и условие цикла `while`. Когда пользователь вводит 0, компьютер прощается с ним. Код этой программы можно найти на сайте-помощнике (courseptr.com/downloads) в папке **Chapter 5**. Файл называется `high_scores2.py`.

```
# Рекорды 2.0
# Демонстрирует вложенные последовательности
scores = []
choice = None
while choice != "0":
    print(
        """
Рекорды 2.0
0 - Выйти
1 - Показать рекорды
2 - Добавить рекорд
"""
    )
    choice = input("Ваш выбор: ")
    print()
    # выход
    if choice == "0":
        print("До свидания.")
```

Вывод результатов, содержащихся во вложенных кортежах

Если введено 1, компьютер перебирает элементы списка `scores` и распаковывает имя каждого рекордсмена и его результат в переменные `score` и `name`, которые затем печатаются на экране.

```
# вывод таблицы рекордов
elif choice == "1":
    print("Рекорды\n")
    print("ИМЯ\tРЕЗУЛЬТАТ")
    for entry in scores:
```

```
score, name = entry
print(name, "\t", score)
```

Добавление результата как вложенного кортежа

Если введено 2, компьютер предлагает пользователю указать имя рекордсмена и его достижение. Эти два значения сохраняются в кортеже `entry`. Я решил первым значением в кортеже сохранять сам результат (`score`), чтобы сортировать рекорды сначала по абсолютной величине, а затем уже по имени (`name`). Введенный пользователем рекорд компьютер добавляет к списку, затем сортирует список и обращает порядок его элементов, чтобы более высокие результаты шли первыми. В последней строке списку `scores` присваивается значение среза, содержащего пять лучших достижений.

```
# add a score
elif choice == "2":
    name = input("Впишите имя игрока: ")
    score = int(input("Впишите его результат: "))
    entry = (score, name)
    scores.append(entry)
    scores.sort(reverse=True)
    scores = scores[:5] # в списке остается только 5 рекордов
```

Обработка ошибочного выбора

На случай, когда пользовательский ввод не равен 0, 1 или 2, нужно заключительное условие `else`. Программа извещает пользователя о том, что его выбор непонятен системе.

```
# непонятный пользовательский ввод
else:
    print("Извините, в меню нет пункта", choice)
```

Ожидаем пользователя

После того как пользователь, желая покинуть программу, введет 0, цикл завершится:

```
input("\n\nНажмите Enter, чтобы выйти.")
```

Распределенные ссылки

Из главы 2 вы узнали о том, что переменная ссылается на значение. Если говорить более детально, это значит, что переменная не хранит в себе копию значения, а только ссылается на такой участок оперативной памяти, где хранится это значение. Например, команда `language = "Python"` говорит о том, что сначала мы сохранили строку "Python" где-либо в памяти компьютера, а затем создали переменную `language`, которая ссылается на адрес этой строки в памяти. Обратите внимание на рис. 5.8, где то же самое показано более наглядно.



Рис. 5.8. Переменная `language` ссылается на адрес в памяти, по которому «прописано» строковое значение «`Python`»

Полагать, что строка `"Python"` хранится в переменной, как жареная курица — в пластиковом контейнере, — это, мягко говоря, неточно. Для некоторых языков программирования такая аналогия удачна, но `Python` не принадлежит к их числу. Лучше представить все так: переменная ссылается на значение, как имя человека ссылается на самого этого человека. Было бы ошибкой и даже глупостью утверждать, будто человек «содержится» в своем имени. Зная имя человека, можно получить доступ к нему самому; зная имя переменной, можно получить доступ к ее значению.

Что же все это значит? Для неизменяемых типов, с которыми вы уже освоились (числа, строки, кортежи), пожалуй, ничего. Но по отношению к таким изменяемым типам, как списки, важность этих замечаний очень велика. Если несколько переменных ссылаются на одно и то же изменяемое значение, такие ссылки называются *распределенными*. Переменных несколько, а значение в памяти одно. Поэтому если через одну из переменных это значение подвергается модификации, то и все остальные переменные будут ссылаться на модифицированное, обновленное значение.

Поясню эту сухую теорию на примере. Вообразим, что я созвал своих друзей и множество мировых знаменитостей на шикарную вечеринку. (Это моя книга. В ней есть даже «Блек-джек» — в главе 9. Так отчего бы не быть вечеринке со знаменитостями?) Разные люди на этом мероприятии называют меня по-разному, хотя я от этого не перестаю быть одним и тем же человеком. Кто-нибудь из друзей назовет меня Майком, чиновник или бизнесмен обратится ко мне как к мистеру Доусону, а моя подружка, обладательница Пулитцеровской премии и супермодель, которая только-только вернулась из мирового благотворительного турне (почему бы мне в собственной книге не придумать себе подружку-супермодель?), — вот эта подружка именует меня «милый». Итак, три человека называют меня тремя разными именами. Точно так же три переменные могут ссылаться на один и тот же список. Вот начало интерактивной сессии, из которой вы поймете, что я имею в виду:

```

>>> mike = ["белая рубашка", "комбинезон цвета хаки", "пиджак"]
>>> mr_dawson = mike
>>> honey = mike
>>> print(mike)
['белая рубашка', 'комбинезон цвета хаки', 'пиджак']
>>> print(mr_dawson)
['белая рубашка', 'комбинезон цвета хаки', 'пиджак']
>>> print(honey)
['белая рубашка', 'комбинезон цвета хаки', 'пиджак']
  
```

Все три переменные — `mike`, `mr_dawson` и `honey` — ссылаются на один и тот же список, представляющий меня самого или по крайней мере мой экстравагантный наряд на вечеринке. Более наглядно то же самое представлено на рис. 5.9.

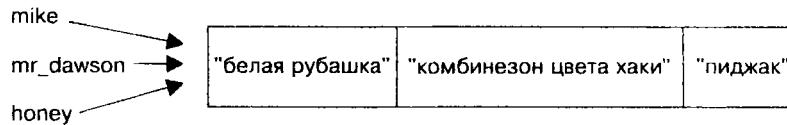


Рис. 5.9. Переменные mike, mr_dawson и honey ссылаются на один и тот же список

Это означает, что изменения, внесенные через любую из трех переменных, будут отражены в списке, на который они все ссылаются. Представим себе, что во время вечеринки подружка подзывает меня («Милый!») и предлагает надеть вместо пиджака красный свитер, который она собственноручно связала. (Она еще и вяжет!) Я, конечно, слушаюсь и повинуюсь. В интерактивной сессии происходящее может быть отражено так:

```

>>> honey[2] = "красный свитер"
>>> print(honey)
['белая рубашка', 'комбинезон цвета хаки', 'красный свитер']
  
```

Получаем то, что и ожидали получить. Элемент в позиции 2 в списке, на который ссылаются переменная honey, теперь уже не «пиджак», а «красный свитер».

Вернемся на вечеринку. Теперь друзья и приглашенные знаменитости будут видеть «Майка» и «мистера Доусона» в красном свитере, хотя от их внимания ускользнуло, когда он успел переодеться. В Python все точно так же. Хотя я изменил значение элемента в позиции 2 через переменную honey, во всех прочих переменных, которые ссылаются на тот же список, значение тоже будет обновлено. Вот логичное продолжение интерактивной сессии:

```

>>> print(mike)
['белая рубашка', 'комбинезон цвета хаки', 'красный свитер']
>>> print(mr_dawson)
['белая рубашка', 'комбинезон цвета хаки', 'красный свитер']
  
```

В позиции номер 2 в списке, на который ссылаются переменные mike и mr_dawson, обнаруживается «красный свитер». Это закономерно, ведь список все время один и тот же.

Мораль сей басни такова: когда работаете с изменяемыми значениями, помните о распределенных ссылках. Если меняется значение в одной переменной, то оно меняется и во всех переменных, приравненных к ней. Этого эффекта можно избежать, если сделать полный срез (копию) списка, например:

```

>>> mike = ["белая рубашка", "комбинезон цвета хаки", "пиджак"]
>>> honey = mike[:]
>>> honey[2] = "красный свитер"
>>> print(honey)
['белая рубашка', 'комбинезон цвета хаки', 'красный свитер']
>>> print(mike)
['белая рубашка', 'комбинезон цвета хаки', 'пиджак']
  
```

Здесь значение переменной honey — копия mike. Переменная honey ссылается не на тот же самый список, а на его копию, поэтому изменения, вносимые в значение honey, никак не сказываются на mike. Это как если бы меня клонировали и подружка

нарядила моего клона в красный свитер, а я по-прежнему оставался в пиджаке. Да уж, хорошенькая получается вечеринка: в свитере, который связала вымышленная супермодель, по залу разгуливает точная копия меня! Наконец оставим эту аналогию: несмотря на ее удобство, шутка несколько затянулась.

Советую вам запомнить, что эффект распределенных ссылок желателен в одних случаях и нежелателен в других. Теперь, когда вы о нем знаете, можете обратить его себе во благо.

Использование словарей

От вас, конечно, не укрылось, что программисты любят разными способами организовывать данные. Так, списки и кортежи, как вы видели, позволяют упорядочить данные в виде последовательностей. Со словарями дело обстоит немного иначе. В них информация представлена не как последовательность элементов, а как набор пар. Заметно сходство с обычным толковым словарем, в котором каждую статью образует пара: слово и его определение. Открыв словарь на каком-либо слове, можно получить определение этого слова; это верно и для словарей Python, которые можно «открывать» на каком-либо *ключе* и получать соответствующее значение.

Знакомство с программой «Переводчик с гикского на русский»

Многие порождения мира высоких технологий так или иначе сказываются на нашей обыденной жизни. Среди них и своеобразная культура компьютерщиков. Техника, с которой они работают, вносит в их речь новые, не существовавшие ранее слова и понятия. Программа «Переводчик с гикского на русский» поможет вам поддержать разговор, если когда-либо придется беседовать с англоговорящим гиком — человеком, «поворнутым» на электронике, программировании и Интернете. В программе создан словарь с терминами компьютерного жаргона (языка гиков) и их русскоязычными определениями и аналогами. Пользователю предоставляется возможность не только читать толкования слов, но и добавлять статьи, удалять их, а также изменять толкования. Работа программы показана на рис. 5.10.

Создание словарей

Первым делом в этой программе я создал словарь, содержащий термины и определения. Слова из языка гиков помещаются слева, а их толкования — справа. Код программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 5. Файл называется `geek_translator.py`.

```
# Переводчик с гикского на русский
# Демонстрирует использование словарей
geek = {"404": "Не знать, не владеть информацией. От сообщения об ошибке 404 'Страница не найдена'.",
        "Googling": "'Гугление'. поиск в Сети сведений о ком-либо."}
```

"Keyboard Plaque" : "Мусор, который скапливается в клавиатуре компьютера." ,

"Link Rot" : "Процесс устаревания гиперссылок на веб-страницах." ,

"Percussive Maintenance" : "О ситуации, когда кто-либо бьет по корпусу неисправного электронного прибора в надежде восстановить его работу." ,

"Uninstalled" : "Обувь снятой с человека. Особенно популярно на рубеже 1990-2000-х годов."}

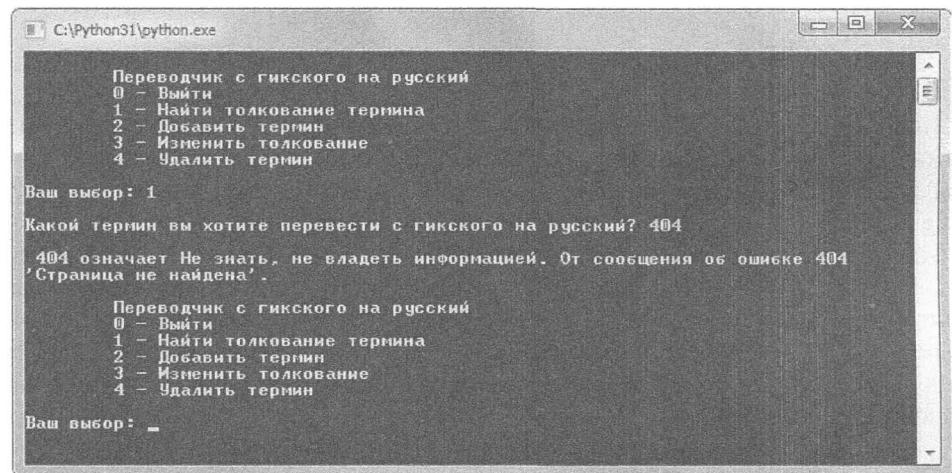


Рис. 5.10. Обувь снятой с человека скажет: «Меня деинсталлировали»

В этом коде создается словарь под названием `geek`. Он состоит из шести пар. Одна из этих пар, например, выглядит так: "Keyboard Plaque" : "Мусор, который скапливается в клавиатуре компьютера". Элементы пары называют ключом и значением. Ключ находится слева от двоеточия, а значение — справа. Так, "Keyboard Plaque" — это ключ, а "Мусор, который скапливается в клавиатуре компьютера" — значение. С помощью ключа можно буквально «открыть» (извлечь) соответствующее значение. Таким образом, по ключу "Keyboard Plaque" доступна строка "Мусор, который скапливается в клавиатуре компьютера".

При создании собственных словарей следуйте моему образцу. Введите ключ, поставьте двоеточие и затем введите значение, соответствующее ключу. Пары «ключ — значение» отделяйте друг от друга запятыми, а весь получившийся набор заключите в фигурные скобки. Подобно кортежам и спискам, можно или расположить словарь на одной строке кода, или разбить его на несколько строк, которые бы заканчивались разделителями — запятыми.

Доступ к значениям в словаре

Одна из самых типичных процедур, которым можно подвергать словарь, — извлечение значений. Можно извлекать значения несколькими разными способами; каждый из них я продемонстрирую в этом подразделе на примере интерактивной сессии.

Доступ к значению по ключу

Проще всего извлечь значение из словаря напрямую, вызвав соответствующий этому значению ключ. Достаточно ввести имя словаря и вслед за ним в квадратных скобках указать ключ. В предположении, что словарь `geek` уже задан, проведем следующую интерактивную сессию:

```
>>> geek["404"]
```

«Не знать, не владеть информацией. От сообщения об ошибке 404 'Страница не найдена'.'

```
>>> geek["Link Rot"]
```

‘Процесс устаревания гиперссылок на веб-страницах.’

Похоже на индексирование последовательности, но есть важное отличие. Извлекая элемент последовательности по его индексу, мы пользуемся номером позиции, а извлекая значение словаря по ключу — ключом. Это единственный способ получить доступ к значению напрямую. Номеров позиций в словарях нет.

Начинающих программистов иногда вводит в замешательство тот факт, что извлечь из словаря ключ по значению нельзя. Это как если бы в обычном словаре, напечатанном на бумаге, мы попробовали найти слово, зная, как оно толкуется. Толковые словари для таких целей просто не предназначены, и словарный тип данных в Python им в этом вторит. Поэтому запомните: вводим ключ — получаем значение, но наоборот.

ЛОВУШКА

Если попробовать извлечь из словаря значение, соответствующее ключу, которого в нем нет, интерпретатор сообщит об ошибке:

```
>>> geek["Dancing Baloney"]
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
geek["Dancing Baloney"]
KeyError: 'Dancing Baloney'
```

В словаре отсутствует ключ "Dancing Baloney", и мы получаем ошибку (кстати, этим выражением — в буквальном переводе «пляшущая дребедень» — зарубежные компьютерщики называют анимированную графику и другие визуальные эффекты, неважные сами по себе, но часто применяемые веб-дизайнерами, чтобы произвести впечатление на заказчика).

Проверка существования ключа с помощью оператора `in`

Поскольку несуществующий ключ вызывает ошибку в работе интерпретатора, значения из словаря принято брать с соблюдением определенных мер предосторожности. Обычно, прежде чем попробовать извлечь значение по ключу, проверяют, есть ли такой ключ в словаре. Проверку поможет выполнить оператор `in`:

```
>>> if "Dancing Baloney" in geek:
    print("Я знаю, что такое Dancing Baloney.")
else:
    print("Я понятия не имею, что такое Dancing Baloney.")
```

Я понятия не имею, что такое Dancing Baloney.

В словаре нет ключа "Dancing Baloney", поэтому условие "Dancing Baloney" in `geek` ложно и компьютер честно признается в своей некомпетентности.

Оператор `in` применяется к словарям почти так же, как к спискам и кортежам. Надо ввести строку, на наличие которой мы проверяем множество ключей словаря, затем `in` и имя словаря. Получается условие. Оно истинно, если такой ключ в словаре есть, и ложно в противном случае.

Целесообразно перед доступом к значению проверять, задан ли соответствующий ключ. А вот обратная процедура (проверять, задано ли значение) невозможна, да и не нужна.

Доступ к значениям с помощью метода `get()`

Есть еще один способ извлечь значение из словаря — применение словарного метода `get()`. Этот метод надежно обрабатывает такие ситуации, как запрос значения по несуществующему ключу. Если ключа нет, то метод возвратит значение по умолчанию, которое вы можете специально задать. Рассмотрим такой пример:

```
>>> print(geek.get("Dancing Baloney", "Понятия не имею."))  
Понятия не имею.
```

Здесь я применил метод `get()` и тем самым определил, что какое-то значение обязательно будет возвращено. Если бы в словаре имелся искомый ключ, мне было бы сообщено толкование термина. На самом деле такого ключа не оказалось, и интерпретатор возвратил заданное мной значение по умолчанию "Понятия не имею".

Метод `get()` требует выбрать только ключ, соответствующее которому значение вы хотите запросить из словаря; значение по умолчанию не является обязательным. Для ключа, который есть в словаре, метод возвращает значение из той же пары, а для ключа, которого нет, — значение по умолчанию или (если вы решили его не задавать) значение `None`. Вот пример:

```
>>> print(geek.get("Dancing Baloney"))  
None
```

Настройка программы

Пришло время вернуться к коду «Переводчика с гикского на русский». Итак, я создал словарь `geek` и вслед за тем реализовал уже знакомое вам меню, на сей раз с пятью пунктами. Как и ранее, если пользователь выбирает пункт 0, компьютер прощается с ним.

```
choice = None  
while choice != "0":  
    print(  
        """  
    Переводчик с гикского на русский  
    0 - Выйти  
    1 - Найти толкование термина  
    2 - Добавить термин  
    3 - Изменить толкование  
    4 - Удалить термин  
    """  
)  
    choice = input("Ваш выбор: ")
```

```
print()
# выход
if choice == "0":
    print("До свидания.")
```

Поиск значения

Если введено 1, пользователю предлагается ввести слово или выражение из жаргона компьютерщиков, которое ему непонятно. Программа проверит, есть ли этот термин в словаре. Если есть, то значение (определение), соответствующее ключу (термину), будет извлечено из словаря и выведено на экран. Если нет, то программа уведомит пользователя об этом.

```
# поиск толкования
elif choice == "1":
    term = input("Какой термин вы хотите перевести с гикского на русский? ")
    if term in geek:
        definition = geek[term]
        print("\n", term, "означает", definition)
    else:
        print("\nУвы, этот термин мне незнаком:", term)
```

Добавление пары «ключ — значение»

Словари — изменяемый тип данных; их элементы можно менять, например добавлять. Вводя 2, пользователь получает возможность добавить термин и его толкование в словарь:

```
# добавление термина с толкованием
elif choice == "2":
    term = input("Какой термин гикского языка вы хотите добавить? ")
    if term not in geek:
        definition = input("\nВпишите ваше толкование: ")
        geek[term] = definition
        print("\nТермин", term, "добавлен в словарь.")
    else:
        print("\nТакой термин уже есть! Попробуйте изменить его толкование.")
```

Программа предлагает пользователю ввести слово или выражение. Если такого слова или выражения в словаре нет, то программа запрашивает пользователя о толковании и добавляет пару в словарь. За этот последний шаг отвечает строка:

```
geek[term] = definition
```

Здесь в словаре `geek` создается новая пара, в которой `term` — ключ, а `definition` — значение. Таким способом вообще принято определять новые пары элементов словаря: пишем имя переменной, ссылающейся на словарь, потом в квадратных скобках ключ, затем оператор присвоения (`=`) и, наконец, значение.

В моей программе предусмотрено, что через пункт меню 2 нельзя заново толковать уже заданный термин. Это мера безопасности, к которой я решил прибегнуть, чтобы пользователь случайно не переопределил одно из слов или выражений. Если он на самом деле хочет поменять определение, пусть обратится к пункту меню 3.

ХИТРОСТЬ

В деле создания компьютерных программ полезна доля пессимизма. Я допустил, что пользователь может попытаться добавить в словарь какой-либо термин, не сознавая, что этот термин уже есть в словаре. Если бы я смотрел на мир более радужно, то по крайней мере часть пользователей не вольно переопределяла бы некоторые термины. Когда вы будете создавать собственные программы, старайтесь быть хоть чуть-чуть пессимистом. Подумайте о возможных вариантах ошибочного поведения человека и предусмотрите обработку этого поведения.

Замена пары «ключ — значение»

Если введено 3, то значение при одном из существующих ключей будет заменено новым благодаря такому коду:

```
# новое толкование известного термина
elif choice == "3":
    term = input("Какой термин вы хотите переопределить? ")
    if term in geek:
        definition = input("Впишите ваше толкование: ")
        geek[term] = definition
        print("\nТермин", term, "переопределен.")
    else:
        print("\nТакого термина пока нет! Попробуйте добавить его в словарь..")
```

Замену значения выполняет та же строка, с помощью которой мы ранее создавали новую пару:

```
geek[term] = definition
```

ЛОВУШКА

Интерпретатор Python просто подставляет новое значение (в данном случае определение) вместо существующего. Если ключу, который входит в словарь, сопоставить новое значение, никакой ошибки не произойдет. Чтобы не переписывать по неосторожности значения существующих ключей, будьте внимательны.

Удаление пары «ключ — значение»

Если введено 4, работает следующий блок elif:

```
# удаление термина вместе с его толкованием
elif choice == "4":
    term = input("Какой термин вы хотите удалить? ")
    if term in geek:
        del geek[term]
        print("\nТермин", term, "удален.")
    else:
        print("\nНичем не могу помочь. Термина", term, "нет в словаре..")
```

Программа спрашивает пользователя, какое из слов языка гиков тот желает удалить из словаря. Затем с помощью оператора `in` выполняется проверка, есть ли на самом деле такое слово в словаре. Если есть, то команда:

```
del geek[term]
```

удаляет всю пару (ключ `term` и соответствующее значение) из словаря `geek`. Любую пару можно удалить из словаря этим способом. Достаточно написать `del`, затем через пробел имя словаря, а вслед за ним в квадратных скобках ключ той пары, которую вам угодно удалить.

Если термин не обнаружится в словаре, то будет выполнено условие `else` и компьютер известит пользователя о неудаче.

ЛОВУШКА

Если попробовать удалить из словаря пару, ключ которой не задан, то интерпретатор выдаст ошибку. Всегда стоит проверять, есть ли в словаре интересующий вас ключ.

Обработка ошибочного выбора

Заключительное условие `else` дает пользователю понять, что его выбор некорректен:

```
# непонятный пользовательский ввод
else:
    print("Извините, в меню нет пункта", choice)
    input("\n\nНажмите Enter, чтобы выйти.")
```

Особенности словарей

Создавая словарь, вы должны помнить несколько вещей.

- Словарь не может содержать несколько пар с одним и тем же ключом. Вспомните еще раз, как организован обычный толковый словарь. Он стал бы бессмысленным по своей сути, если бы одно и то же слово встречалось в нем несколько раз с совершенно противоположными значениями.
- Ключ должен быть неизменяемого типа. Это может быть строка, число или кортеж — согласитесь, выбор достаточно широк. От ключа требуют неизменяемости потому, что при модификации данных изменяемого типа можно в конце концов получить в словаре два одинаковых ключа. А это, как вы только что узнали, запрещено.
- Если ключи должны быть уникальны, то значения — нет. Кроме того, они могут быть как изменяемыми, так и неизменяемыми, то есть, в общем-то, какими угодно.

Я показал вам далеко не все приемы работы со словарями. В табл. 5.2 перечислены некоторые методы, с помощью которых вы можете расширить функциональность словарей.

Таблица 5.2. Избранные словарные методы

Метод	Описание
<code>get(ключ, [умолчание])</code>	Возвращает значение по ключу. Если ключ не определен в словаре, то возвращается умолчание. Если, в свою очередь, умолчание не задано, возвращается <code>None</code>
<code>keys()</code>	Возвращает набор всех ключей словаря

Метод	Описание
values()	Возвращает набор всех значений словаря
items()	Возвращает набор всех пар в словаре. Каждая пара представляет собой кортеж из двух элементов: первый — ключ, второй — соответствующее значение

ЛОВУШКА

Наборы ключей, значений и пар — так называемые виды (views) словаря, возвращаемые методами keys(), values() и items(), — в каком-то смысле подобны спискам. Их элементы можно перебирать с помощью цикла for. Но в действительности это не списки. К примеру, из них нельзя выбирать элементы по индексу, и добавок виды являются динамическими, то есть их содержимое основано на содержимом соответствующих словарей. Изменение в словаре сразу же отразят виды этого словаря. Подробнее о них читайте в разделе документации на официальном сайте Python (www.python.org).

Вернемся к игре «Виселица»

Для того чтобы вы могли применить на практике все, что изучили в этой главе, я приглашаю вас поучаствовать в разработке игры «Виселица», упомянутой в начале главы. Это очень длинная программа, но опасаться, что в ней будет много непонятного, не надо: код лишь чуть-чуть сложнее, чем в других проектах этой главы. Больше всего места в программе занимает мой скромный опыт художника — восемь вариантов картинки с повешенным, а основного кода всего пара десятков строк.

Настройка программы

Начнем сначала. Как обычно, код открывают комментарии, в которых изложена суть программы. Затем я импортировал модуль random, который понадобится для случайного выбора слова из последовательности.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 5. Файл называется hangman.py.

```
# Виселица
#
# Классическая игра "Виселица". Компьютер случайным образом выбирает слово,
# которое игрок должен отгадать буква за буквой. Если игрок не сумеет
# отгадать за отведенное количество попыток, на экране появится фигурка повешенного.
# импорт модуля
import random
```

Создание констант

Хотя приводимый далее в этом разделе код растянулся на несколько страниц, он отвечает всего лишь за создание трех констант. Сначала я создал кортеж, самый большой из тех, что вы когда-либо видели: последовательность из восьми элементов, каждый из которых — строка в тройных кавычках — сам охватывает 12 строк. Эти строки — псевдографические картинки с изображением виселицы. От картинки к картинке все полнее проявляется фигурка повешенного. При каждом

следующем неправильном ответе игрока будет выведена очередная картинка. Восьмое по счету изображение — последнее: на нем повешенного уже не вернуть к жизни. Если на экране появляется эта картинка, игра заканчивается. Псевдографика в моей программе хранится в кортеже `HANGMAN` (название переменной набрано прописными буквами, потому что это константа).

```
# константы  
HANGMAN = (
```

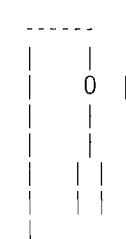
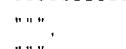
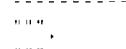
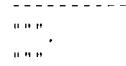
100

三三〇

0

- - - - -

9-18



""")

Затем я создал константу, значение которой — максимальное количество ошибок, дозволенных игроку:

```
MAX_WRONG = len(HANGMAN) - 1
```

Ошибок может быть ровно на одну меньше, чем элементов в кортеже HANGMAN. Это обусловлено тем, что изображение пустой виселицы есть на экране уже тогда, когда ни одной попытки отгадать слово игрок еще не израсходовал. Поэтому в HANGMAN восемь псевдографических картинок, а игрок может ошибиться семь раз, прежде чем игра будет окончена.

Здесь же я создал кортеж, из которого компьютер будет случайным образом выбирать слово для отгадывания. В него входят не очень широко известные английские слова; не бойтесь исправить код и создать свой собственный список.

```
WORDS = ("OVERUSED", "CLAM", "GUAM", "TAFFETA", "PYTHON")
```

Инициализация переменных

После этого я инициализировал переменные. С помощью функции random.choice() я выбрал случайное слово из кортежа WORDS и сделал его значением переменной word.

```
# инициализация переменных
word = random.choice(WORDS)      # слово для отгадывания
```

Еще одну строку — so_far — я создал для представления букв, уже отгаданных игроком. Начальный вид этой строки — набор дефисов, по одному на каждый символ. После того как игрок правильно отгадывает букву, дефис в той позиции (или позициях), где эта буква входит в слово, заменяется буквой.

```
so_far = "-" * len(word)        # по одному дефису на каждую букву, которую надо отгадать
```

Я создал переменную wrong с начальным значением 0. Она будет содержать число, определяющее количество ошибок, допущенных игроком при отгадывании:

```
wrong = 0      # количество ошибок, которые сделал игрок
```

Наконец, каждая очередная буква, введенная игроком, попадает в список used. В самом начале этот список пуст:

```
used = []      # буквы, которые игрок уже предлагал
```

Создание основного цикла

Я создал цикл, работа которого продолжается до тех пор, пока игрок не совершил слишком много ошибок или не отгадал все буквы в слове:

```
print("Добро пожаловать в игру 'Виселица'. Удачи вам!")
while wrong < MAX_WRONG and so_far != word:
    print(HANGMAN[wrong])
    print("\nВы уже предлагали следующие буквы: \n", used)
    print("\nОтгаданное вами в слове сейчас выглядит так: \n", so_far)
```

Этот цикл сначала выводит на экран картинку с текущим видом виселицы. Номер позиции картинки равен количеству допущенных ошибок; чем больше ошибок сделано, тем ближе повешенный к гибели. После этого выводится список букв, которые игрок уже предлагал, и частично отгаданное слово so_far.

Получение ответа игрока

Очередную букву, которую предлагает игрок, я получаю из пользовательского ввода и привожу к верхнему регистру, чтобы ее можно было найти в загаданном слове (все слова, доступные программе для загадывания, набраны прописными). После этого я проверяю, не использовал ли игрок ту же самую букву ранее. Если такая попытка уже была, то программа просит игрока ввести другой символ. Это повторится до тех пор, пока не будет наконец введена буква, о которой игрок еще не спрашивал. Допустимую, то есть еще не проверенную версию программы приводит к верхнему регистру и вносит в список used.

```
guess = input("\n\nВведите букву: ")
guess = guess.upper()
while guess in used:
    print("Вы уже предлагали букву", guess)
    guess = input("\n\nВведите букву: ")
    guess = guess.upper()
used.append(guess)
```

Проверка наличия буквы в слове

Теперь надо узнать, есть ли в загаданном слове буква, которую предложил игрок. Если есть, то программа уведомит об этом игрока и создаст новую версию строки so_far, в которой отгаданная буква появится во всех позициях, где она присутствует в самом слове.

```
if guess in word:
    print("\nДа! Буква", guess, "есть в слове!")
    # новая строка so_far с отгаданной буквой или буквами
    new = ""
    for i in range(len(word)):
        if guess == word[i]:
            new += guess
        else:
            new += so_far[i]
    so_far = new
```

Если нет, то программа с грустью сообщит об этом игроку и увеличит счетчик wrong на единицу.

```
else:
    print("\nК сожалению, буквы", guess, "нет в слове.")
    wrong += 1
```

Завершение игры

Когда исполнение кода дойдет до этого места, можно считать игру оконченной. Если количество ошибок достигло максимума, значит, игрок потерпел поражение. В этом случае на экране появляется заключительная картинка с повешенным. Если

же предельное количество ошибок не достигнуто, то программа поздравит игрока. В обоих случаях сообщается, какое слово было загадано.

```
if wrong == MAX_WRONG:
    print(HANGMAN[wrong])
    print("\nВас повесили!")
else:
    print("\nВы отгадали!")
print("\nБыло загадано слово", word)
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

Из этой главы вы узнали все о списках и словарях — двух новых типах данных. Вы рассмотрели списки как изменяемые последовательности и увидели, как добавлять, удалять, сортировать элементы списка и даже располагать их в обратном порядке. Но, как вы убедились, в некоторых случаях, несмотря на столь мощную функциональность списков, более удобным (или единственным) выбором остаются сравнительно негибкие кортежи. Теперь вы знаете, к чему может привести распределенная ссылка на данные изменяемого типа и как можно при необходимости обойтись без нее. Вы научились создавать и применять вложенные последовательности, с помощью которых можно представить данные еще более сложной структуры, например список рекордов. Кроме того, вы узнали, как создавать словари и работать с данными, организованными в виде пар «ключ — значение».

ЗАДАЧИ

- Создайте программу, которая будет выводить список слов в случайном порядке. На экране должны печататься без повторений все слова из представленного списка.
- Напишите программу «генератор персонажей» для ролевой игры. Пользователю должно быть предоставлено 30 пунктов, которые можно распределить между четырьмя характеристиками: Сила, Здоровье, Мудрость и Ловкость. Надо сделать так, чтобы пользователь мог не только брать эти пункты из общего «пула», но и возвращать их туда из характеристик, которым он решит присвоить другие значения.
- Напишите программу «Кто твой пapa?», в которой пользователь будет вводить имя человека, а программа — называть отца этого человека. Чтобы было интереснее, можно «научить» программу родственным отношениям среди литературных персонажей, исторических лиц и современных знаменитостей. Предоставьте пользователю возможность добавлять, заменять и удалять пары «сын — отец».
- Доработайте программу «Кто твой пapa?» так, чтобы можно было, введя имя человека, узнать, кто его дед. Программа должна по-прежнему пользоваться одним словарем с парами «сын — отец». Подумайте, как включить в этот словарь несколько поколений.

6 Функции. Игра «Крестики-нолики»

Программы, которые вы писали до сих пор, представляли собой длинные, целостные серии команд. Когда программа достигает определенного уровня сложности, такая организация кода становится неоптимальной. К счастью, есть способы разбить большую программу на более мелкие фрагменты, с каждым из которых в отдельности легче управляться. Одному из этих способов — созданию нестандартных функций — и посвящена данная глава.

Здесь вы научитесь:

- писать собственные функции;
- передавать этим функциям какие-либо значения с помощью параметров;
- получать значения, возвращаемые функциями;
- работать с глобальными переменными и константами;
- разрабатывать компьютерного противника для игры-стратегии.

Знакомство с игрой «Крестики-нолики»

Проект, вокруг которого построена эта глава, покажет вам, как, применяя на практике некоторые базовые идеи искусственного интеллекта (ИИ), создать виртуального противника. Компьютер будет противостоять игроку-человеку в захватывающем интеллектуальном шоу «Крестики-нолики». Пользователю суждено столкнуться с грозным (хотя и не совсем безупречным) соперником, который к тому же очень самоуверен и спесив; общение с ним заставит не раз улыбнуться. Игровой процесс отражен на рис. 6.1–6.3.

Создание функций

Вы уже видели, как работают некоторые из встроенных, или стандартных, функций, например `len()` и `range()`. Когда встроенных функций Python оказывается недостаточно, программист пишет свои собственные, нестандартные функции. Они работают точно так же, как и поставляемые вместе с дистрибутивом языка: выполняется какая-либо частная задача, и затем программа продолжает работу. У нестандартных функций много достоинств, главное из которых состоит в том, что с их помощью код

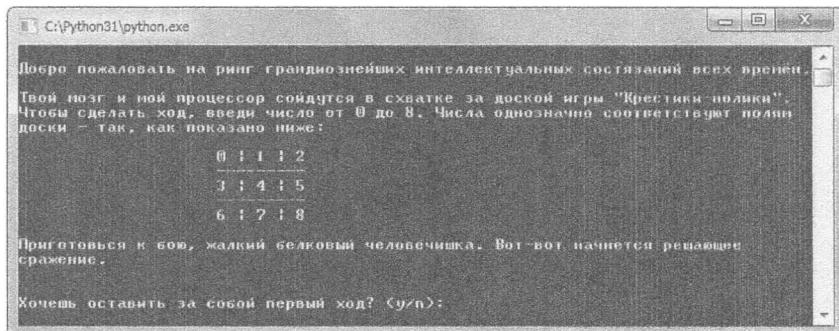


Рис. 6.1. Компьютер проявляет завидное... гм... чувство собственного достоинства

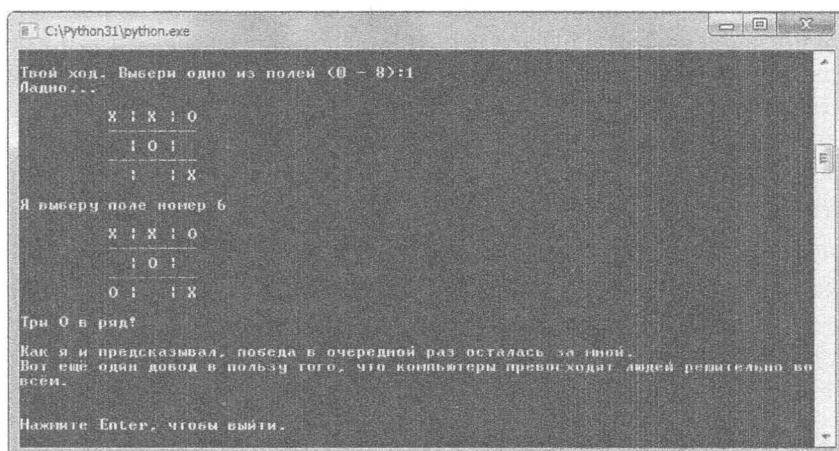


Рис. 6.2. Я просмотрел эту ловушку. Несмотря на простоту алгоритма, программа иногда делает отличные ходы

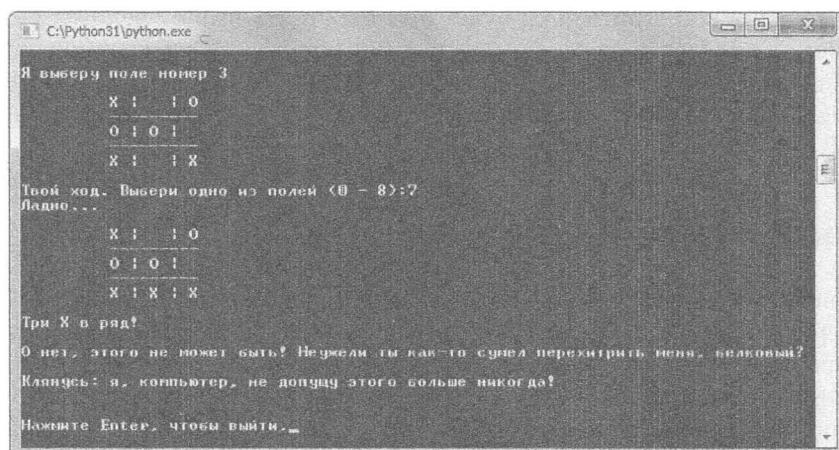


Рис. 6.3. На этот раз я нашупал слабость в обороне противника и выиграл

можно разбить на несколько фрагментов. Программу, которая представляет собой один большой набор команд, не разделенный на смысловые части, трудно разрабатывать, понимать и улучшать, а программу, собранную из функций, — гораздо легче. Необходимо только, чтобы ваши собственные функции, как и стандартные, были оптимизированы для решения строго определенных задач.

Знакомство с программой «Инструкция»

Скриншоты игры «Крестики-нолики», показанные выше, не оставляют сомнений в том, что компьютерный противник ведет себя с изрядной долей высокомерия. Кичливость проглядывает уже в той краткой инструкции, которую компьютер показывает пользователю перед началом игры. Код, выводящий на экран эту инструкцию, я оформил отдельной программой, которая так и называется — «Инструкция». Код имеет несколько непривычный для вас вид, потому что для вывода текста на экран я создал функцию. Пробный запуск программы показан на рис. 6.4.

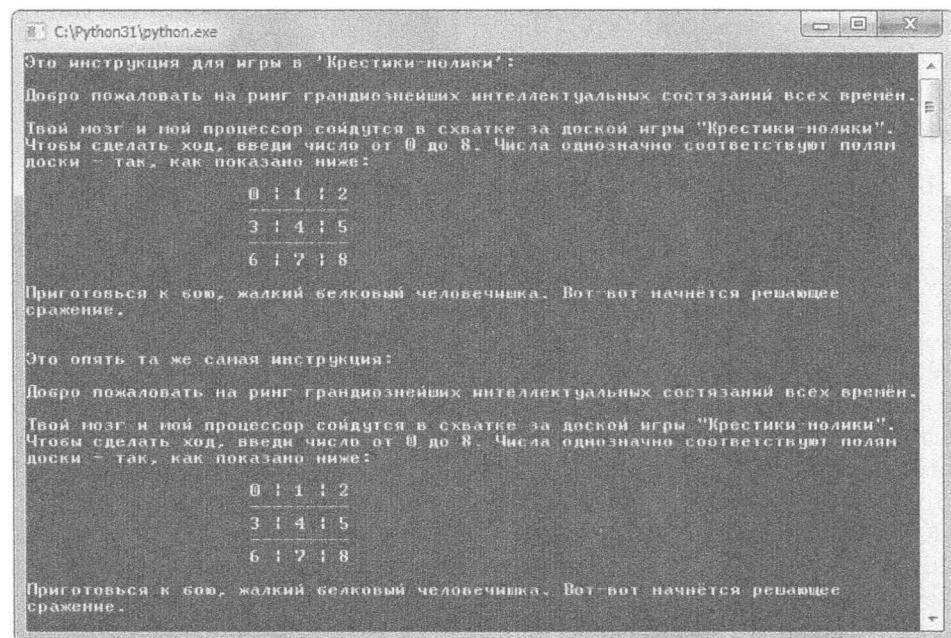


Рис. 6.4. Вывод инструкции на экран запускается всего одной строкой кода, которая вызывает нестандартную функцию

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 6. Файл называется `instructions.py`.

```

# Инструкция
# Демонстрирует, как создавать собственные функции
def instructions():
    """Выводит на экран инструкцию для игрока."""

```

```

print(
"""

Добро пожаловать на ринг грандиознейших интеллектуальных состязаний всех времен.
Твой мозг и мой процессор сойдутся в схватке за доской игры "Крестики-нолики".
Чтобы сделать ход, введи число от 0 до 8. Числа однозначно соответствуют полям
доски - так, как показано ниже:

0 | 1 | 2
-----
3 | 4 | 5
-----
6 | 7 | 8

Приготовься к бою, жалкий белковый человечишко. Вот-вот начнется решающее сражение.\n"""
)

# основная часть
print("Это инструкция для игры в 'Крестики-нолики':")
instructions()
print("Это опять та же самая инструкция:")
instructions()
print("Надеюсь, теперь смысл игры ясен.")
input("\n\nНажмите Enter, чтобы выйти.")

```

Объявление функции

Запись новой функции я начал с такой строки кода:

```
def instructions():
```

Эта строка сообщает интерпретатору, что следующий блок кода — функция `instructions()`. Будем называть его блоком выражений. Каждый раз, когда в программе будет вызвана функция `instructions()`, компьютер исполнит блок выражений. Стока с оператором `def` и блок выражений, вместе взятые, — это *объявление функции*. Хотя из него ясно, какую задачу решает функция, работу функции оно не запускает. Встретив объявление, интерпретатор лишь замечает, что есть такая функция и что в будущем ее можно применить. Эта функция не сработает до тех пор, пока не будет вызвана далее в программе.

Когда будете писать свои нестандартные функции, следуйте моему образцу. Начните с `def`, затем введите имя функции, пару круглых скобок, двоеточие и, наконец, блок выражений с отступом. В именах функций придерживайтесь тех же простых правил, что и в названиях переменных. Позаботьтесь также о том, чтобы имя отражало суть задачи, которую решает функция, или результаты ее работы.

Документирование функции

Есть особый механизм, пользуясь которым можно оставлять пояснения к функциям. Речь идет о так называемых *документирующих строках*. Функцию `instructions()` документирует такая строка:

```
"""Выводит на экран инструкцию для игрока."""
```

Документирующая строка в функциях представляет собой, как правило, строку в тройных кавычках. В блоке выражений она обязательно идет первой по порядку. Если вы решили документировать несложную функцию, то можете, как я, описать одним предложением, что именно делает эта функция. Если документирующей строки нет, на работе функции это не скажется. Однако лучше пусть она будет: так вы скорее привыкнете комментировать свой код и научитесь ясно формулировать задачи, которые ставите перед своими функциями. Интересно также, что в IDLE, когда вы будете набирать вызов своей функции, документирующая ее строка появится на экране в качестве подсказки.

Вызов нестандартной функции

Нестандартная функция вызывается точно так же, как и стандартная: указывается имя функции и пара скобок вслед за ним. Я вызвал свою новую функцию два раза с помощью одного и того же кода:

```
instructions()
```

Эта строка велит компьютеру вернуться к функции, объявленной ранее, и выполнить ее. При каждом вызове компьютер выводит игровую инструкцию на экран.

Что такое абстракция

Создавая и вызывая функции, вы прибегаете к так называемой *абстракции*. Абстракция позволяет сконцентрироваться на целом, не задерживаясь на мелких деталях. Например, в программе, разобранной выше, я применяю функцию `instructions()` и не беспокоюсь о том, как именно в ней реализован вывод текста на экран. Достаточно вызвать функцию одной строкой кода — и задача решена.

К абстракции люди прибегают в самых неожиданных и причудливых обстоятельствах. Рассмотрим такую ситуацию: один служащий фастфуда говорит другому, что минуту назад «обслужил заказ № 3». Как понимает адресат, говорящий имеет в виду, что он выслушал клиента, подошел к тепловому шкафу, достал оттуда гамбургер, перешел к фритюрнице, выбрал из нее большую порцию картофеля фри и упаковал, потом взял большой пластиковый стакан, наполнил его газированкой из резервуара, передал эту еду клиенту, взял у него деньги и вернул сдачу. Вдаваться в такие подробности было бы и скучно, и попросту бесполезно, ведь оба сотрудника понимают, что такое «обслужить заказ № 3». Вместо того чтобы заботиться о деталях, они прибегают к абстракции.

Параметры и возвращаемые значения

О встроенных функциях вы уже знаете то, что можно и передавать значения им, и получать значения, которые они возвращают. Так, функция `len()` принимает на вход последовательность и возвращает ее длину. Ваши собственные функции тоже могут принимать и возвращать значения, что дает им возможность взаимодействовать с прочими элементами программы.

Знакомство с программой «Принимай — возвращай»

В программе «Принимай — возвращай» я создал три функции, на примере которых вы познакомитесь с разными сочетаниями принимаемых и возвращаемых величин. Первая функция только принимает значение как свой аргумент, вторая функция лишь возвращает значение, а третья и принимает и возвращает. На рис. 6.5 можно видеть результат работы программы.

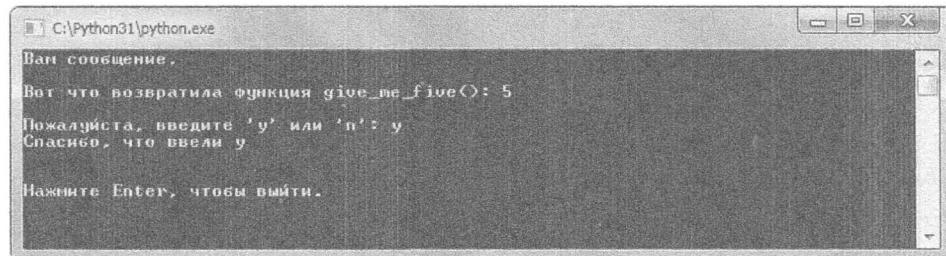


Рис. 6.5. Взаимодействовать с основной частью этой программы ее функциям позволяют параметры и возвращаемые значения

Код программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 6. Файл называется `receive_and_return.py`.

```
# Принимай - возвращай
# Демонстрирует параметры и возвращаемые значения
def display(message):
    print(message)
def give_me_five():
    five = 5
    return five
def ask_yes_no(question):
    """Задает вопрос с ответом 'да' или 'нет'."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
# основная часть
display("Вам сообщение.\n")
number = give_me_five()
print("Вот что возвратила функция give_me_five():", number)
answer = ask_yes_no("\nПожалуйста, введите 'y' или 'n': ")
print("Спасибо, что ввели", answer)
input("\n\nНажмите Enter, чтобы выйти.")
```

Передача данных с помощью параметров

Первая функция, которую я объявил, называется `display()`. Она принимает значение и выводит его на экран. Передачу значения функции осуществляет так назы-

ваемый *параметр*. Параметры — это, в сущности, имена переменных, заключаемые в скобки, которые следуют за именем функции:

```
def display(message):
```

Значения, которые программист желает передать функции, параметры берут из аргументов, содержащихся в вызове функции. Например, в моей программе при вызове `display()` переменной `message` присваивается значение строки "Вам сообщение.\n", потому что в основной части программы язываю `display()` строкой кода:

```
display("Вам сообщение.\n")
```

Когда в переменной `message` появляется значение, функция срабатывает.

Как и любой параметр, переменная `message` доступна внутри функции в качестве ее аргумента. Поэтому строка кода:

```
print(message)
```

выводит на экран "Вам сообщение".

Если бы я не передал значение параметру `message`, то интерпретатор выдал бы ошибку и сказал, что функция `display()` принимает ровно один аргумент.

Хотя у `display()` всего один параметр, у функции их может быть, вообще говоря, несколько. Чтобы задать функции несколько параметров, надо их перечислить в скобках через запятую.

Возврат значений функциями

Следующая функция-образец, которую я написал, — `give_me_five()` — только возвращает значение. Невероятно, но факт: возврат значения обеспечивает команда `return five`

Когда выполняется эта строка кода, функция передает значение переменной `five` обратно в строку кода, которая ее вызвала, и прекращает работу. После выполнения команды `return` работа функции всегда прекращается. Дальнейшее выполняет та часть программы, которая вызвала функцию и затребовала от нее значения, чтобы далее как-либо ими манипулировать. Вот фрагмент основного раздела программы, где вызывается наша функция:

```
number = give_me_five()
print("Вот что возвратила функция give_me_five():". number)
```

Результат работы вызванной функции, то есть значение, которое она возвращает, я присваиваю переменной `number`. Теперь после завершения работы функции в `number` остается числовое значение 5 — результат вызова `give_me_five()`. Чтобы убедиться, что с этим значением все в порядке, я вывожу его на экран следующей строкой кода.

Получать от функции можно и несколько значений. Достаточно перечислить их все после команды `return` через запятую.

ЛОВУШКА

Убедитесь, что все значения, которые возвращает функция, вы распаковываете в переменные. При попытке присвоить набору переменных набор значений, который отличается по численности, система выдаст ошибку.

Что такое инкапсуляция

Возможно, вам кажется нецелесообразным, чтобы ваши собственные функции возвращали значения. В самом деле, почему бы не пользоваться переменной `five` прямо в коде основной части программы? А потому, что нельзя — переменная `five` не существует вне блока выражений функции `give_me_five()`. Вообще ни одна переменная, созданная внутри функции (в том числе и параметры), непосредственно не доступна извне. Эта полезная техника называется **инкапсуляцией**. Она помогает сохранять независимость отдельных фрагментов кода; для этого прячутся (инкапсулируются) частности. Параметры и возвращаемые значения используются именно затем, чтобы передавать важную информацию и игнорировать прочую. Кроме того, за значениями переменных, созданных внутри функции, не приходится следить во всем остальном коде. Чем больше программа, тем значительнее выгода от этого.

Инкапсуляция, как она была здесь объяснена, похожа на абстракцию, и это не случайно. Инкапсуляция тесно связана с абстракцией и является собой ее важнейший элемент. Абстракция позволяет не заботиться о деталях, а инкапсуляция попросту прячет детали. Рассмотрим для примера пульт дистанционного управления телевизором, где за уровень звука отвечают кнопки «Больше» и «Меньше». Когда вы берете пульт ДУ и убавляете или прибавляете звук — это абстракция, так как вы не стремитесь знать, что именно происходит в это время внутри телевизора. Теперь предположим, что у телевизора десять уровней громкости. Пульт ДУ позволяет их все перебрать, но не дает прямого доступа к их выбору. Определенный уровень звука нельзя назначить, указав его номер. Надо нажимать на пульте «Больше» или «Меньше» до тех пор, пока искомая громкость не будет достигнута. Таким образом, порядковый номер уровня громкости инкапсулирован.

ПОДСКАЗКА

Не огорчайтесь, если пока не уловили тонкую разницу между абстракцией и инкапсуляцией. Это понятия родственные, и зачастую границу между ними провести нельзя. Вы снова встретитесь с инкапсуляцией и абстракцией в главах 8 и 9, когда пойдет речь об объектах и объектно-ориентированном программировании.

Функции, которые и принимают и возвращают значения

Последняя из написанных мной функций — `ask_yes_no()` — принимает одно значение и возвращает другое. Она принимает вопрос компьютера пользователю и возвращает пользовательский ввод: букву "у" или "н". У функции один параметр — вопрос `question`:

```
def ask_yes_no(question)
```

В переменную `question` попадает значение аргумента, переданного функции. В данном случае аргумент — строка "\nПожалуйста, введите 'у' или 'н': ". Дальнейший код печатает эту строку на экране и просит пользователя дать ответ:

```
response = None
while response not in ("y", "n"):
    response = input(question).lower()
```

Благодаря циклу `while` вопрос будет задаваться снова и снова до тех пор, пока пользователь не введет букву `y`, `Y`, `n` или `N`. Символы пользовательского ввода всегда преобразуются в нижний регистр.

Когда пользователь наконец введет корректную букву, функция отошлет ее в виде строки обратно в основную часть программы, откуда осуществляется вызов:

```
return response
```

После этого работа функции будет остановлена.

В основной части результата работы функции присваивается переменной `answer` и выводится на экран:

```
answer = ask_yes_no("\nПожалуйста, введите 'y' или 'n': ")
print("Спасибо, что ввели", answer)
```

Что такое повторное использование кода

У функций есть еще одна сильная сторона: их можно с легкостью применять во многих программах. Вот, скажем, «да/нет»-вопрос пользователю — очень типичный элемент текстового меню. Вы можете взять готовую функцию `ask_yes_no()` и пользоваться ею везде, где это необходимо, вместо того чтобы заново создавать аналогичный код. Это так называемое *повторное использование кода*. Итак, пишите качественные функции: это сбережет вам время и усилия в работе не только над нынешним проектом, но и над несколькими будущими!

НА САМОМ ДЕЛЕ

На то, чтобы каждый раз заново «изобретать колесо», может уйти слишком много труда. Вот почему в промышленной разработке ПО так широко практикуется повторное использование кода, при котором существующие разработки и их отдельные части кочуют из старых проектов в новые. Повторное использование способно сделать следующее.

- Повысить производительность фирмы. На проект, в состав которого будут входить уже существующий код и другие готовые элементы, компания может затратить меньше усилий.
- Улучшить качество программ. Если какой-либо фрагмент кода тщательно протестирован, в дальнейшем его можно повторно использовать с уверенностью в том, что он не содержит ошибок.
- Обеспечить единообразие программных продуктов. Так, например, если фирма оснастит свой новый продукт старым пользовательским интерфейсом, то клиенту не надо будет переучиваться.
- Увеличить эффективность ПО. Если известен хороший способ решения какой-либо задачи, то применять его стоит, чтобы не изобретать колесо или случайно не придумать менее эффективное колесо, чем уже имеющееся.

Один способ повторного использования функций — копировать их в новую программу из старой. Но есть лучший способ: создавать собственные модули и из них загружать функции в новые программы точно так же, как импортируются функции из стандартных модулей Python. Вы научитесь создавать модули и пользоваться ими в одном из разделов главы 9.

Именованные аргументы и значения параметров по умолчанию

Передача значений из аргументов в параметры снабжает функцию данными. Но я рассказал пока лишь о самом примитивном способе такой передачи данных. Гораздо больше гибкости и мощи обеспечат механизмы Python, новые для вас: значения параметров по умолчанию и именованные аргументы.

Знакомство с программой «День рождения»

В программе «День рождения», пробный запуск которой показан на рис. 6.6, для поздравлений созданы две очень похожие функции. В первой из них используется уже знакомый вам тип параметров — так называемые *позиционные параметры*, а во второй применены *значения параметров по умолчанию*. Разницу лучше всего увидеть в деле.

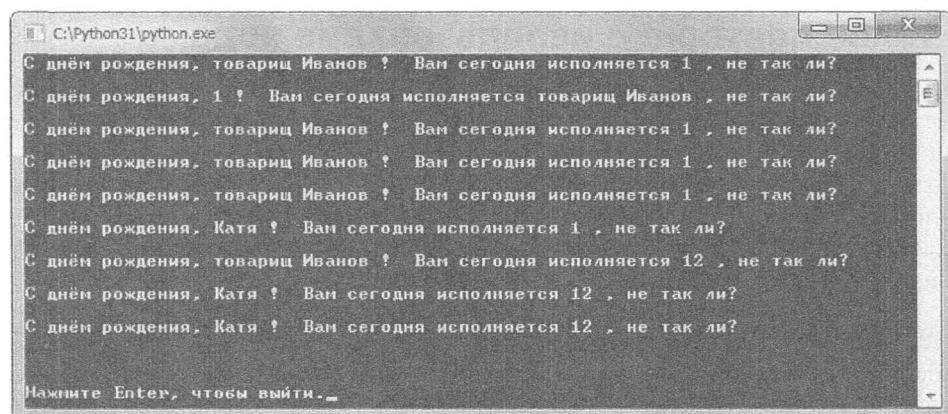


Рис. 6.6. Именованные аргументы и значения параметров по умолчанию дают много гибкости в вызове функций

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 6. Файл называется `birthday_wishes.py`.

```
# День рождения
# Демонстрирует именованные аргументы и значения параметров по умолчанию
# позиционные параметры
def birthday1(name, age):
    print("С днем рождения.", name, "!", " Вам сегодня исполняется", age, ". не так ли?\n")
# параметры со значениями по умолчанию
def birthday2(name = "товарищ Иванов", age = 1):
    print("С днем рождения.", name, "!", " Вам сегодня исполняется", age, ". не так ли?\n")
birthday1("товарищ Иванов", 1)
birthday1(1, "товарищ Иванов")
```

```

birthday1(name = "товарищ Иванов", age = 1)
birthday1(age = 1, name = "товарищ Иванов")
birthday2()
birthday2(name = "Катя")
birthday2(age = 12)
birthday2(name = "Катя", age = 12)
birthday2("Катя", 12)
input("\n\nНажмите Enter, чтобы выйти.")

```

Позиционные параметры и позиционные аргументы

Если, объявляя функцию, просто перечислить имена переменных, будут созданы позиционные параметры:

```
def birthday1(name, age):
```

Соответственно, когда в вызове такой функции просто перечислены значения, они являются позиционными аргументами:

```
birthday1("товарищ Иванов", 1)
```

Это говорит о том, что значения параметров определяются исходя только из порядка, в котором перечислены аргументы. Первый параметр принимает такое же значение, как первый аргумент, второй параметр — как второй аргумент и т. д. В показанном здесь вызове функции параметр name принимает значение "товарищ Иванов", а параметр age — значение 1. Функция печатает на экране: С днем рождения, товарищ Иванов ! Вам сегодня исполняется 1 , не так ли?.

Если поменять аргументы местами, то значения в параметрах тоже поменяются. Так, вызов:

```
birthday1(1, "товарищ Иванов")
```

передаст параметру name значение 1, а параметру age — "товарищ Иванов". В результате поздравление будет выглядеть несколько неожиданно: С днем рождения, 1 ! Вам сегодня исполняется товарищ Иванов . не так ли?.

Этот способ создания и вызова функций вам уже знаком. Но списки параметров и аргументов в ваших программах могут принимать и иной вид.

Позиционные параметры и именованные аргументы

Позиционные параметры принимают значения в том порядке, в котором те переданы. Но функциям можно велеть присваивать определенные значения определенным параметрам, невзирая на порядок. Таковы именованные аргументы. С помощью этого механизма имена параметров, объявленные в заголовке функции, можно произвольным образом связать со значениями. Если в вызове функции birthday1() написать:

```
birthday1(name = "товарищ Иванов", age = 1).
```

то в параметре name окажется "товарищ Иванов", а в age — 1 и функция отобразит нужный нам текст: С днем рождения, товарищ Иванов ! Вам сегодня исполняется 1 . не так ли?. Это пока не выглядит впечатляюще, потому что того же самого можно добиться и без именованных аргументов, просто передавая значения в известном порядке. Но изящество именованных аргументов состоит в том, что для них порядок неважен; значения присваиваются параметрам в той последовательности, которую укажет программист. Вызов:

```
birthday1(age = 1, name = "товарищ Иванов")
```

тоже выведет текст С днем рождения, товарищ Иванов ! Вам сегодня исполняется 1 . не так ли?, хотя порядок значений здесь иной.

Именованные аргументы позволяют передавать значения в любом порядке. Но главное их достоинство — ясность. Когда вызов функции содержит имена параметров, становится гораздо понятнее, какое из значений чему соответствует.

ЛОВУШКА

В одном вызове функции можно сочетать позиционные и именованные аргументы, но от этого возникают некоторые трудности. Если где-либо в вызове функции вы указали именованный аргумент, то все следующие за ним аргументы тоже должны быть именованы. Чтобы обойтись без лишних усложнений, старайтесь применять или только позиционные, или только именованные аргументы.

Значения параметров по умолчанию

Параметрам можно, хотя и не обязательно, присваивать значения по умолчанию. Эти значения будут передаваться им в том случае, если вызов функции не содержит соответствующих аргументов. Именно таким образом я реализовал функцию birthday2(), которая отличается от birthday1() только заголовком:

```
def birthday2(name = "товарищ Иванов", age = 1):
```

Итак, если параметру name не будет передано никакое значение, он будет содержать строку "товарищ Иванов", а если не указать значение age, то здесь окажется единица. Поэтому вызов:

```
birthday2()
```

не влечет ошибок. Параметрам просто присваиваются значения по умолчанию, и функция выводит на экран: С днем рождения, товарищ Иванов ! Вам сегодня исполняется 1 . не так ли?.

ЛОВУШКА

Если присвоить значение по умолчанию одному из параметров в списке, то надо будет проделать то же и со всеми следующими параметрами. Вот такой заголовок функции совершенно корректен:

```
def monkey_around(bananas = 100, barrel_of = "yes", uncle = "monkey's"):
```

А вот такой — нет:

```
def monkey_around(bananas = 100, barrel_of, uncle):
```

Прочитав его, интерпретатор выдаст ошибку.

Пока ничего сложного, да? Нарушает эту симпатичную простоту возможность заменить значение по умолчанию любого из параметров каким-либо иным значением. Вызов:

```
birthday2(name = "Катя")
```

переопределяет значение параметра name, в котором теперь оказывается "Катя". А вот age по-прежнему содержит 1. На экране появится сообщение: С днем рождения, Катя ! Вам сегодня исполняется 1 . не так ли?.

В следующем вызове функции:

```
birthday2(age = 12)
```

переопределено значение по умолчанию параметра age, который теперь равен 12, тогда как name по-прежнему равен строке "товарищ Иванов". На экране появится сообщение: С днем рождения, товарищ Иванов ! Вам сегодня исполняется 12 . не так ли?.

Вызов:

```
birthday2(name = "Катя", age = 12)
```

переопределяет оба значения по умолчанию: name становится равным "Катя", а age — числу 12. Выводится текст: С днем рождения, Катя ! Вам сегодня исполняется 12 . не так ли?.

Наконец, вызов:

```
birthday2("Катя", 12)
```

действует подобно предыдущему. Оба значения по умолчанию заменяются соответственно значениями "Катя" и 12, и программа выводит уже знакомый текст: С днем рождения, Катя ! Вам сегодня исполняется 12 . не так ли?.

ХИТРОСТЬ

Значения параметров по умолчанию удобны тогда, когда в большинстве вызовов одному из параметров функции передается одна и та же величина. Чтобы освободить тех программистов, которые будут пользоваться вашей функцией, от набора лишнего, избыточного текста, вы можете применять значения по умолчанию.

Использование глобальных переменных и констант

Благодаря инкапсуляции те функции, которые мы теперь создаем, как бы «запечатаны», обособлены друг от друга и от основной части программы. Единственный способ передать им информацию — использовать параметры, единственный механизм извлечения информации из них — указание возвращаемых значений, думаете вы. Однако это не совсем верно. Есть еще один путь передачи информации из одной части вашей программы в другую: с помощью глобальных переменных.

Что такое области видимости

Области видимости — это способ представления разных частей программы, отделенных друг от друга. Так, например, у каждой объявленной нами функции своя

область видимости. Вот почему эти функции не имеют доступа к переменным друг друга. Понять, о чем идет речь, вам поможет наглядное пособие (рис. 6.7).

```
def func1():
    variable1
def func2():
    variable2
variable0
```

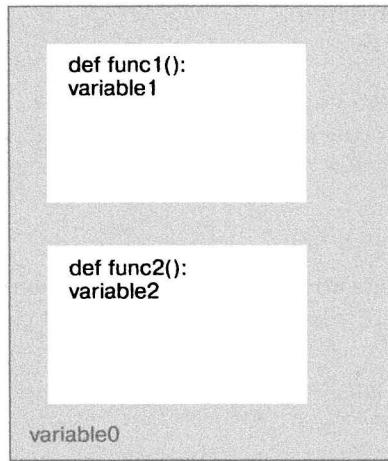


Рис. 6.7. В этой простой программе три области видимости: по одной для каждой функции и еще одна — глобальная

На рис. 6.7 показана программа с тремя областями видимости. Первая область видимости связана с функцией `func1()`, вторая — с функцией `func2()`, а третья — глобальная, которая автоматически задается в любой программе. В нашем примере глобальная область видимости — это все, что вовне функций; на рисунке она обозначена затемненным участком. Любая переменная, которую вы создадите в глобальной области видимости, называется *глобальной*, а переменная, созданная внутри функции, — *локальной* (она лежит в области видимости данной функции).

Поскольку переменная `variable1` задана внутри функции `func1()`, она локальна и доступна лишь из области видимости `func1()`, но не из всех остальных. Никакое из выражений, заключенных в `func2()` или в глобальной области видимости, не может обращаться к переменной `variable1`, изменять ее значение и т. д.

Запомнить это вам поможет такая метафора. Представьте себе, что области видимости — это коттеджи, а инкапсуляция — затемненные окна, которыми хозяева защищаются от вмешательства в их частную жизнь. Пребывая внутри дома, вы видите, что происходит снаружи. Но вот если выйти наружу, происходящее внутри становится недоступным для глаз¹. Так и с функциями. Пока вы «внутри» функци-

¹ Эта метафора на самом деле крайне забавна, так как, выходя из коттеджа, вы убиваете всех, кто находился внутри на тот момент, когда вы в него зашли. Правда, войдя обратно, вы обнаружите тех же людей, живых и невредимых, которые ведут себя так, будто вы сюда не входили. — *Примеч. науч. ред.*

ции, вы имеете доступ ко всем ее переменным, а выйдя «вовне», например в глобальную область видимости, вы теряете доступ.

Если две переменные в двух разных функциях названы одинаково, они останутся совершенно отдельными и никак не будут связаны между собой. К примеру, если бы я создал переменную `variable2` внутри функции `func1()`, это было бы вовсе не то же самое, что переменная `variable2` в области видимости `func2()`. Ввиду инкапсуляции они будут существовать как будто в разных мирах и никак не сказываться друг на друге. Впрочем, как видите, маленькую непоследовательность в инкапсуляции создают глобальные переменные.

Знакомство с программой «Доступ отовсюду»

Программа «Доступ отовсюду» показывает, как читать и даже менять значения глобальных переменных внутри функций. Результаты работы программы отражены на рис. 6.8.

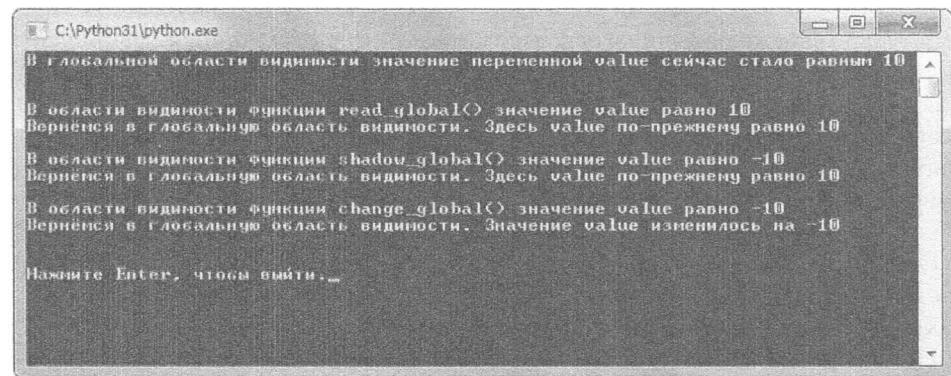


Рис. 6.8. Значение глобальной переменной можно прочесть, «затенить» и даже изменить внутри функции

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 6. Файл называется `global_reach.py`.

```
# Доступ отовсюду
# Демонстрирует работу с глобальными переменными
def read_global():
    print("В области видимости функции read_global() значение value равно",
          value)
def shadow_global():
    value = -10
    print("В области видимости функции shadow_global() значение value равно",
          value)
def change_global():
    global value
    value = -10
    print("В области видимости функции change_global() значение value равно",
          value)
# основная часть
```

```
# value - глобальная переменная, потому что сейчас мы находимся в глобальной области видимости
value = 10
print("В глобальной области видимости значение переменной value сейчас стало равным",
      value, "\n")
read_global()
print("Вернемся в глобальную область видимости. Здесь value по-прежнему равно",
      value, "\n")
shadow_global()
print("Вернемся в глобальную область видимости. Здесь value по-прежнему равно",
      value, "\n")
change_global()
print("Вернемся в глобальную область видимости. Значение value изменилось на",
      value)
input("\n\nНажмите Enter, чтобы выйти.")
```

Чтение глобальной переменной внутри функции

Хотя теперь вы уже наверняка освоились с идеей инкапсуляции, я решил еще немного вас озадачить: значение глобальной переменной можно прочесть отовсюду, из любой области видимости. Не думайте, однако, будто аналогия с домами и тонированными стеклами здесь не проходит. Как вы помните, затемненные окна хранят тайну частной жизни дома (то есть функции). Но вместе с тем они позволяют видеть происходящее снаружи. Вот почему из функции всегда можно «выглянуть» в глобальную область видимости, чтобы узнать значение глобальной переменной. Это я и сделал в созданной мной функции `read_global()`. Она без труда печатает на экране значение глобальной переменной `value`.

Хотя любая функция имеет доступ к глобальным переменным, это доступ только на чтение; непосредственно поменять значение глобальной переменной функция не сможет (во всяком случае, если не запросит специальных полномочий). Так, если бы в `read_global()` мы попробовали написать что-нибудь такое:

```
value += 1
```

то случилась бы неприятная ошибка.

Возвращаясь к жилищной аналогии, поясню, что глобальную переменную можно «увидеть» из функции через «затемненное окно», но нельзя «потрогать», ведь она находится снаружи. Итак, значение глобальной переменной внутри функции может быть прочитано, но не изменено, если не включать особый режим доступа.

Затенение глобальной переменной внутри функции

Если переменной внутри функции вы присваиваете то же самое имя, что и глобальной переменной, это называется «затенять» глобальную переменную — скрывать ее за одноименной локальной. Может показаться, что таким образом можно изменить значение глобальной переменной, но в действительности меняется

только ее локальная тезка. Так работает моя функция `shadow_global()`. Когда я коммандой:

```
value = -10
```

присвоил `value` значение `-10`, я не изменил значение `value` глобально. Напротив, я лишь создал локальную версию `value` внутри функции и уже ей присвоил `-10`. Удостовериться, что все случилось именно так, мы можем из того, что после завершения работы функции команда:

```
print("Вернемся в глобальную область видимости. Здесь value по-прежнему равно".
      value, "\n")
```

в основной части программы выводит на экран глобальное значение `value`, которое оказывается по-прежнему равно `10`.

ЛОВУШКА

Вредно затенять глобальные значения внутри функции. Это может привести к погрешности: вам будет казаться, что работает глобальная переменная, а на самом деле это будет локальная. При сматривайте за всеми глобальными переменными в своей программе и не пользуйтесь их именами для обозначения посторонних сущностей.

Изменение глобальной переменной внутри функции

Полный доступ к глобальной переменной дает служебное слово `global`, которым я воспользовался в функции `change_global()`:

```
global value
```

Теперь функции доступно значение переменной `value` в том числе и для перезаписи. Я изменил его коммандой:

```
value = -10
```

Теперь глобальная переменная `value` равна `-10`. Когда в основной части кода значение `value` вновь выводится на экран коммандой:

```
print("Вернемся в глобальную область видимости. Значение value изменилось на".
      value).
```

печатается `-10`. Так мы смогли изменить значение глобальной переменной изнутри самой функции.

Когда использовать глобальные переменные и константы

«Могу» не значит «должен» — неплохой девиз для программиста, как мне кажется. Некоторые вещи технически осуществимы, но тратить время на них, в принципе, незачем. Одна из таких вещей — использование глобальных переменных. Они, вообще говоря, запутывают код, потому что за их меняющимися значениями иногда трудно следить. Постарайтесь ограничить их применение, насколько возможно.

С другой стороны, глобальные константы (переменные, значения которых вы решили не менять) не запутывают, а только проясняют структуру программы. Вообразим, например, что вы пишете бизнес-приложение для подсчета суммы налогов, которые кто-либо должен уплатить. Как любой толковый программист в такой ситуации, вы, конечно, создадите несколько функций для разных задач. И что же, неужели в каждой функции величина налога будет вычисляться исходя из загадочной величины .28 (процентной ставки)? Лучше, конечно, создать глобальную константу TAX_RATE и присвоить ей значение .28, а в каждой из функций вместо .28 писать TAX_RATE. Выгода от этого двоякая: код станет, во-первых, более понятным, а во-вторых, более удобным для исправлений. Если, например, ставка налога изменится, то достаточно будет поправить одну строку кода.

Вернемся к игре «Крестики-нолики»

Игра «Крестики-нолики», кратко описанная в начале этой главы, — пока самый амбициозный из проектов, в которых вам приходилось участвовать. У вас уже достаточно мастерства, чтобы разработать такую игру, но не спешите переходить прямо к разбору кода. Сначала я расскажу о том, какой план лежал в основе работы над программой. Отсюда вы сумеете почерпнуть некоторое понятие о разработке больших приложений, да и просто составить целостное впечатление об игре.

План программы «Крестики-нолики»

Не устану повторять вам, если вы еще не запомнили: чтобы написать хорошую программу, надо прежде хорошо спланировать свой труд. Без карты автодорог вы не сможете добраться в незнакомый город или доберетесь туда, но длинным окружным путем. Эта аналогия применима и к труду программиста.

Написание псевдокода

Вот опять этот язык, который на самом деле даже не язык: псевдокод. Поскольку за решение большинства задач программы будут отвечать функции, я могу позволить себе думать о программе на весьма общем уровне. Каждая строка псевдокода должна соответствовать одному вызову функции. Впоследствии мне придется лишь реализовать те функции, которые заложены в план.

Вот псевдокод:

```
вывести на экран инструкции для игрока
решить, кому принадлежит первый ход
создать пустую доску для игры в "Крестики-нолики"
отобразить эту доску
до тех пор пока никто не выиграл или не состоялась ничья
    если сейчас ход пользователя
        получить ход из пользовательского ввода
        изменить вид доски
```

иначе

 рассчитать ход компьютера

 изменить вид доски

 вывести на экран обновленный вид доски

 осуществить переход хода

поздравить победителя или констатировать ничью

Представление данных

Итак, у меня отличный план, но он чересчур абстрактен и оперирует такими вещами, сущность которых пока неясна даже мне самому. Я понимаю, например, что ход нужно рассматривать как помещение фишki на одну из клеток игровой доски. Но как именно должны быть представлены доска, фишка и, соответственно, ход?

Поскольку я намерен выводить игровую доску на экран, почему бы не рассматривать фишку как один символ: "X" или "0"? Пустому полю может соответствовать пробел. Саму доску следует представить списком, так как после каждого хода одного из игроков какой-либо из ее элементов меняется. На доске для игры в крестики-нолики девять полей, поэтому мы создадим список из девяти элементов. Каждому полю будет отвечать порядковый номер в списке, которым представлена доска. Наглядно все показано на рис. 6.9.

0	1	2
3	4	5
6	7	8

Рис. 6.9. Порядковые номера полей совпадают с номерами позиций в списке, которым представлена доска

Как видно, каждому полю (позиции на игровой доске) отвечает число от 0 до 8. Соответственно, список состоит из девяти элементов с номерами позиций от 0 до 8. Поскольку ход — это всего лишь указание на поле, в которое игрок желает поместить свою фишку, то и его также целесообразно представить числом от 0 до 8. Род «оружия» каждого из противников (крестики или нолики) можно обозначить теми же символами "X" и "0", что и фишки. Таким образом, переменная, определяющая, чей сейчас ход, будет содержать или "X", или "0".

Создание списка функций

Из псевдокода непосредственно следует список функций, в которых нуждается программа. Чтобы разработать такой список, я задумался о том, что каждая из функций должна делать, какие параметры принимать и какие значения возвращать. Результат моих раздумий приведен в табл. 6.1.

Таблица 6.1. Функции игры «Крестики-нолики»

Функция	Описание
display_instruct()	Выводит инструкцию для игрока
ask_yes_no(question)	Задает вопрос, ответом на который может быть «Да» или «Нет». Принимает текст вопроса, возвращает "у" или "н"
ask_number(question, low, high())	Просит ввести число из указанного диапазона. Принимает текст вопроса, нижнюю (low) и верхнюю (high) границы диапазона. Возвращает целое число не меньше low и не больше high
pieces()	Определяет принадлежность первого хода человеку или компьютеру. Возвращает типы фишек соответственно компьютера и человека
new_board()	Создает пустую игровую доску. Возвращает эту доску
display_board(board)	Отображает игровую доску на экране. Принимает эту доску
legal_moves(board)	Создает список доступных ходов. Принимает доску. Возвращает список доступных ходов
winner(board)	Определяет победителя игры. Принимает доску. Возвращает тип фишек победителя: "Ничья" или None
human_move(board, human)	Узнает, какой ход желает совершить игрок. Принимает доску и тип фишек человека. Возвращает ход человека
computer_move(board, computer, human)	Рассчитывает ход компьютерного противника. Принимает доску, тип фишек компьютера и тип фишек человека. Возвращает ход компьютера
next_turn(turn)	Осуществляет переход к следующему ходу. Принимает тип фишек. Возвращает тип фишек
congrat_winner(the_winner, computer, human)	Поздравляет победителя или констатирует ничью. Принимает тип фишек победителя, тип фишек компьютера и тип фишек человека

Настройка программы

Приступая к написанию программы, я первым делом назначил несколько глобальных констант. Эти значения будут использоваться в нескольких функциях. Если создать их сейчас, код функций станет яснее и вносить изменения будет проще.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 6. Файл называется `tic-tac-toe.py`.

```
# Крестики-нолики
# Компьютер играет в крестики-нолики против пользователя
# глобальные константы
X = "X"
O = "O"
EMPTY = " "
TIE = "Ничья"
NUM_SQUARES = 9
```

X — сокращенное обозначение для крестика ("X"), O — для нолика ("O"). EMPTY представляет пустое поле на игровой доске. В качестве значения этой константы выбран пробел, потому что на экране он выглядит как пустой прямоугольник. TIE представляет состояние ничьей. NUM_SQUARES — количество полей на доске «Крестиков-ноликов».

Функция `display_instruct()`

Эта функция выводит на экран инструкцию для игрока. Она вам уже знакома:

```
def display_instruct():
    """Выводит на экран инструкцию для игрока."""
    print(
        """
        Добро пожаловать на ринг грандиознейших интеллектуальных состязаний всех времен.
        Твой мозг и мой процессор сойдутся в схватке за доской игры "Крестики-нолики".
        Чтобы сделать ход, введи число от 0 до 8. Числа однозначно соответствуют полям
        доски - так, как показано ниже:
        0 | 1 | 2
        -----
        3 | 4 | 5
        -----
        6 | 7 | 8
        Приготовься к бою, жалкий белковый человечишко. Вот-вот начнется решающее сражение.\n"""
    )
```

Добро пожаловать на ринг грандиознейших интеллектуальных состязаний всех времен.

Твой мозг и мой процессор сойдутся в схватке за доской игры "Крестики-нолики".

Чтобы сделать ход, введи число от 0 до 8. Числа однозначно соответствуют полям доски - так, как показано ниже:

0 | 1 | 2

3 | 4 | 5

6 | 7 | 8

Приготовься к бою, жалкий белковый человечишко. Вот-вот начнется решающее сражение.\n

)

Я изменил только имя функции ради большего единобразия программы.

Функция `ask_yes_no()`

Функция задает вопрос, ответом на который должно быть «Да» или «Нет». Она принимает текст вопроса и возвращает букву "y" или "n". Эту функцию вы тоже уже видели.

```
def ask_yes_no(question):
    """Задает вопрос с ответом 'Да' или 'Нет'."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
```

Функция `ask_number()`

Эта функция запрашивает о вводе числа, лежащего в заданном диапазоне. Она принимает текст вопроса, нижнюю и верхнюю границы диапазона, а возвращает число из этого диапазона.

```
def ask_number(question, low, high):
    """Просит ввести число из диапазона."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response
```

Функция pieces()

Эта функция спрашивает пользователя, желает ли тот оставить первый ход за собой, и, исходя из его выбора, определяет типы фишек компьютера и человека. В соответствии со старинной традицией первый ход принадлежит играющему крестиками.

```
def pieces():
    """Определяет принадлежность первого хода."""
    go_first = ask_yes_no("Хочешь оставить за собой первый ход? (y/n): ")
    if go_first == "y":
        print("\nНу что ж, даю тебе фору: играй крестиками.")
        human = X      computer = O
    else:
        print("\nТвоя удача тебя погубит... Буду начинать я.")
        computer = X      human = O
    return computer, human
```

Заметьте, что внутри этой функции вызывается другая моя функция — `ask_yes_no()`. Это отнюдь не запрещено: одна функция может вызывать другую.

Функция new_board()

Функция создает новую игровую доску — список из девяти элементов, равных `EMPTY`, и возвращает ее:

```
def new_board():
    """Создает новую игровую доску."""
    board = []
    for square in range(NUM_SQUARES):
        board.append(EMPTY)
    return board
```

Функция display_board()

Эта функция выводит на экран переданную ей доску. Поскольку элементы доски — это пробелы, буквы "X" и "O", то их можно просто печатать функцией `print`. Чтобы нарисовать пристойного вида доску для «Крестиков-ноликов», я использовал еще некоторые символы.

```
def display_board(board):
    """Отображает игровую доску на экране."""
    print("\n\t", board[0], "|", board[1], "|", board[2])
    print("\t-----")
    print("\t", board[3], "|", board[4], "|", board[5])
    print("\t-----")
    print("\t", board[6], "|", board[7], "|", board[8], "\n")
```

Функция legal_moves()

Эта функция принимает доску и возвращает список доступных ходов. Ее будут вызывать другие функции нашей программы: `human_move()` — чтобы убедиться, что

человек намерен поместить свою фишку на одно из свободных полей, computer_move() — чтобы ограничить перебор альтернатив компьютером.

Каждый из разрешенных ходов — это порядковый номер пустого поля. Так, например, пока центральное из девяти полей не занято, 4 — доступный ход. Если бы были свободны только угловые поля, то список доступных ходов имел бы вид [0, 2, 6, 8] (если не совсем понятно, вернитесь к рис. 6.9). Таким образом, функция просто перебирает список, которым представлена доска. Каждый раз, обнаруживая пустое поле, она добавляет его номер в список разрешенных ходов. После окончания цикла возвращается этот список.

```
def legal_moves(board):
    """Создает список доступных ходов."""
    moves = []
    for square in range(NUM_SQUARES):
        if board[square] == EMPTY:
            moves.append(square)
    return moves
```

Функция winner()

Эта функция принимает доску и возвращает победителя игры. Может быть возвращено одно из четырех значений: X или 0 — если один из игроков побеждает; TIE — если все поля заполнены, но победы не достиг ни один из противников; наконец, None — если хотя бы одно из полей пусто и при этом победитель не определен.

Первым делом в этой функции я задал константу WAYS_TO_WIN, описывающую все восемь возможных способов поместить три фишки в ряд. Каждый способ достичь победы изображается кортежем — тройкой таких позиций на доске, что фишки на соответствующих полях образуют ряд. Возьмем для примера первый кортеж: (0, 1, 2). Он представляет верхний ряд, поля с позициями номер 0, 1 и 2. Следующий кортеж — (3, 4, 5) — представляет средний ряд и т. д.

```
def winner(board):
    """Определяет победителя в игре."""
    WAYS_TO_WIN = ((0, 1, 2),
                   (3, 4, 5),
                   (6, 7, 8),
                   (0, 3, 6),
                   (1, 4, 7),
                   (2, 5, 8),
                   (0, 4, 8),
                   (2, 4, 6))
```

Затем цикл for перебирает все способы достижения победы, чтобы узнать, нет ли уже у одного из игроков трех фишек в ряд. Применяемая здесь конструкция if для каждой тройки полей проверяет, не пусты ли они и одинаковое ли в них значение. Если все именно так, то, значит, тройка содержит ровно три крестика или нолика и один из игроков одержал верх. Одну из фишек выигравшего ряда функция делает значением переменной winner, возвращает эту переменную и останавливается.

```

for row in WAYS_TO_WIN:
    if board[row[0]] == board[row[1]] == board[row[2]] != EMPTY:
        winner = board[row[0]]
        return winner

```

Если ни один игрок не победил, работа функции продолжается. Теперь надо проверить, не осталось ли пустых полей на доске. Если нет, то игра сыграна вничью (потому что перебор в цикле for не выявил победителя) и возвращается значение TIE.

```

if EMPTY not in board:
    return TIE

```

В противном случае работа функции продолжается. Если победитель не определен, но вместе с тем игра не сыграна вничью, то, значит, конец игры еще не достигнут и функция возвращает None.

```
return None
```

Функция `human_move()`

Эта функция принимает доску и тип фишек игрока-человека, а возвращает номер поля, на которое игрок хочет поместить свою фишку.

Первым делом функция рассчитывает список доступных ходов. Затем до тех пор, пока пользователь не введет одно из чисел этого списка, функция продолжает спрашивать его, какое из свободных полей он желает занять. Получив корректный ответ, функция возвращает ход человека.

```

def human_move(board, human):
    """Получает ход человека."""
    legal = legal_moves(board)
    move = None
    while move not in legal:
        move = ask_number("Твой ход. Выбери одно из полей (0 - 8):", 0, NUM_SQUARES)
        if move not in legal:
            print("\nСмешной человек! Это поле уже занято. Выбери другое.\n")
    print("Ладно...")
    return move

```

Функция `computer_move()`

Функция `computer_move()` принимает игровую доску, тип фишек компьютера и тип фишек человека, а возвращает ход компьютера.

ХИТРОСТЬ —

Это, пожалуй, самая емкая и интересная функция во всей программе. Предполагая, что так и будет, я сперва создал маленькую временную функцию, которая делала случайный выбор из списка доступных ходов. Чтобы обдумать стратегию компьютерного противника, мне нужно было время, но я не хотел замедлять разработку проекта. Вот почему я сначала довел игру до рабочего состояния, а потом уже вернулся к расчету ответного хода и написал функцию, которая делает более или менее аргументированный выбор.

Такой гибкостью программа обязана своему «модульному» устройству, построению из функций. Я понимал, что `computer_move()` — совершенно независимый компонент, который можно впоследствии спокойно заменить другим. Даже и сейчас я мог бы создать вместо существующей функции новую, которая вычисляла бы ходы еще эффективнее. (Кстати, вот вам первая задача. Подумайте над ней на досуге.)

С списком, которым в программе представлена игровая доска, надо быть осторожным. Список — изменяемый тип, и функция поиска лучшего хода компьютера может менять элементы этого списка. Проблема в том, что любое изменение, которое внесет эта функция, будет актуально и для остальной части программы. Этот нежелательный эффект вызван распределенными ссылками, о которых вы узнали в разделе «Распределенные ссылки» главы 5. Сначала есть только один экземпляр списка (игровой доски), и любые изменения в любой из переменных, ссылающихся на этот экземпляр, отразятся в нем. Поэтому первым на очереди для меня было создать локальную копию доски, с которой можно манипулировать, ничего не опасаясь:

```
def computer_move(board, computer, human):
    """Делает ход за компьютерного противника."""
    # создадим рабочую копию доски, потому что функция будет менять некоторые значения в списке
    board = board[:]
```

ПОДСКАЗКА

Каждый раз, передавая функции изменяемое значение, действуйте с осторожностью. Если внутри вашей функции значение будет меняться, лучше создать копию и изменять ее.

ЛОВУШКА

Возможно, вам кажется удобным экспериментировать с расстановками на самой игровой доске («первом экземпляре» списка), а не ее копии, ведь тогда можно будет не возвращать никаких значений, а просто поменять вид доски так, чтобы на ней отразился очередной ход компьютера. Такое (непосредственное) изменение значений параметров считается побочным эффектом. Не все побочные эффекты плохи, но конкретно этот программисты (среди них и я), мягко говоря, недолюбливают. Гораздо лучше, чтобы ваши функции взаимодействовали с остальной частью программы через возвращаемые значения. В этом случае совершенно ясно, какой информацией обмениваются блоки кода.

Для компьютерного противника я придумал следующую стратегию действий.

- Если существует такой ход компьютера, после которого тот одерживает победу, надо сделать этот ход.
- Если существует такой ход человека, после которого тот может одержать победу на следующем ходу, надо предупредить этот ход.
- В ином случае компьютер выбирает для своего хода «лучшее» из доступных полей.

Будем считать, что самое лучшее поле — центральное, ему уступают по качеству угловые поля, а тем — все остальные. Чтобы передать этот порядок, далее в коде создадим кортеж:

```
# поля от лучшего к худшему
BEST_MOVES = (4, 0, 2, 6, 8, 1, 3, 5, 7)
print("Я выберу поле номер", end=" ")
```

Теперь создадим список доступных ходов. В каждое из свободных полей, перечисленных в этом списке, цикл помещает крестик или нолик (тип фишек, которыми играет компьютер) и проверяет, не ведет ли данный ход к победе. Если да, то этот ход и следует сделать; функция возвращает его и останавливается. Если нет, то отменим ход и попробуем следующую альтернативу из списка.

```
for move in legal_moves(board):
    board[move] = computer
# если следующим ходом может победить компьютер, выберем этот ход
    if winner(board) == computer:
        print(move)
        return move
# выполнив проверку, отменим внесенные изменения
    board[move] = EMPTY
```

Поскольку выполнение функции дошло до этого места, компьютер не может выиграть на текущем ходу. Значит, надо проверить, не может ли человек выиграть на следующем ходу. Цикл опять перебирает список доступных ходов, помещает в каждое из свободных полей нолик или крестик (тип фишек, которыми играет человек) и проверяет, не ведет ли данный ход к победе. Если да, то компьютер должен заблаговременно сделать этот ход, чтобы блокировать действия человека; функция возвращает номер позиции и останавливается. Если нет, отменим ход и попробуем следующую альтернативу из списка.

```
for move in legal_moves(board):
    board[move] = human
# если следующим ходом может победить человек, блокируем этот ход
    if winner(board) == human:
        print(move)
        return move
# выполнив проверку, отменим внесенные изменения
    board[move] = EMPTY
```

Поскольку выполнение функции дошло до этого места, ни одна сторона не может добиться победы на своем ближайшем ходу. Компьютер перебирает кортеж BEST_MOVES, в котором ходы упорядочены от лучших к худшим, и выбирает первый по счету доступный ход: как только номер позиции обнаруживается в списке разрешенных ходов, функция его возвращает.

```
# поскольку следующим ходом ни одна сторона не может победить.
# выберем лучшее из доступных полей
for move in BEST_MOVES:
    if move in legal_moves(board):
        print(move)
        return move
```

НА САМОМ ДЕЛЕ

В программе «Крестики-нолики» анализируются только ближайшие возможные ходы компьютера и человека. Симуляторы более серьезных стратегических игр, например шахматные программы, просчитывают следствия каждого хода намного дальше. В дереве ходов, которое они строят, много уровней ветвления. Мощность современных компьютеров такова, что они способны за короткое время анализировать множество игровых позиций, а у специализированных машин, таких как шахматный компьютер IBM Deep Blue, который одержал победу над чемпионом мира Гарри Каспаровым, производительность совершенно выдающаяся. Deep Blue перебирает в секунду свыше 200 миллионов позиций. Это, конечно, впечатляет, но только тех, кому неизвестно, что общее количество возможных расстановок фигур в шахматной игре имеет порядок 1050. Чтобы перебрать их все, компьютеру Deep Blue понадобилось бы как минимум 1 квадриллион 585 триллионов 489 миллиардов лет (для сравнения, возраст нашей Вселенной оценивают всего в 15 миллиардов лет).

Функция next_turn()

Функция принимает тип фишки, которой был сделан последний на данный момент ход, и возвращает тип фишки, которой должен быть сделан следующий ход. Параметр turn, принимающий значение X или 0, показывает, чей был последний ход – крестиков или ноликов.

```
def next_turn(turn):
    """Осуществляет переход хода."""
    if turn == X:
        return 0
    else:
        return X
Эта функция используется для того, чтобы чередовать ходы по мере того, как игроки будут их совершать.
```

Функция congrat_winner()

Функция принимает тип фишек победителя игры, типы фишек компьютерного противника и игрока человека. Она вызывается только в самом конце игры, поэтому в переменную the_winner может попасть X или 0, если один из противников добился перевеса, а также TIE, если игра закончилась вничью.

```
def congrat_winner(the_winner, computer, human):
    """Поздравляет победителя игры."""
    if the_winner != TIE:
        print("Три", the_winner, "в ряд!\n")
    else:
        print("Ничья!\n")
    if the_winner == computer:
        print("Как я и предсказывал, победа в очередной раз осталась за мной.\n")
        print("Вот еще один довод в пользу того, что компьютеры превосходят людей решительно во всем!")
    elif the_winner == human:
        print("О нет, этого не может быть! Неужели ты как-то сумел перехитрить меня, белковый?\n")
        print("Клянусь: я, компьютер, не допущу этого больше никогда!")
```

```
elif the_winner == TIE:
    print("Тебе несказанно повезло, дружок: ты сумел свести игру вничью. \n" \
          "Радуйся же сегодняшнему успеху! Завтра тебе уже не сужено его повторить.")
```

Функция main()

Основную часть программы я решил не оставлять на глобальном уровне, а поместить в отдельную функцию. Благодаря этому инкапсулируется и главная последовательность команд. Вообще, если программа, которую вы пишете, не совсем короткая и простая, ее основную часть лучше оформить в виде функции. Такую функцию вовсе не обязательно называть `main()`, ведь имя — чистая условность, но это общепринятая практика, которой не помешает следовать.

Итак, вот код основной части программы. Как видите, он удивительно близко совпадает с ранее написанным псевдокодом:

```
def main():
    display_instruct()
    computer, human = pieces()
    turn = X      board = new_board()
    display_board(board)
    while not winner(board):
        if turn == human:
            move = human_move(board, human)
            board[move] = human
        else:
            move = computer_move(board, computer, human)
            board[move] = computer
        display_board(board)
        turn = next_turn(turn)
    the_winner = winner(board)
    congrat_winner(the_winner, computer, human)
```

Запуск программы

Следующая строка вызывает на глобальном уровне функцию `main` (которая, в свою очередь, будет вызывать остальные функции):

```
# запуск программы
main()
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

В этой главе вы научились создавать собственные функции. Вы увидели, какими способами эти функции могут принимать и возвращать значения. Вы узнали, что такое области видимости и как можно читать и изменять глобальные переменные, находясь в области видимости одной из функций. Кроме того, вы усвоили привыч-

ку не злоупотреблять глобальными переменными, но применять, где это будет необходимо, глобальные константы. Наконец, вы чуть-чуть погрузились в мир искусственного интеллекта и поучаствовали в разработке компьютерного противника для игры-стратегии.

ЗАДАЧИ

- Доработайте функцию `ask_number()` так, чтобы ее можно было вызывать еще с одним параметром — кратностью (величиной шага). Сделайте шаг по умолчанию равным 1.
- Доработайте игру «Отгадай число» из главы 3 так, чтобы в ней нашла применение функция `ask_number()`.
- Доработайте новую версию игры «Отгадай число» (которую вы создали, решая предыдущую задачу) так, чтобы основная часть программы стала функцией `main()`. Для того чтобы игра началась, не забудьте вызвать эту функцию глобально.
- Напишите такую функцию `computer_move()`, которая сделала бы стратегию компьютера безупречной. Проверьте, можно ли создать непобедимого противника.

7 Файлы и исключения. Игра «Викторина»

Переменные — отличный механизм хранения информации и доступа к ней внутри работающей программы. Однако зачастую хочется сохранить данные, чтобы можно было обратиться к ним позже. Из этой главы вы узнаете, как использовать файлы в качестве такого долговременного хранилища информации. Помимо этого, вы овладеете техникой обработки ошибок, которые может вызвать работа вашего кода.

Прочитав эту главу, вы научитесь следующему:

- читать данные из текстовых файлов;
- записывать данные в текстовые файлы;
- осуществлять чтение структурированных данных и их запись в файл;
- перехватывать и обрабатывать ошибки во время исполнения программы.

Знакомство с игрой «Викторина»

Игра «Викторина» проверяет знания игрока, задавая ему вопросы на выбор ответа из нескольких вариантов. Каждая игра представляет собой «эпизод», относительно единий по тематике. Чтобы продемонстрировать возможности программы, я создал «Эпизод, от которого вы не сможете отказаться». Все его вопросы в той или иной степени связаны с мафией (некоторые — весьма косвенно).

Сильная сторона программы в том, что вопросы игры не «зашиты» в код, а хранятся в отдельном файле. Содержимое этого игрового файла легко изменить. Более того, с помощью простейшего текстового редактора, например **Блокнота** в Windows, вы можете создавать собственные эпизоды (игры) на любые темы — от зоологии до японских мультфильмов. На рис. 7.1 показан игровой процесс на примере моего эпизода.

Чтение текстового файла

Python позволяет с легкостью читать строки из *текстовых* файлов. Хотя есть разные типы текстовых файлов, в дальнейшем под текстовыми я буду понимать

исключительно файлы, содержащие текст, а не какую-то двоичную информацию¹. По некоторым причинам текстовые файлы удобны для долговременного хранения несложных данных. С ними легко работать: в большинстве операционных систем имеются по крайней мере базовые инструменты просмотра и редактирования текста.

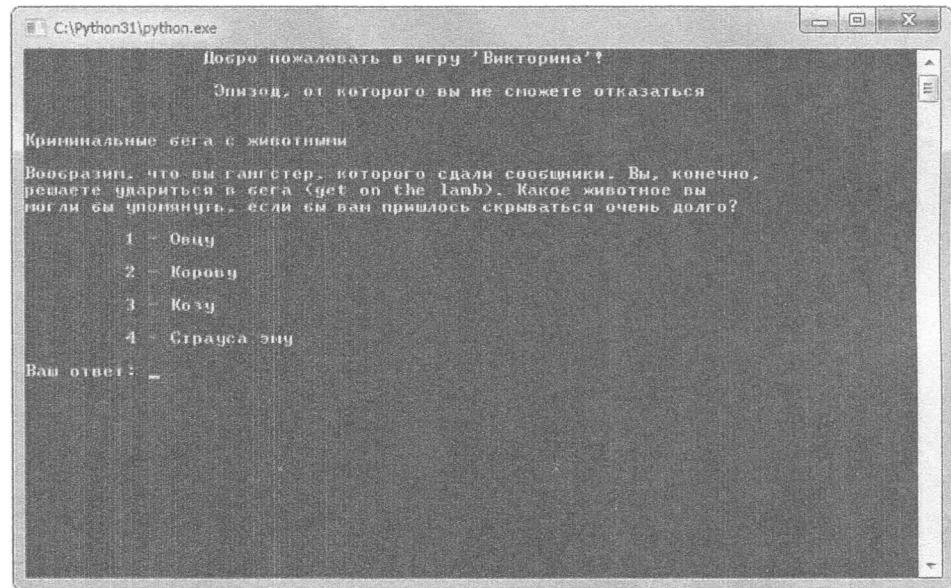


Рис. 7.1. Игроκу предлагаются четыре правдоподобные альтернативы, из которых верна лишь одна

Знакомство с программой «Прочитаем»

Программа «Прочитаем» иллюстрирует несколько способов чтения строк из текстового файла. В ней показано, как прочесть любой объем текста, от одного символа до целого файла. Кроме того, программа демонстрирует несколько приемов построчного чтения. Работа программы отражена на рис. 7.2.

Программа читает простой текстовый файл, который я создал в Блокноте. Этот файл вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 7; он называется `read_it.txt`. Вот его содержимое²:

```
Строка 1
Это строка 2
Этой строке достался номер 3
```

¹ Автор рассматривает такой текст как текст, составленный только из символов кодовой таблицы ASCII, в которой отсутствует кириллица. Но кириллический текст, как правило, не вызывает сбоев в работе интерпретатора. — Примеч. пер.

² Интерактивная сессия, которая показана далее, иллюстрирует чтение из этого русскоязычного, а не оригинального англоязычного текстового файла. — Примеч. пер.

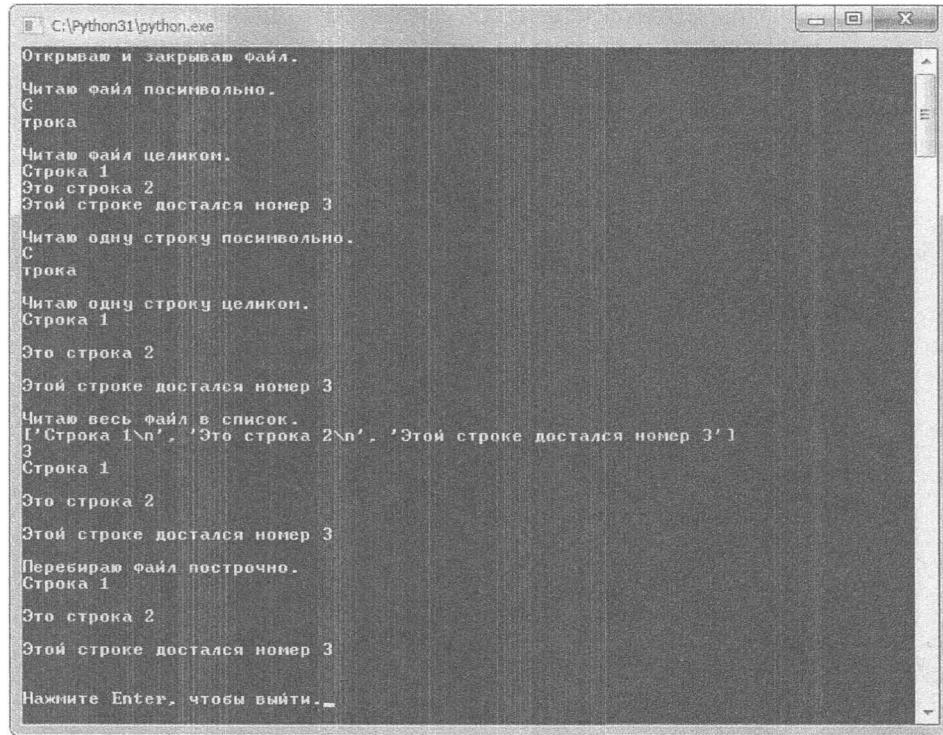


Рис. 7.2. Читаем файл несколькими разными способами

Код программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 7. Файл называется `read_it.py`:

```
# Прочитаем
# Демонстрирует чтение из текстового файла
print("Открываю и закрываю файл.")
text_file = open("read_it.txt", "r", encoding='utf-8')
text_file.close()
print("\nЧитаю файл посимвольно.")
text_file = open("read_it.txt", "r")
print(text_file.read(1))
print(text_file.read(5))
text_file.close()
print("\nЧитаю файл целиком.")
open("read_it.txt", "r", encoding='utf-8')
whole_thing = text_file.read()
print(whole_thing)
text_file.close()
print("\nЧитаю одну строку посимвольно.")
text_file = open("read_it.txt", "r", encoding='utf-8')
print(text_file.readline(1))
print(text_file.readline(5))
text_file.close()
```

```

print("\nЧитаю одну строку целиком.")
text_file = open("read_it.txt", "r", encoding='utf-8')
print(text_file.readline())
print(text_file.readline())
print(text_file.readline())
text_file.close()
print("\nЧитаю весь файл в список.")
text_file = open("read_it.txt", "r", encoding='utf-8')
lines = text_file.readlines()
print(lines)
print(len(lines))
for line in lines:
    print(line)
text_file.close()
print("\nПеребираю файл построчно.")
text_file = open("read_it.txt", "r", encoding='utf-8')
for line in text_file:
    print(line)
text_file.close()
input("\n\nНажмите Enter, чтобы выйти.")

```

Работу этого кода мы подробно рассмотрим в интерактивной сессии.

Открытие и закрытие файла

Прежде чем осуществлять чтение из файла (или запись в файл), вы должны его открыть. С этого я и начал программу «Прочитаем»:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')
```

Я применил функцию `open()`, чтобы открыть файл, а результат ее работы присвоил переменной `text_file`. В вызове функции три строковых аргумента: имя файла, режим доступа и кодировка файла.

С первым аргументом — `"read_it.txt"` — все обстоит очень просто. Поскольку я не расписал полный путь к файлу, Python ищет файл в той же директории, в которой находится сама программа. Доступ к файлу, находящемуся в какой угодно директории, возможен при условии, что вы сообщите путь к нему.

В качестве второго аргумента я указал `"r"`. Из этого Python заключает, что я хочу открыть файл на чтение. Файл можно открывать на чтение, на запись, а также на то и другое вместе. Некоторые разрешенные режимы доступа к текстовым файлам описаны в табл. 7.1. Параметр кодировки файла указан как `'utf-8'` для того, чтобы использовать Unicode в содержимом файла. В этом случае код будет работать независимо от операционной системы: Windows, Linux, Mac OS или любой другой.

Таблица 7.1. Избранные режимы доступа к текстовым файлам

Режим	Описание
<code>"r"</code>	Чтение из текстового файла. Если файл не существует, Python сообщит об ошибке
<code>"w"</code>	Запись в текстовый файл. Если файл существует, его содержимое будет заменено. Если файл не существует, он будет создан

Таблица 7.1 (продолжение)

Режим	Описание
"a"	Дозапись в текстовый файл. Если файл существует, новые данные будут дописаны в конец. Если файл не существует, он будет создан
"r+"	Чтение и запись в текстовый файл. Если файл не существует, Python сообщит об ошибке
"w+"	Запись и чтение из текстового файла. Если файл существует, его содержимое будет заменено. Если файл не существует, он будет создан
"a+"	Дозапись и чтение из текстового файла. Если файл существует, новые данные будут дописаны в конец. Если файл не существует, он будет создан

Открытый файл доступен через переменную `text_file`, которая представляет файловый объект. У файлового объекта есть несколько полезных методов, простейший из которых `close()`. Вызвав его, мы закроем файл и тем самым утратим возможность читать из него или записывать в него данные до тех пор, пока не откроем этот файл снова. Далее в программе я вызываю `close()`:

```
>>> text_file.close()
```

Считается хорошим тоном закрывать файл, завершая работу с ним.

Чтение текстового файла

Чтобы файл был хоть сколько-нибудь полезен, надо как-либо манипулировать его содержимым, а не просто открывать и закрывать. Вот почему следующим шагом я открываю файл и читаю его содержимое методом `read()` файлового объекта. Метод `read()` позволяет прочесть из файла указанное количество символов; эти символы метод возвращает как строку. Повторно открыв файл, я читаю из него ровно один символ:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')
>>> print(text_file.read(1))
C
```

Количество символов указывается в скобках после `read`.

Теперь прочитаем и выведем на экран следующие пять символов:

```
>>> print(text_file.read(5))
тока
```

Заметьте: прочитаны пять символов, которые следуют за `C`. Python запоминает, где я в последний раз остановился. Интерпретатор как будто «оставляет закладку» в этом месте файла, так что каждый следующий вызов `read()` начинается оттуда, где завершил работу предыдущий. Если дочитать до конца файла, то очередной вызов `read()` возвратит пустую строку.

Чтобы вернуться в начало файла, его можно закрыть и снова открыть. Это я и сделал:

```
>>> text_file.close()
>>> text_file = open("read_it.txt", "r")
```

Если не указать количество читаемых символов, будет возвращен весь текстовый файл одной строкой. Далее я осуществляю чтение файла в строку и присваиваю ее переменной, которую вывожу на экран:

```
>>> whole_thing = text_file.read()  
>>> print(whole_thing)  
Строка 1  
Это строка 2  
Этой строке достался номер 3
```

Если файл достаточно мал, то может быть целесообразным прочесть его целиком.

Теперь, поскольку прочтен весь файл, новые вызовы `read()` будут возвращать пустые строки. Чтобы предотвратить это, я снова закрываю файл:

```
>>> text_file.close()
```

Посимвольное чтение строки

Часто приходится читать файл строка за строкой. Символы из текущей строки текста можно читать методом `readline()`. Достаточно передать методу количество символов с начала строки, и он вернет эти символы как строку. Если не передавать никакого числа, метод прочтет все символы с текущей позиции до конца строки. После того как все символы строки прочитаны, место текущей строки занимает следующая за ней. Итак, заново открыв файл, я читаю первый символ первой строки:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')  
>>> print(text_file.readline(1))  
С
```

Потом я читаю следующие пять символов той же строки:

```
>>> print(text_file.readline(5))  
трома  
>>> text_file.close()
```

По этому примеру вы можете заключить, что `readline()` не отличается от `read()`. Однако `readline()` читает символы только текущей строки, тогда как методу `read()` доступен весь файл. Поэтому `readline()` обычно вызывают при необходимости прочесть ровно одну строку текста. В следующем коде я читаю файл построчно:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')  
>>> print(text_file.readline())  
Строка 1  
>>> print(text_file.readline())  
Это строка 2  
>>> print(text_file.readline())  
Этой строке достался номер 3  
>>> text_file.close()
```

Заметьте, что вместе с каждой строкой текстового файла на экран выводится пустая строка. Это обусловлено тем, что каждая из строк заканчивается escape-символом "\n".

Чтение всех строк файла в список

Еще один способ работы с отдельными строками текстового файла — использование метода `readlines()`, читающего текстовый файл в список, элементами которого являются строки. Вызовем этот метод:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')
>>> lines = text_file.readlines()
```

Теперь переменная `lines` ссылается на список, в котором каждый элемент совпадает с одной строкой исходного текстового файла:

```
>>> print(lines)
['Строка 1\n', 'Это строка 2\n', 'Этой строке достался номер 3\n']
```

Список `lines` ничем не хуже любого другого списка. Можно найти его длину и даже перебрать его элементы с помощью цикла:

```
>>> print(len(lines))
3
>>> for line in lines:
    print(line)
Строка 1
Это строка 2
Этой строке достался номер 3
>>> text_file.close()
```

Перебор строк файла

Кроме того, можно организовать цикл `for` для непосредственного перебора строк исходного файла:

```
>>> text_file = open("read_it.txt", "r", encoding='utf-8')
>>> for line in text_file:
    print(line)
Строка 1
Это строка 2
Этой строке достался номер 3
>>> text_file.close()
```

Как видите, управляющая переменная цикла (в данном случае `line`) последовательно принимает каждую из строк файла. При первой итерации цикла в ней окажется первая строка, при второй — вторая и т. д. Это наиболее изящный прием построчного чтения файла.

Запись в текстовый файл

Чтобы текстовый файл был приемлемой формой хранения данных, надо уметь записывать в него информацию. В Python добавлять строки в текстовые файлы столь же просто, как и читать из них. Покажу два основных способа записи.

Знакомство с программой «Запишием»

Программа «Запишием» создает текстовый файл с тем же содержанием, что и ранее рассмотренный `read_it.txt`. На самом деле новый файл создается и записывается дважды с помощью разных методов. Результат работы программы отображен на рис. 7.3.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 7. Файл называется `write_it.py`.

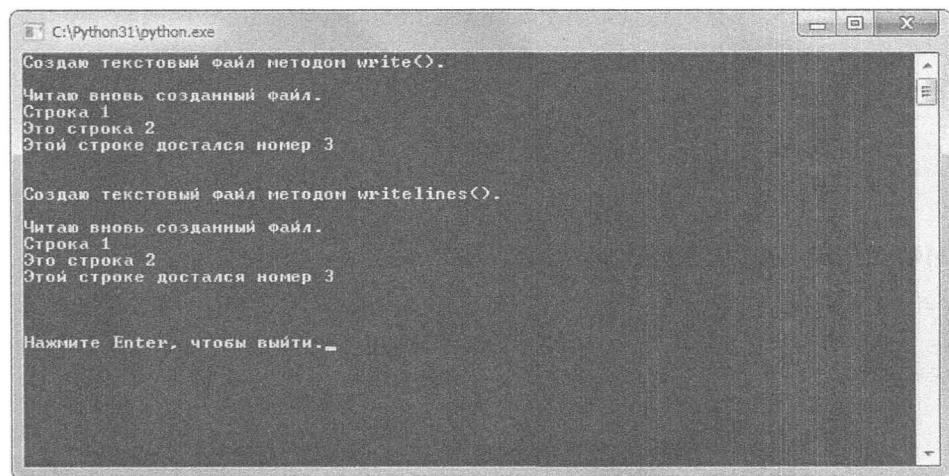


Рис. 7.3. Один и тот же файл создается дважды, разными методами файлового объекта

Запись строк в файл

Как и ранее, чтобы воспользоваться файлом, мы должны его открыть в требуемом режиме. Поэтому сначала моя программа открывает файл на запись:

```
# Запишием  
# Демонстрирует запись в текстовый файл  
print("Создаю текстовый файл методом write().")  
text_file = open("write_it.txt", "w", encoding='utf-8')
```

Файл `write_it.txt` создается как пустой текстовый файл, готовый к записи какого-либо текста из программы. Если бы файл `write_it.txt` уже существовал, то на замену ему был бы создан пустой файл, а все предыдущее содержимое утратилось бы.

Вслед за тем я применил метод файлового объекта `write()`, который записывает строку в файл:

```
text_file.write("Строка 1\n")  
text_file.write("Это строка 2\n")  
text_file.write("Этой строке достался номер 3\n")
```

Метод `write()` не вставляет символ перехода на новую строку автоматически после каждой строки, которую он пишет в файл. Символы `\n` вы можете помещать

там, где хотите видеть их в файле. Например, если убрать их из предшествующего кода, то программа записала бы одну длинную строку текста. Отнюдь не обязательно заканчивать строку, которая пишется в файл, символом \n. Конечный результат остался бы прежним, если бы я сформировал единую строку "Строка 1\n Это строка 2\n Этой строке достался номер 3\n" и записал его в файл одним-единственным вызовом метода write().

После этого я закрываю файл:

```
text_file.close()
```

И наконец, чтобы убедиться, что запись выполнена успешно, я читаю весь файл и вывожу его содержимое на экран:

```
print("\nЧитаю вновь созданный файл.")
text_file = open("write_it.txt", "r", encoding='utf-8')
print(text_file.read())
text_file.close()
```

Запись списка строк в файл

Теперь я создам тот же самый файл с помощью метода writelines() файлового объекта. Как и его аналог readlines(), writelines() работает со списком строк, но в противоположном направлении: не читает список из файла, а записывает в файл.

Сначала откроем файл на запись:

```
print("\nСоздаю текстовый файл методом writelines().")
text_file = open("write_it.txt", "w", encoding='utf-8')
```

Поскольку я открываю тот же самый файл write_it.txt, существующий файл уничтожается и его место занимает пустой.

Вслед за этим создам список строк, которые надо превратить в последовательные строки файла:

```
lines = ["Строка 1\n",
         "Это строка 2\n",
         "Этой строке достался номер 3\n"]
```

Опять же переходы на новую строку здесь расставлены так, как в записываемом файле.

Теперь помещу список lines в файл методом writelines():

```
text_file.writelines(lines)
```

После этого закрою файл:

```
text_file.close()
```

Наконец, чтобы удостовериться в тождестве с предыдущей версией, я печатаю содержимое файла на экране:

```
print("\nЧитаю вновь созданный файл.")
text_file = open("write_it.txt", "r", encoding='utf-8')
print(text_file.read())
text_file.close()
input("\n\nНажмите Enter, чтобы выйти.")
```

Итак, вы познакомились с большим количеством методов, обслуживающих чтение из файла и запись в файл. Краткое резюме по ним содержится в табл. 7.2.

Таблица 7.2. Избранные методы файлового объекта

Метод	Описание
close()	Закрывает файл. Закрытый файл недоступен для чтения и записи до тех пор, пока не будет открыт снова
read([число])	Читает указанное количество символов из файла и возвращает их в виде строки. Если число не указано, метод возвратит все символы, начиная с текущей позиции и до конца файла
readline([число])	Читает указанное количество символов из текущей строки файла и возвращает их в виде строки. Если число не указано, метод возвратит все символы, начиная с текущей позиции и до конца строки
readlines()	Читает все строки файла и возвращает список, элементами которого они все являются
write(строка)	Записывает строку в файл
writelines(список строк)	Записывает список строк текста в файл

Хранение структурированных данных в файлах

Текстовые файлы удобны тем, что типичный набор операций с ними доступен в любом текстовом редакторе. Но их функциональность ограничена: в них могут храниться только последовательности символов. Между тем иногда требуется сохранять данные более сложной природы, например список или словарь. Очевидный вариант — попробовать превратить содержимое такой структуры данных в цепочку символов, которые и сохранить в текстовом файле. Но Python предлагает гораздо лучшую альтернативу. Структурированные данные можно сохранять в файл всего одной строкой кода. Более того, в одном файле может находиться небольшая база данных, организованная наподобие словаря.

Знакомство с программой «Законсервируем»

Консервация — это долговременное неповреждающее сохранение. Именно так она и понимается в Python. Законсервировать структуру данных, такую как список или словарь, — значит в неприкосновенном виде сохранить ее в файле¹ (разница, пожалуй, только в том, что ваши руки не пропахнут маринадом).

Рассматриваемая программа консервирует три списка строк, сохраняет их в одном бинарном файле и извлекает их оттуда. Все перечисленное программа выполняет дважды: сначала с последовательным доступом (очень похоже на механизм чтения символов из текстового файла), а затем с произвольным доступом. Результаты

¹ Консервация — частный случай сериализации данных, которая известна и в других языках программирования. — Примеч. пер.

работы программы показаны на рис. 7.4, а ее код вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 7. Файл называется `pickle_it.py`.

```
C:\Python31\python.exe
Консервация списков.

Деконсервация списков.
['огурцы', 'помидоры', 'капуста']
['целые', 'кубиками', 'соломкой']
['Главпродукт', 'Чумак', 'Бондюэль']

Помещение списков на полку.

Извлечение списков из файла полки:
торговые марки - ['Главпродукт', 'Чумак', 'Бондюэль']
формы - ['целые', 'кубиками', 'соломкой']
виды овощей - ['огурцы', 'помидоры', 'капуста']

Нажмите Enter, чтобы выйти.
```

Рис. 7.4. Каждый список остается неприкосновенным, будучи помещенным в файл и извлеченным оттуда

Консервация данных и запись в файл

Первое, что мне нужно сделать в этой программе, — импортировать два новых модуля:

```
# Законсервируем
# Демонстрирует консервацию данных и доступ к ним через интерфейс полки
import pickle, shelve
```

Модуль `pickle` позволяет консервировать структуры данных и сохранять их в файлах, а модуль `shelve` обеспечивает произвольный доступ к законсервированным объектам в файлах.

Суть консервации очень проста: вместо того чтобы записывать в файл символы, мы будем туда записывать объект, представленный как последовательность байтов. Консервированные объекты хранятся в файлах так же, как и текстовые данные: доступ к ним на запись и чтение — последовательный.

Я создам три списка, которые буду консервировать и сохранять в файл:

```
print("Консервация списков.")
variety = ["огурцы", "помидоры", "капуста"]
shape = ["целые", "кубиками", "соломкой"]
brand = ["Главпродукт", "Чумак", "Бондюэль"]
Теперь открою на запись новый файл:
f = open("pickles1.dat", "wb")
```

Консервированные объекты следует хранить в бинарных (двоичных) файлах, а не в текстовых. Поэтому файл `pickles1.dat` открывается на запись двоичных данных, о чем свидетельствует режим доступа `"wb"`.

Режимы доступа к бинарным файлам, которые могут вам пригодиться, перечислены в табл. 7.3.

Таблица 7.3. Избранные режимы доступа к бинарным файлам

Режим	Описание
"rb"	Чтение из бинарного файла. Если файл не существует, то Python сообщит об ошибке
"wb"	Запись в бинарный файл. Если файл существует, его содержимое будет заменено. Если файл не существует, он будет создан
"ab"	Дозапись в бинарный файл. Если файл существует, новые данные будут дописаны в конец. Если файл не существует, он будет создан
"rb+"	Чтение и запись в бинарный файл. Если файл не существует, то Python сообщит об ошибке
"wb+"	Запись и чтение из бинарного файла. Если файл существует, его содержимое будет заменено. Если файл не существует, то он будет создан
"ab+"	Дозапись и чтение из бинарного файла. Если файл существует, новые данные будут дописаны в конец. Если файл не существует, он будет создан

Теперь я законсервирую три списка — `variety`, `shape` и `brand` — и сохраню их в файле `pickles1.dat` с помощью функции `pickle.dump()`. Эта функция принимает два аргумента: данные для консервации и файл, в котором их предлагается сохранить.

```
pickle.dump(variety, f)
pickle.dump(shape, f)
pickle.dump(brand, f)
f.close()
```

Итак, сначала будет законсервирован список, на который ссылается переменная `variety`. Кроме того, он будет помещен в файл `pickles1.dat` в виде целостного объекта. Затем то же самое последовательно произойдет со списками, на которые ссылаются переменные `shape` и `brand`. После этого программа закрывает файл.

Консервировать можно объекты разных типов, в частности:

- числа;
- строки;
- кортежи;
- списки;
- словари.

Чтение и расконсервация данных из файла

Теперь я извлеку три списка из файла и верну их из консервированного состояния в обычное с помощью функции `pickle.load()`. Эта функция принимает один аргумент: файл, из которого должен быть загружен следующий по порядку консервированный объект.

```
print("\nРасконсервация списков.")
f = open("pickles1.dat", "rb")
variety = pickle.load(f)
```

```
shape = pickle.load(f)
brand = pickle.load(f)
```

Программа читает первый консервированный объект из файла, расшифровывает его, в результате чего получается список ["огурцы", "помидоры", "калуста"]. Затем она делает этот список значением переменной *variety*. Потом программа читает следующий консервированный объект, приводит к виду списка ["целье", "кубиками", "соломкой"] и делает этот список значением переменной *shape*. В заключение программа читает третий (последний) консервированный объект и превращает в список ["Главпродукт", "Чумак", "Бондюэль"], который присваивает переменной *brand*.

В доказательство того, что механизм работает, я вывожу списки на экран:

```
print(variety)
print(shape)
print(brand)
f.close()
```

Функции консервации и расконсервации описаны в табл. 7.4.

Таблица 7.4. Избранные функции модуля *pickle*

Функция	Описание
<code>dump(объект, файл, [,bin])</code>	Пишет законсервированную версию объекта в файл. Если параметр <i>bin</i> равен <i>True</i> , объект записывается в двоичном формате, а если <i>False</i> — в менее практическом, но более удобочитаемом текстовом формате. По умолчанию <i>bin = False</i>
<code>load(файл)</code>	Расшифровывает очередной консервированный объект из файла и возвращает его

Полка для хранения консервированных данных

Теперь разовьем идею консервирования и поместим списки на *полку* в составе одного файла. С помощью модуля *shelve* (от англ. «ставить на полку») можно создать полку, работающую подобно словарю, в котором предусмотрен произвольный доступ к элементам.

Для начала создам полку *s*:

```
print("\nПомещение списков на полку.")
s = shelve.open("pickles2.dat")
```

Функция *shelve.open()* работает почти так же, как файловая функция *open()*, с той лишь разницей, что *shelve.open()* открывает не текстовый файл, а файл с консервированными объектами. В данном случае управление полкой осуществляется через переменную *s*; она функционирует как словарь, содержимое которого хранится в одном или нескольких файлах.

ПОДСКАЗКА

При вызове *shelve.open()* интерпретатор Python может прибавить расширение к указанному вами имени файла, а также создать для поддержки вновь созданной полки дополнительные файлы.

Функция `shelve.open()` принимает один аргумент — имя файла. Вторым, необязательным аргументом в ней является режим доступа. Если его не указать (как я и поступил), используется значение по умолчанию "с". Подробнее о режимах доступа к полке можно прочесть в табл. 7.5.

После всего этого я добавляю на полку три списка:

```
s["variety"] = ["огурцы", "помидоры", "капуста"]
s["shape"] = ["целые", "кубиками", "соломкой"]
s["brand"] = ["Главпродукт", "Чумак", "Бондюэль"]
```

Полка `s` по функциональности не отличается от словаря. В ней ключу "variety" поставлено в соответствие значение ["огурцы", "помидоры", "капуста"], ключу "shape" — значение ["целые", "кубиками", "соломкой"], а ключу "brand" — значение ["Главпродукт", "Чумак", "Бондюэль"]. Важно отметить, что ключ в составе полки может быть только строкой.

Осталось только вызвать метод `sync()` объекта-полки:

```
s.sync() # убедимся, что данные записаны
```

Правку, вносимую в файл полки, Python сохраняет в буфере, а затем периодически записывает буфер в файл. Для удостоверения того факта, что все сделанные изменения внесены в файл, можно вызвать метод полки `sync()`. Файл полки также обновляется при ее закрытии методом `close()`.

ПОДСКАЗКА

Имитировать полку можно простой консервацией словаря, но модуль `shelve` более эффективно работает с памятью. Поэтому, если вам понадобится произвольный доступ к консервированным объектам, лучше создайте полку.

Извлечение консервированных данных через интерфейс полки

Поскольку полка работает как словарь, объекты, законсервированные в ней, произвольно доступны по ключу. Докажем это: извлечем с полки `s` хранящиеся на ней списки в порядке, противоположном порядку их записи:

```
print("\nИзвлечение списков из файла полки:")
print("торговые марки - ", s["brand"])
print("формы - ", s["shape"])
print("виды овощей - ", s["variety"])
Наконец, закроем файл:
s.close()
input("\n\nНажмите Enter, чтобы выйти.")
```

Таблица 7.5. Режимы доступа к полке

Режим	Описание
"с"	Открытие файла на чтение или запись. Если файл не существует, он будет создан
"н"	Создание нового файла для чтения или записи. Если файл существует, его содержимое будет заменено
"r"	Чтение из файла. Если файл не существует, Python сообщит об ошибке
"w"	Запись в файл. Если файл не существует, Python сообщит об ошибке

НА САМОМ ДЕЛЕ

Консервация и расконсервация — удобные способы хранения структурированных данных и доступа к ним, но чем богаче структура, тем больше гибкости и мощи нужно для работы с ней. Два популярных способа представления сложноструктурной информации — базы данных и XML. Для поддержки и того и другого в Python есть модули. Подробнее об этом читайте на сайте <http://www.python.org>.

Обработка исключений

Когда Python сталкивается с ошибкой, он останавливает работу программы и выводит сообщение об ошибке. Точнее говоря, генерируется исключительная ситуация, или просто *исключение* (exception), которое свидетельствует, что произошло нечто из ряда вон выходящее. Если программа никак не реагирует на исключение, то Python прерывает ее работу и печатает на экране подробное описание возникшей ситуации.

Вот простой пример исключения в Python:

```
>>> num = float("Привет!")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    num = float("Привет!")
ValueError: could not convert string to float: Привет!
```

В этой интерактивной сессии интерпретатор Python попытался превратить строку "Привет!" в десятичную дробь. Поскольку этого сделать не получилось, было сгенерировано исключение, о котором сообщила система.

С помощью имеющихся в Python функций обработки исключительных ситуаций вы можете перехватывать и обрабатывать исключения. Как следствие, ваша программа не будет спонтанно «вылетать» даже в том случае, если пользователь вместо числа введет Привет!. Как минимум она будет благополучно завершаться, а как максимум — продолжать работу.

Знакомство с программой «Обработка»

Программа «Обработка» сначала дает пользователю шанс создать несколько исключений, а потом сама целенаправленно генерирует определенное количество исключительных ситуаций. Но работа программы не прерывается, а длится до самого конца благодаря обработке сгенерированных исключений. Работа программы показана на рис. 7.5. Ее код вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 7. Файл называется `handle_it.py`.

Применение конструкций try/except

Самый типичный способ обработки («перехвата») исключений — конструкция `try/except`. Оператор `try` отделяет фрагмент кода, который потенциально способен вызвать исключение. Потом пишется оператор с `except`, а блок кода, следующий за этим служебным словом, исполняется лишь в том случае, если система сгенерировала исключение.

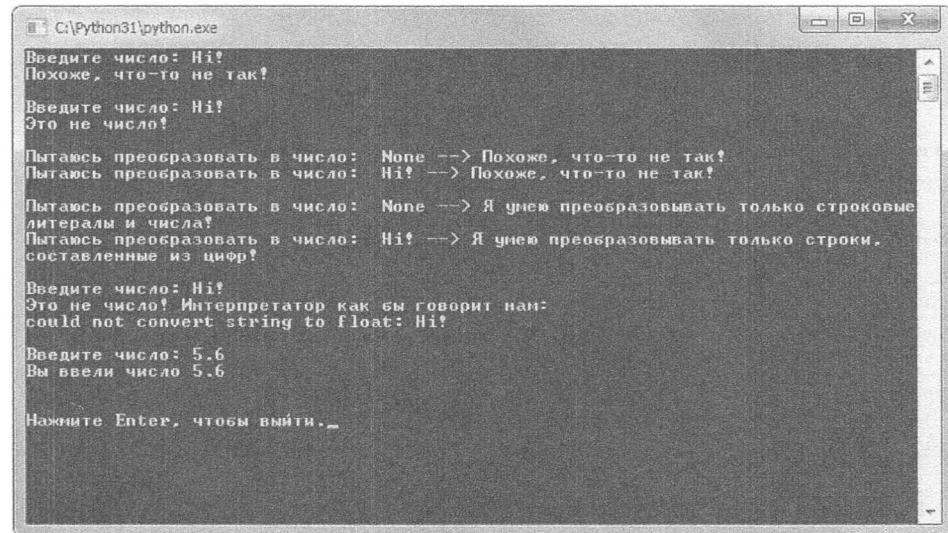


Рис. 7.5. Хотя «Привет!» нельзя превратить в число, программу не останавливает сгенерированное исключение

Первым делом программа «Обработаем» просит пользователя ввести число. Из пользовательского ввода программа получает строку и пытается превратить ее в десятичную дробь. Для обработки возможных исключений я применил конструкцию с `try` и `except`.

```
# Обработаем
# Демонстрирует обработку исключительных ситуаций
# try/except
try:
    num = float(input("Введите число: "))
except:
    print("Похоже, это не число!")
```

Если вызов `float()` генерирует исключение — например, из-за того, что пользователь ввел неконвертируемую строку, например "Привет!", — программа перехватывает возникшую исключительную ситуацию и уведомляет пользователя: Похоже, это не число!. Если же исключение не сгенерировано, то в переменной `num` окажется введенное пользователем число, а программа, проигнорировав блок с `except`, перейдет к дальнейшему коду.

Типы исключений

Разные виды ошибок влекут за собой разные типы исключений. Так, например, если попытаться преобразовать "Привет!" в число функцией `float()`, будет сгенерировано исключение `ValueError`, потому что символы в строке имеют неожиданное значение (это не цифры и не разделитель целой и дробной части в десятичной

дроби). Всего типов исключений больше 20; в табл. 7.6 перечислены только самые распространенные.

Таблица 7.6. Избранные типы исключений

Тип исключения	Описание
IOError	Генерируется, если невозможно выполнить операцию ввода/вывода, например, открыть на чтение несуществующий файл
IndexError	Генерируется, если в последовательности не найден элемент с указанным индексом
KeyError	Генерируется, если в словаре не найден указанный ключ
NameError	Генерируется, если не найдено имя (например, имя функции или переменной)
SyntaxError	Генерируется, если интерпретатор обнаруживает в коде синтаксическую ошибку
TypeError	Генерируется, если стандартная операция или функция применяется к объекту неподходящего типа
ValueError	Генерируется, если стандартная операция или функция принимает аргумент подходящего типа, но с неподходящим значением
ZeroDivisionError	Генерируется, если в операции деления или нахождения остатка второй аргумент — ноль

Оператор `except` позволяет точно указать, какой тип исключения будет обрабатываться. Чтобы назначить один и только один тип исключения, достаточно вписать его после `except`. Теперь я снова прошу пользователя ввести число, но перехватываться будет не всякое исключение, а только `ValueError`:

```
# обработка исключения определенного типа
try:
    num = float(input("\nВведите число: "))
except ValueError:
    print("Это не число!")
```

Функция `print` будет вызвана лишь в том случае, если генерируется `ValueError`. Как следствие, в этом случае можно напечатать на экране сообщение конкретного характера: Это не число!. Впрочем, если внутри конструкции `try` произойдет исключительная ситуация другого типа, то оператор `except` не перехватит ее и программа совершил аварийную остановку.

Указывать типы исключений — хорошая привычка: она позволяет в каждом частном случае применить особый прием. Единообразный перехват всех исключений, показанный в моем первом примере, может быть вреден и даже опасен. Страйтесь избегать его в своей практике.

ПОДСКАЗКА

В каких случаях следует перехватывать исключения? Везде, где взаимодействие программы с «внешним миром» наводит на мысли о возможных ошибках. Захват исключений нужен при открытии файла на чтение, даже если вам известно, что файл должен уже существовать. Можно также обрабатывать исключения при конвертировании данных из внешнего источника, например пользовательского ввода.

ХИТРОСТЬ

Итак, вы решили перехватить возможное исключение, но не знаете, какого оно типа. Разгадка одна: надо создать такое исключение самостоятельно. Так, например, если вы желаете обработать ошибку деления на ноль, но забыли английское название соответствующего исключения, откройте интерпретатор и попробуйте разделить число на ноль:

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    1/0
ZeroDivisionError: int division or modulo by zero
```

Из этой интерактивной сессии ясно, что исключение называется `ZeroDivisionError`. Вообще интерпретатор не стесняется рассказывать о типах исключений, вызываемых вашими командами.

Обработка нескольких типов исключений

Один и тот же фрагмент кода может вызвать исключения нескольких разных типов. К счастью, «отлов» их всех не представляет труда. Обычный способ перехвата нескольких типов исключений — перечислить их все в одном операторе `except` через запятую, заключив в скобки:

```
# обработка исключений нескольких разных типов
print()
for value in (None, "Привет!"):
    try:
        print("Пытаюсь преобразовать в число: ", value, "-->, end=" ")
        print(float(value))
    except (TypeError, ValueError):
        print("Похоже, это не число!")
```

Этот код пытается преобразовать два разных значения в десятичную дробь. Оба случая вызывают ошибки, но это ошибки разных типов. Вызов `float(None)` генерирует исключение `TypeError`, потому что функция может применяться только к строкам и числам. Вызов же `float("Привет!")` генерирует `ValueError`, потому что "Привет!" — это строка, но символы в ней имеют некорректное значение (это не цифры). Оператор `except` перехватывает оба типа исключений.

Есть и другой способ уловить исключения разных типов: несколько операторов `except`. В одной конструкции с `try` доступно сколько угодно последовательных операторов `except`:

```
print()
for value in (None, "Привет!"):
    try:
        print("Пытаюсь преобразовать в число: ", value, "-->, end=" ")
        print(float(value))
    except TypeError:
        print("Я умею преобразовывать только строки и числа!")
    except ValueError:
        print("Я умею преобразовывать только строки, составленные из цифр!")
```

Теперь на каждый тип исключения приходится свой блок кода. Когда преобразуемое значение равно `None`, генерируется `TypeError` и на экран выводится текст:

"Я умею преобразовывать только строки и числа!". При попытке преобразовать "Привет!" генерируется `ValueError`, а на экран будет выведено: "Я умею преобразовывать только строки, составленные из цифр!".

Несколько самостоятельных выражений `except` позволяют уникальным образом реагировать на каждый тип исключения в одной конструкции с `try`. В данном случае программа, исходя из типа исключения, сообщает пользователю о характере возникшей ошибки.

Аргумент исключения

Когда происходит исключительная ситуация, с ней может быть связано значение – так называемый *аргумент* исключения. Обычно это тот самый описывающий ошибку текст, который выводит интерпретатор Python. Аргумент можно извлечь в переменную, если указать ее имя после типа исключения через служебное слово `as`. В следующем примере я извлекаю аргумент исключения в переменную `e` и вывожу на экран вместе со своим обычным известием об ошибке:

```
# получение аргумента
try:
    num = float(input("\nВведите число: "))
except ValueError as e:
    print("Это не число! Интерпретатор как бы говорит нам:")
    print(e)
```

Добавление блока `else`

После всех блоков с `except` в конструкции с оператором `try` можно добавить заключительный блок `else`. Он будет выполнен лишь в том случае, если блок `try` сработает безошибочно.

```
# try/except/else
try:
    num = float(input("\nВведите число: "))
except ValueError:
    print("Это не число!")
else:
    print("Вы ввели число ", num)
input("\n\nНажмите Enter, чтобы выйти.")
```

Данный код выведет на экран значение `num` лишь в том случае, если первая команда в блоке с `try`, которая и присваивает значение этой переменной, не сгенерирует исключительную ситуацию. Значит, только существующие, успешно присвоенные значения `num` будут напечатаны на экране.

Вернемся к игре «Викторина»

Вооружившись основами работы с файлами и исключениями, вы готовы подступиться к игре «Викторина», поверхностный взгляд на которую могли бросить в на-

чале этой главы. Одна из особенностей этой программы состоит в том, что задания читаются из текстового файла. Значит, вы можете создавать свои собственные игровые эпизоды; для этого нужны только текстовый редактор и чуть-чуть вдохновения. При разборе кода вы увидите, что текстовый файл `trivia.txt`, который программа читает, должен находиться в той же директории, что и файл с кодом. Чтобы игра шла на ваших вопросах, достаточно заменить содержимое `trivia.txt` плодами вашего творчества.

Как организованы данные в текстовом файле

Прежде чем перейти собственно к коду, вам следует познакомиться с тем, как структурирован файл `trivia.txt`. Самая первая строка текста в нем — название эпизода. Все остальное содержимое файла — блоки по семь строк длиной, которые соответствуют вопросам. Блоков (вопросов) может быть сколько угодно. Вот общая схема блока:

```
<тема вопроса>
<вопрос>
<ответ 1>
<ответ 2>
<ответ 3>
<ответ 4>
<правильный ответ>
<комментарий>
```

Созданный мной игровой файл начинается так:

Эпизод, от которого вы не сможете отказаться

Обгоняя метеориты

Вообразим, что вы гангстер, которого сдали сообщники. Вы, конечно, /решаете удариться в бега. Какое животное вы /могли бы упомянуть, если бы вам пришлось скрываться очень долго?

Овцу

Корову

Козу

Страуса

1

Всех овец когда-нибудь стригут!

Проходите, "крестный отец" ожидает вас

Вообразим, что вы приглашены отобедать с крестным отцом. Как вы /обратитесь к своему собеседнику?

мистер Ричард

мистер Домино

мистер Браун

мистер Чеккер

3

Джеймса Брауна называют "крестным отцом" музыки в стиле Soul.

Для экономии места я показал только первые 15 строк файла — два вопроса. Весь файл `trivia.txt` вы можете увидеть на сайте-помощнике (www.courseptr.com/downloads), где он доступен в папке Chapter 7.

Запомните, что самая первая строка файла (Эпизод, от которого вы не сможете отказаться) — это название эпизода, то есть общая тема игры. Следующие семь строк представляют первый вопрос, а дальнейшие семь строк — второй вопрос. Так, строка "Обгоняя метеориты" — тема (или, как иногда говорят, категория) первого вопроса. Категория — это всего лишь удобный способ предвосхитить очередной вопрос. Следующая строка — "Вообразим, что вы гангстер, которого сдали сообщники. Вы, конечно, решаете удариться в бега. Какое животное вы /могли бы упомянуть, если бы вам пришлось скрываться очень долго?" — первый вопрос игры. Последующие четыре строки — "Овцу", "Корову", "Козу", "Страуса" — представляют четыре альтернативы, из которых игрок должен выбрать. Очередная строка, содержащая только цифру 1, кодирует номер правильного ответа. В данном случае правильный ответ — первый ("Овцу"). И наконец, еще одна строка — "Всех овец когда-нибудь стригут!" — объясняет, почему правилен один ответ и ошибочны остальные. Прочие вопросы построены по той же модели.

Важно заметить, что в некоторых строках я использовал слеш (/). Это обозначение перехода на новую строку. Оно требуется в связи с тем, что Python не умеет автоматически переносить по словам текст, выводимый на экран. Читая строки из файла, программа вместо слеша везде вставляет escape-символ новой строки. При разборе дальнейшего кода вы увидите, как именно это делается.

Функция open_file()

Первым делом надо объявить функцию open_file(), которая будет принимать имя файла и режим доступа к нему (оба аргумента — строки), а возвращать соответствующий файловый объект. С помощью try и except я предупреждаю исключение типа IOError, генерируемое при ошибках ввода-вывода, например при попытке чтения из несуществующего файла.

Если перехвачено исключение, то открыть файл с вопросами не удалось. При таком раскладе нечего и думать о продолжении игры; программа выведет сообщение об ошибке и вызовет функцию sys.exit(). Эта функция генерирует такое исключение, которое неизбежно останавливает программу. Следует прибегать к sys.exit() лишь как к последнему средству прервать работу программы. Заметьте, что для вызова sys.exit() я импортировал модуль sys.

```
# Викторина
# Игра на выбор правильного варианта ответа.
# вопросы которой читаются из текстового файла
import sys
def open_file(file_name, mode):
    """Открывает файл."""
    try:
        the_file = open(file_name, mode, encoding='utf-8')
    except IOError as e:
        print("Невозможно открыть файл", file_name, ". Работа программы будет завершена.\n", e)
        input("\n\nНажмите Enter, чтобы выйти.")
```

```

        sys.exit()
else:
    return the_file

```

Функция next_line()

Затем я объявляю функцию `next_line()`, которая принимает файловый объект и возвращает очередную строку текста из него:

```

def next_line(the_file):
    """Возвращает в отформатированном виде очередную строку игрового файла."""
    line = the_file.readline()
    line = line.replace("/", "\n")
    return line

```

Прежде чем возвратить строку, я решил выполнить кое-какое форматирование: заменить все слеши escape-символами новой строки. Это связано с тем, что Python не умеет автоматически переносить по словам текст, печатаемый на экране. Прием, использованный мной, дает автору игрового файла некоторую власть над форматом вывода: он может указать, где именно нужен переход на новую строку, чтобы не разрывать слово. Чтобы увидеть, как это работает, сравните файл `trivia.txt` и формат игровых вопросов на экране. Можете попробовать удалить слеши из текстового файла и посмотреть, что получается.

Функция next_block()

Функция `next_block()` читает очередной блок строк, соответствующий одному вопросу. Она принимает файловый объект и возвращает четыре строки (тема вопроса, текст вопроса, правильный ответ, комментарий), а также список (четыре строки — возможные ответы на вопрос).

```

def next_block(the_file):
    """Возвращает очередной блок данных из игрового файла."""
    category = next_line(the_file)
    question = next_line(the_file)
    answers = []
    for i in range(4):
        answers.append(next_line(the_file))
    correct = next_line(the_file)
    if correct:
        correct = correct[0]
    explanation = next_line(the_file)
    return category, question, answers, correct, explanation

```

После того как будет достигнут конец файла, чтение очередной строки возвращает пустую строку. Поэтому, когда программа добирается до конца `trivia.txt`, в переменную `category` попадает `""`. После того как `category` становится пустой строкой, функция `main()` завершает игру.

Функция welcome()

Функция `welcome()` приветствует игрока и объявляет название эпизода. Название передается функции как строка; название выводится на экран вслед за приветствием.

```
def welcome(title):
    """Приветствует игрока и сообщает тему игры."""
    print("\t\tДобро пожаловать в игру 'Викторина'!\n")
    print("\t\t", title, "\n")
```

Настройка игры

Теперь я создам функцию `main()`, которая будет отвечать за основную часть игрового процесса. В начале функции выполняется настройка: надо открыть игровой файл, извлечь из него тему игры (первую строку), поприветствовать игрока и сделать счет равным 0.

```
def main():
    trivia_file = open_file("trivia.txt", "r")
    title = next_line(trivia_file)
    welcome(title)
    score = 0
```

Задание вопроса

Вслед за тем я читаю семь строк, которые соответствуют первому вопросу, в переменные и запускаю цикл `while`, который будет продолжать задавать вопросы до тех пор, пока `category` не станет равной пустой строке. В этом последнем случае очевидно, что достигнут конец игрового файла и новые итерации цикла не нужны.

Вопрос задается так: на экран выводятся тема (категория), текст вопроса и четыре доступных ответа.

```
# извлечение первого блока
category, question, answers, correct, explanation = next_block(trivia_file)
while category:
    # вывод вопроса на экран
    print(category)
    print(question)
    for i in range(4):
        print("\t", i + 1, "-", answers[i])
```

Получение ответа

Ответ игрока извлекается из пользовательского ввода:

```
# получение ответа
answer = input("Ваш ответ: ")
```

Проверка ответа

Теперь надо сравнить ответ игрока с правильным ответом. Если они совпадут, то программа поздравляет игрока и увеличивает его счет на единицу; если совпадения нет, игрок извещается об ошибке. В обоих случаях на экран выводится комментарий (который поясняет, почему верен правильный ответ) и текущий счет.

```
# проверка ответа
if answer == correct:
    print("\nДа!", end=" ")
    score += 1
else:
    print("\nНет.", end=" ")
print(explanation)
print("Счет:", score, "\n\n")
```

Переход к следующему вопросу

Теперь вызовем функцию `next_block()`, чтобы прочитать блок строк, соответствующих очередному вопросу. Если вопросы закончились, в переменной `category` окажется пустая строка и следующей итерации цикла не будет.

```
# переход к следующему вопросу
category, question, answers, correct, explanation = next_block(trivia_file)
```

Завершение игры

После окончания цикла я закрываю файл `trivia.txt` и вывожу результат игрока на экран:

```
trivia_file.close()
print("Это был последний вопрос!")
print("На вашем счету", score)
```

Запуск функции `main()`

Последние строки кода отвечают за запуск функции `main()` и выход из игры:

```
main()
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

Из этой главы вы узнали о файлах и исключениях. Вы научились читать данные из текстовых файлов, увидели, как можно прочесть в переменную отдельный символ и даже целый файл. Вы освоили несколько способов построчного чтения из текстового файла (а это, пожалуй, самый распространенный способ доступа к тексту). Кроме того, вам стало известно, как записать в файл любой объем текста, от одного

символа до целого списка строк. Теперь вы располагаете сведениями о том, как сохранять в файлах сложные структуры данных, консервируя их, и как работать с группой объектов, законсервированных в двоичном файле, с помощью полки. Вы научились обрабатывать исключительные ситуации, генерируемые во время исполнения программы: «перехватывать» исключения определенных типов и создавать код, обрабатывающий их. В конце главы вы применили на практике известные вам сведения о файлах и исключениях, создав программу, которая позволит всем желающим играть в «Викторину» и писать собственные игровые вопросы.

ЗАДАЧИ

- Доработайте игру «Викторина» таким образом, чтобы у каждого вопроса появился «номинал» — уникальное количество очков. В конце игры сумма очков пользователя должна стать равной сумме номиналов вопросов, на которые он ответил верно.
- Доработайте игру «Викторина» таким образом, чтобы она хранила в файле список рекордов. В список должны попадать имя и результат игрока-рекордсмена. Используйте для хранения таблицы рекордов консервированный объект.
- Реализуйте ту же самую функциональность, что и в предыдущей задаче, иным способом: на этот раз сохраните список рекордов в обычном текстовом файле.
- Создайте эпизод игры «Викторина», который будет проверять осведомленность игрока о работе с файлами и исключениями в Python.

8 Программные объекты. Программа «Моя зверюшка»

Объектно-ориентированное программирование (ООП) — особый способ мышления программиста. Это современная методика, которая широко применяется в промышленной разработке программ; на ее основе создается большинство современных коммерческих приложений. Основная строевая единица в ООП — это *программный объект*, который часто называют просто объектом. В этой главе вы прочитаете об объектах и тем самым сделаете свои первые шаги на пути к пониманию ООП.

Вам предстоит научиться:

- создавать классы, определяющие функциональность объектов;
- писать для объектов методы и создавать атрибуты;
- создавать объекты как экземпляры классов;
- ограничивать доступ к атрибутам объекта.

Знакомство с программой «Моя зверюшка»

Программа «Моя зверюшка» позволяет пользователю завести виртуальное домашнее животное и заботиться о нем. Пользователь сам присваивает зверюшке имя; судьба прирученного им питомца оказывается полностью в его руках, но уход за живым существом — немалый труд. Чтобы зверюшка оставалась в хорошем настроении, надо кормить ее и играть с ней. Кроме того, у зверюшки можно спрашивать, как она себя чувствует; диапазон возможных ответов — от полной умиротворенности до страха и отвращения. Работа программы «Моя зверюшка» проиллюстрирована на рис. 8.1–8.3.

Хотя такую программу можно написать и без объектов, я реализовал зверюшку как программный объект. Это упрощает доработку программы, в частности ее расширение. Ведь если создана одна зверюшка, то совсем не проблематично вызвать к жизни еще десяток. Так недалеко и до фермы виртуальных питомцев, о которой, кстати, я предложу вам задуматься в задачах к этой главе.

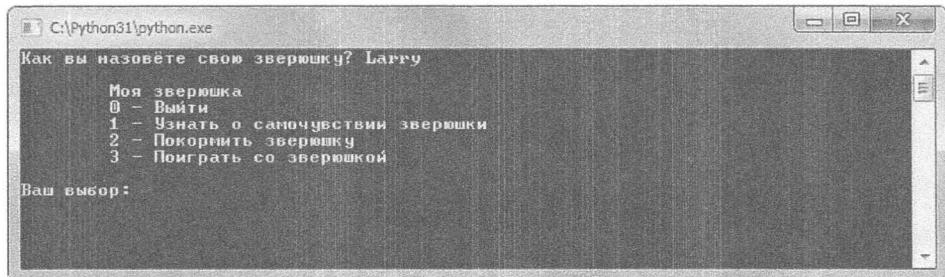


Рис. 8.1. Выберите своему питомцу имя

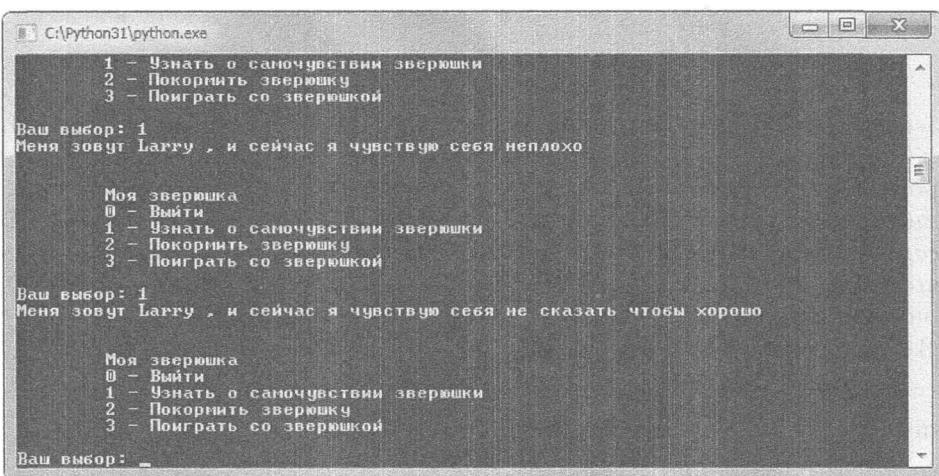


Рис. 8.2. Если не кормить или не развлекать зверюшку, ее настроение ухудшится

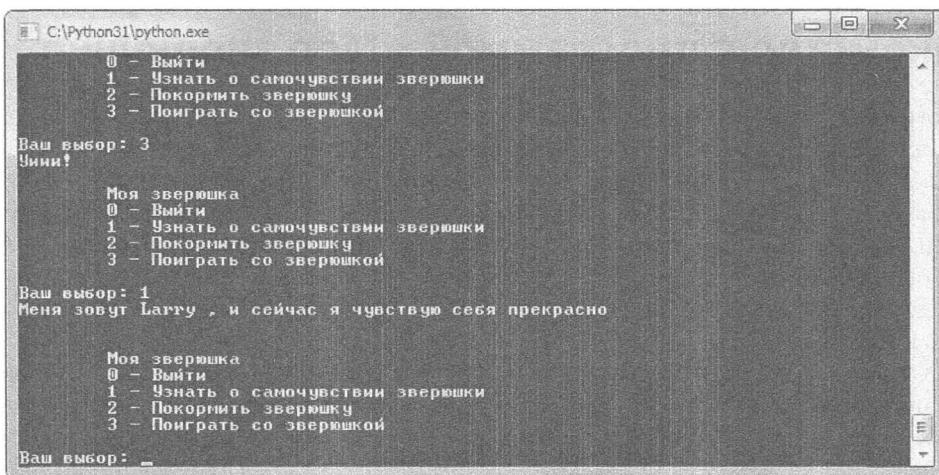


Рис. 8.3. Однако при должной заботе ваш зверек вернется в прежнее, хорошее расположение духа

Основы объектно-ориентированного подхода

Традиционно считается, что ООП — сложная техника, но мне кажется, что этот подход даже проще, чем некоторые из понятий, уже освоенных вами. В сущности, ООП просто представляет содержимое программы наиболее естественным для нашего мира способом.

Большинство реалий, отражаемых в программах (начиная с банковского счета и заканчивая инопланетным космическим кораблем), как-либо связано с внешним миром. ООП позволяет представить реальный объект как программный объект. Как и в жизни, у программного объекта есть несколько характеристик, которые на языке ООП называются *атрибутами*, и способов поведения, которые принято называть *методами*. Например, космический корабль пришельцев как программный объект мог бы иметь в числе атрибутов местоположение, уровень топлива, а в числе методов — способность двигаться и применять оружие.

Объекты создаются (или, как иногда говорят приверженцы ООП, *инстанцируются*) на основе описаний, называемых *классами*. Класс — это такой фрагмент кода, в котором объявлены атрибуты и методы. Классы подобны чертежам: это не объекты, а схемы объектов. Как прораб может возвести несколько зданий по одному и тому же чертежу, так и программист может создать сколько угодно объектов одного и того же класса. В результате у объектов одного и того же класса, также называемых *экземплярами* этого класса, будет единообразная структура. Если, например, есть класс для банковских счетов, то на его основе можно создать множество объектов — банковских счетов, не отличающихся по базовой структуре. У каждого может быть, например, атрибут со значением текущего баланса.

Когда два дома, построенных по одному и тому же чертежу, независимо подвергаются отделке, это можно сравнить с тем, как разным объектам одного и того же класса присваиваются уникальные значения атрибутов. К примеру, баланс одного счета может быть равен 100, а другого — 1 000 000.

ПОДСКАЗКА

Если рассуждения об ООП пока не совсем ясны вам, не волнуйтесь. Я хотел дать лишь самое общее представление о том, что такое объекты. Как и всегда в программировании, для усвоения новых понятий только читать недостаточно. Увидев работающий код на Python, в котором объявляются классы и создаются объекты, вы скоро поймете ООП и начнете применять этот подход в своей практике.

Создание классов, методов и объектов

Чтобы построить объект, нужен чертеж — класс. Практически всегда в состав классов входят методы — «умения» объектов. Можно создать класс и без методов, но это будет неинтересно.

Знакомство с программой «Просто зверюшка»

«Просто зверюшка» — первый для вас пример класса, написанного на Python. В этой программе задана исключительно несложная зверюшка, которая умеет

только приветствовать пользователя (несложно, зато вежливо). Результат работы программы показан на рис. 8.4.

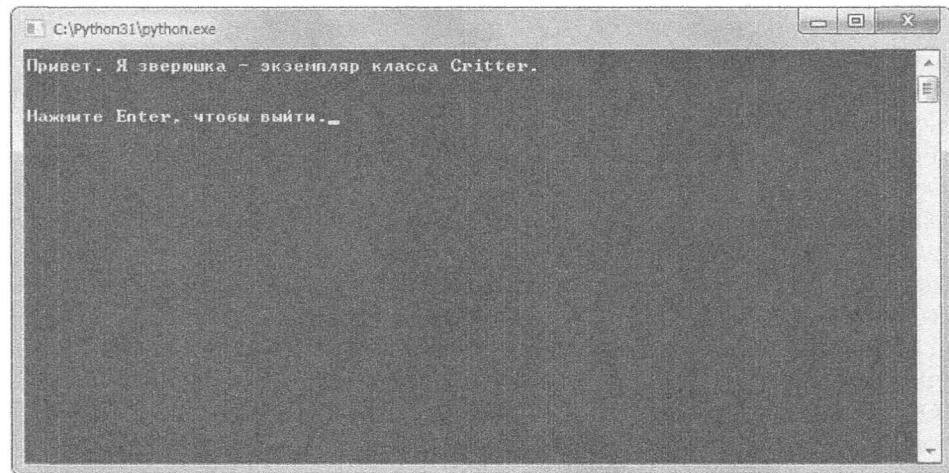


Рис. 8.4. Когда программа вызывает метод talk() объекта Critter, зверюшка приветствует всех

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 8. Файл называется simple_critter.py. Вся программа очень коротка:

```
# Просто зверюшка
# Демонстрирует простейшие класс и объект
class Critter(object):
    """Виртуальный питомец"""
    def talk(self):
        print("Привет. Я зверюшка - экземпляр класса Critter.")
# основная часть
crit = Critter()
crit.talk()
input("\n\nНажмите Enter, чтобы выйти.")
```

Объявление класса

Программа начинается с объявления класса — «чертежа», по которому будет создана моя первая зверюшка. Первая строка объявления называется *заголовком класса*:

```
class Critter(object):
```

Я набрал служебное слово `class` и вслед за ним выбранное имя класса — `Critter`. Как вы можете заметить, имя класса начинается с прописной буквы. Строго говоря, Python этого не требует, но такова общепринятая практика. Всегда начинайте имена классов с прописной буквы.

Я попросил интерпретатор создать мой новый класс с опорой на фундаментальный встроенный тип `object`. Новый класс можно создать на основе от `object` или любого ранее объявленного класса, но подробнее об этом мы поговорим в следующей главе. В этой главе все классы формируются на основе `object`.

Следующая строка кода документирует класс. Хорошая документирующая строка сообщает, какого рода объекты можно создавать с помощью класса. Я документировал свой класс предельно просто:

```
"""Виртуальный питомец"""
```

Объявление метода

В заключение в составе класса объявляется метод. По своей структуре метод очень схож с функцией:

```
def talk(self):  
    print("Привет. Я зверюшка - экземпляр класса Critter.")
```

В сущности, можно представлять себе методы как функции, связанные с объектами (этот подход уже знаком вам на примере строковых и списочных методов). Метод `talk()` выводит на экран строку Привет. Я зверюшка – экземпляр класса `Critter`.

Легко заметить, что у `talk()` заявлен один параметр `self`, который в действительности не используется. Каждый *метод экземпляра*, то есть метод, которым обладают все объекты данного класса, должен иметь особый первый параметр, по договоренности называемый `self`. Он позволяет методу сослаться на сам объект. Сейчас вам рано задумываться о параметре `self`: как он работает, вы увидите несколько дальше в этой главе.

ЛОВУШКА

Если создать метод экземпляра без параметров, то при попытке его вызвать интерпретатор выдаст ошибку. Запомните, что у всех методов экземпляра должен быть специальный первый параметр `self`.

Создание объекта

После того как я создал класс, инстанцировать его (создать объект-экземпляр) можно всего одной строкой кода:

```
crit = Critter()
```

Этот код создает новый объект класса `Critter` и делает его значением переменной `crit`. Заметьте, что за именем класса `Critter` в этой команде следуют скобки. Это жизненно важно для создания нового объекта.

Вновь созданный экземпляр класса – объект – можно связать с любой переменной, имя которой может быть и несозвучно с именем класса. Не следует, однако, пользоваться тем же названием, что и у класса, но набранным строчными буквами: это ведет к двусмысленности.

Вызов метода

У моего нового объекта есть метод `talk()`. Этот метод, как и любой другой из числа уже известных вам, — не что иное, как функция, принадлежащая объекту. Вызвать этот метод можно обычным образом, с помощью точечной нотации:

```
crit.talk()
```

Эта строка кода вызывает метод `talk()` по отношению к объекту класса `Critter`, который связан с переменной `crit`. Метод просто печатает на экране текст: Привет. Я зверюшка — экземпляр класса `Critter`..

Применение конструкторов

Теперь вы знаете на примере `talk()`, как создавать методы. Есть такой особый метод, называемый **конструктором**, который будет автоматически вызван сразу после создания нового объекта. Объявлять методы-конструкторы чрезвычайно полезно. Думаю, в своей практике вы будете создавать конструктор чуть ли не в каждом классе. Обычно конструктор применяется затем, чтобы установить начальные значения атрибутов объекта, хотя в следующем нашем примере его задача иная.

Знакомство с программой «Зверюшка с конструктором»

В программе «Зверюшка с конструктором» будет создан новый класс `Critter`, в котором содержится простейший метод-конструктор. Из этой программы вы увидите также, насколько легко создать несколько объектов одного и того же класса. Пробный запуск программы показан на рис. 8.5.

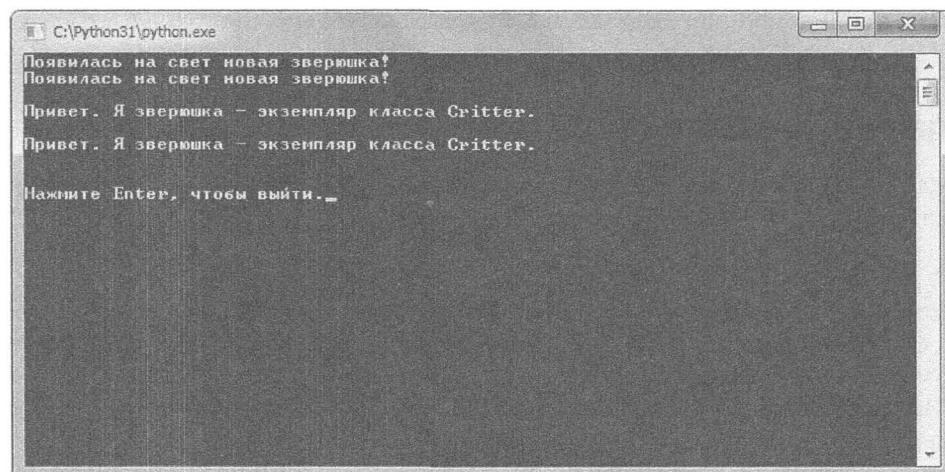


Рис. 8.5. Созданы две различные зверюшки, каждая из которых приветствует пользователя

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 8. Файл называется `constructor_critter.py`.

```
# Зверюшка с конструктором
# Демонстрирует метод-конструктор
class Critter(object):
    """Виртуальный питомец"""
    def __init__(self):
        print("Появилась на свет новая зверюшка!")
    def talk(self):
        print("\nПривет. Я зверюшка - экземпляр класса Critter.")
# основная часть
crit1 = Critter()
crit2 = Critter()
crit1.talk()
crit2.talk()
input("\n\nНажмите Enter, чтобы выйти.")
```

Создание конструктора

Первое, что в объявлении этого класса бросается в глаза, — метод-конструктор (иногда называемый также *методом инициализации*):

```
def __init__(self):
    print("Появилась на свет новая зверюшка!")
```

Обычно имена методов выбирает сам программист, но в данном случае я использовал особое имя, понятное интерпретатору. Если обозначить метод как `__init__`, Python сочтет, что это метод-конструктор. Будучи таковым, `__init__()` будет автоматически вызываться при возникновении каждого очередного объекта класса `Critter`. Как свидетельствует код метода, новый объект сообщает всем о себе, печатая на экране: Появилась на свет новая зверюшка!.

ПОДСКАЗКА

В Python есть набор встроенных «специальных методов», имена которых начинаются и заканчиваются двумя знаками подчеркивания. Метод-конструктор `__init__` — один из них.

Создание нескольких объектов

После того как написан класс, создать несколько его экземпляров проще простого. В основной части программы я создаю два объекта:

```
# основная часть
crit1 = Critter()
crit2 = Critter()
```

Появляются два объекта. При их инстанцировании метод-конструктор сообщает: Появилась на свет новая зверюшка!.

Каждый объект — настоящая, ни от кого не зависящая зверюшка. Чтобы доказать это, применю метод `talk()` к обоим объектам:

```
crit1.talk()
crit2.talk()
```

Хотя на экране печатается один и тот же текст: "\nПривет. Я зверюшка – экземпляр класса Critter.", – это результаты работы разных объектов.

Применение атрибутов

После создания объекта можно автоматически создавать и инициализировать его атрибуты с помощью метода-конструктора. Это очень удобно и широко распространено.

Знакомство с программой «Зверюшка с атрибутом»

«Зверюшка с атрибутом» создает новый тип объекта с атрибутом name. В классе Critter будет реализован метод-конструктор, который создает и инициализирует name. Этот новый атрибут нужен для того, чтобы зверюшка, приветствуя пользователя, могла представиться. Работу программы иллюстрирует рис. 8.6.

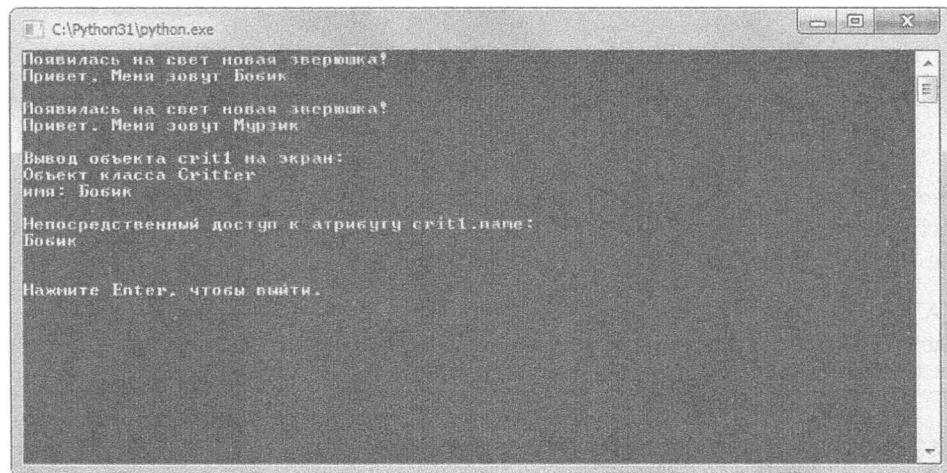


Рис. 8.6. На этот раз у каждого объекта класса Critter есть атрибут name, который используется в методе talk()

Вот код программы:

```
# Зверюшка с атрибутами
# Демонстрирует создание атрибутов объекта и доступ к ним
class Critter(object):
    """Виртуальный питомец"""
    def __init__(self, name):
        print("Появилась на свет новая зверюшка!")
```

```

        self.name = name
def __str__(self):
    rep = "Объект класса Critter\n"
    rep += "имя: " + self.name + "\n"
    return rep
def talk(self):
    print("Привет. Меня зовут", self.name, "\n")
# основная часть
crit1 = Critter("Бобик")
crit1.talk()
crit2 = Critter("Мурзик")
crit2.talk()
print("Вывод объекта crit1 на экран:")
print(crit1)
print("Непосредственный доступ к атрибуту crit1.name:")
print(crit1.name)
input("\n\nНажмите Enter, чтобы выйти.")

```

Инициализация атрибутов

В этой программе, как и в «Зверюшке с конструктором», метод-конструктор печатает сообщение Появилась на свет новая зверюшка!, но не останавливается на этом. Следующая строка кода создает у нового объекта атрибут `name`, значение которого становится равным значению параметра `name`. Поэтому в основной части программы код:

```
crit1 = Critter("Бобик")
```

создает новый объект класса `Critter`, у которого атрибут `name` равен строке «Бобик». Этот объект связывается с переменной `crit1`.

Чтобы вы могли глубже понять все происходящее, я наконец раскрою смысл таинственного параметра `self`. Как первый параметр любого метода, `self` автоматически становится ссылкой на объект, по отношению к которому вызван метод. Это значит, что через `self` метод получает доступ к вызывающему объекту, к его атрибутам и методам (он может, в частности, создавать у объекта новые атрибуты).

ПОДСКАЗКА

Первый параметр в заголовке метода теоретически можно назвать не `self`, а как-нибудь иначе, но поступать так не следует. Это противоречит типу мышления Python, который вы, как и другие программисты, обязаны уважать.

Таким образом, если вернуться к нашему методу-конструктору, будет ясно, что при создании объекта класса `Critter` параметр `self` становится ссылкой на этот объект, а параметр `name` принимает значение «Бобик». Стока кода:

```
self.name = name
```

создает атрибут `name` у объекта и приравнивает его к переменной `name`, значение которой равно «Бобик».

В основной части программы наш новый объект связывается с переменной crit. Иными словами, crit ссылается на объект с принадлежащим ему атрибутом name, значение которого равно "Бобик". Итак, у зверюшки появилось собственное имя!

Строка кода:

```
crit2 = Critter("Мурзик")
```

запускает ту же самую цепочку событий. Но на этот раз объект класса Critter создается со значением атрибута name, равным "Мурзик", и связывается с переменной crit2.

Доступ к атрибутам

Если бы атрибуты были недоступны, они были бы бесполезны. Вот почему я усовершенствовал метод talk() так, что теперь он пользуется атрибутом name экземпляра класса Critter: приветствуя пользователя, зверюшка представляется.

Чтобы моя первая зверюшка сообщила о себе, надо вызывать ее метод talk():

```
crit1.talk()
```

Через параметр self этот метод автоматически получает ссылку на объект:

```
def talk(self):
```

Затем функция print выводит на экран текст Привет. Меня зовут Бобик. Для этого с помощью self.name выполняется доступ к атрибуту name нашего объекта:

```
print("Привет. Меня зовут", self.name, "\n")
```

Те же самые события происходят при вызове метода применительно ко второму объекту:

```
crit2.talk()
```

Но на этот раз метод talk() выводит на экран текст Привет. Меня зовут Мурзик, потому что атрибут name зверюшки crit2 равен "Мурзик".

Как правило, атрибуты объекта доступны для чтения и изменения не только внутри класса, но и вне его. Так, в основной части моей программы выполняется прямой доступ к атрибуту name зверюшки crit1:

```
print(crit1.name)
```

Этот код выведет на экран текст Бобик. Вообще для доступа к атрибуту объекта вне класса, к которому относится этот объект, вы можете применять точечную нотацию вида: имя_переменной.имя_атрибута.

ПОДСКАЗКА

Обычно стремятся избежать прямого доступа к атрибутам объекта вне объявления класса. Подробнее о том, почему так происходит, вы прочтете в разделе «Что такое инкапсуляция объектов» этой главы.

Вывод объекта на экран

Если попробовать вывести объект на экран просто командой print(crit1), Python ответит очень загадочно, например так:

```
<__main__.Critter object at 0x00A0BA90>
```

Такая реакция свидетельствует о том, что в основной части моей программы я вывел на экран объект класса Critter. Но никакой полезной информации об объекте отсюда почерпнуть нельзя. Впрочем, все легко поменять: если в объявлении класса будет реализован специальный метод `__str__()`, это позволит представлять объекты строками для вывода на экран. Если мы попросим интерпретатор напечатать нам объект, то на экран будет выводиться та строка, которую возвращает метод `__str__()`.

Реализованный мной метод возвращает строку, которая содержит значение атрибута `name`. Вот почему при исполнении кода:

```
print(crit1)
```

на экране появляется текст:

```
Объект класса Critter  
имя: Бобик
```

ХИТРОСТЬ

Если даже вы не собираетесь выводить объекты на экран, создать метод `__str__()` все же не помешает. Взглянув на значения атрибутов объекта, можно понять, как именно работает программа или почему она не работает.

Применение атрибутов класса и статических методов

Атрибуты позволяют присвоить уникальные значения разным объектам одного и того же класса. Если у вас, например, заведено десять зверюшек, то каждая из них может носить особое имя. Но бывает и такая информация, которая относится не к индивидуальным объектам, а ко всему классу. Например, вы хотите следить за количеством созданных зверюшек. Можно присвоить каждому объекту класса Critter атрибут под названием `total`. Но тогда каждый раз при создании нового объекта атрибут `total` у всех уже существующих объектов должен быть увеличен на единицу. Реализовать это очень непросто.

К счастью, в Python можно создать значение, связанное с целым классом; это будет так называемый *атрибут класса*. Если класс — чертеж, то атрибут класса — приклеенная к чертежу записка. Она существует в одном экземпляре и не копируется¹ по мере того как будут возводиться все новые и новые однотипные дома.

В Python есть также методы, связанные с целым классом. Они называются *статическими* и в силу своих свойств часто применяются вместе с атрибутами класса.

Знакомство с программой «Классово верная зверюшка»

Нет, персонажи этой программы не имеют никакого отношения к социализму и классовой борьбе. Здесь всего лишь используются атрибуты и методы, принадлежащие

¹ Однако у каждого экземпляра появляется прокси-«записка», ссылающаяся на «записку» класса. — Примеч. науч. ред.

классу, а не отдельному объекту. В программе будет объявлен атрибут класса, который будет следить за количеством созданных объектов Critter, а также статический метод, выводящий соответствующее число. Результаты работы программы показаны на рис. 8.7.

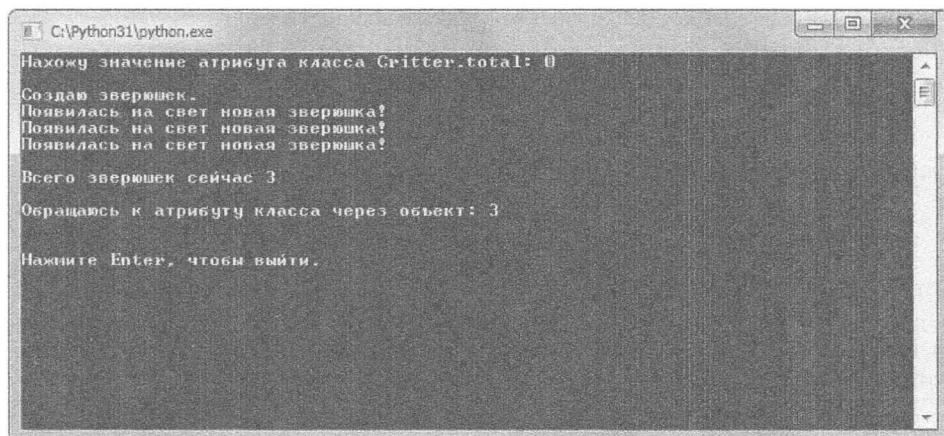


Рис. 8.7. Сколько зверушек родилось! Их количество программа отслеживает с помощью атрибута класса, а печатает это на экране статический метод

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 8. Файл называется `classy_critter.py`.

```

# Классово верная зверюшка
# Демонстрирует атрибуты класса и статические методы
class Critter(object):
    """Виртуальный питомец"""
    total = 0
    @staticmethod
    def status():
        print("\nВсего зверушек сейчас", Critter.total)
    def __init__(self, name):
        print("Появилась на свет новая зверюшка!")
        self.name = name
        Critter.total += 1
# основная часть
print("Нахожу значение атрибута класса Critter.total:", end=" ")
print(Critter.total)
print("\nСоздаю зверюшку.")
crit1 = Critter("зверюшка 1")
crit2 = Critter("зверюшка 2")
crit3 = Critter("зверюшка 3")
Critter.status()
print("\nОбращаюсь к атрибуту класса через объект:", end=" ")
print(crit1.total)
input("\n\nНажмите Enter, чтобы выйти.")

```

Создание атрибута класса

Вторая строка в моем объявлении класса:

```
total = 0
```

создает атрибут `total` класса `Critter` и присваивает ему значение 0. Атрибут класса создает каждая команда, которая расположена внутри класса (но вне любого из его методов) и присваивает значение ранее не существовавшей переменной. Присвоение выполняется всего один раз — когда Python впервые видит объявление класса. Таким образом, атрибут класса возникает прежде, чем появится хотя бы один объект, и значит, атрибутами класса можно пользоваться даже тогда, когда ни одного объекта данного класса в наличии нет.

Доступ к атрибуту класса

Несложно осуществить доступ к атрибуту класса; в своей программе я делаю это несколько раз. В основном разделе программы я вывожу атрибут `total` класса `Critter` на экран командой:

```
print Critter.total
```

В статическом методе `status()` я печатаю значение того же атрибута вслед за строкой:

```
print("\nВсего зверушек сейчас". Critter.total)
```

В методе-конструкторе я увеличиваю значение того же атрибута на единицу командой:

```
Critter.total += 1
```

После того как эта строка кода выполняется — а это происходит каждый раз при создании нового объекта, — значение `total` возрастает на 1.

Для доступа к атрибуту класса пользуйтесь точечной нотацией такого вида: `имя_класса.имя_атрибута`.

Можно, наконец, получить доступ к атрибуту класса через любой из объектов этого класса. Это и делает в основной части программы следующий код:

```
print(crit1.total)
```

Эта строка кода выводит на экран значение атрибута `total`, относящегося к классу `Critter`, а не к самому объекту. Прочесть значение атрибута класса можно из любого объекта, который принадлежит этому классу: я мог бы сказать `print(crit2.total)` или `print(crit3.total)`, и результат остался бы прежним.

ЛОВУШКА

Хотя для чтения атрибута класса можно воспользоваться объектом этого класса, запись в таком режиме невозможна. Чтобы изменить значение атрибута класса, осуществите доступ к нему через имя класса.

Создание статического метода

Первый метод класса называется `status()`:

```
def status():
    print("\nВсего зверюшек сейчас", Critter.total)
```

Это объявление — часть статического метода. Заметьте, что в списке параметров в заголовке метода отсутствует параметр `self`. Это потому, что `status()`, как и все статические методы, будет применяться к классу, а не к объекту. Значит, методу незачем передавать ссылку на объект и параметр для сохранения такой ссылки, подобный `self`, теряет смысл. У статических методов, конечно, могут быть параметры, но именно этот совершенно не нужен.

Прямо перед объявлением метода `status()` я поместил *декоратор* — команду, которая, как можно вообразить, чуть-чуть меняет («декорирует») смысл метода или функции. Благодаря этому декоратору метод в конце концов оказывается статическим:

```
@staticmethod
```

Итак, в результате работы показанных трех строк кода в классе появится статический метод `status()`, который выводит на экран количество объектов класса `Critter`, печатая значение атрибута класса `total`.

Для создания собственного статического метода следуйте моему образцу. Сначала укажите декоратор `@staticmethod`, а затем объявиите метод класса, который, поскольку он относится к целому классу, а не к одному из объектов, должен быть лишен параметра `self`.

Вызов статического метода

Вызов статического метода не представляет труда. В первой строке основной части программы я вызываю статический метод:

```
Critter.status()
```

Как вы наверняка догадываетесь, вызов возвратит 0, потому что класс еще не был инстанцирован. Обратите внимание на тот факт, что этот метод можно вызывать в ситуации, когда ни одного объекта данного класса нет. Поскольку статические методы вызываются через класс, для них неважно, существуют ли объекты.

После этого я создаю три объекта и сновазываю `status()`; теперь метод уведомляет, что есть три зверюшки. Механизм, как вы убедились ранее, работает благодаря тому, что в процессе исполнения метода-конструктора каждого объекта атрибут класса `total` увеличивается на 1.

Что такое инкапсуляция объектов

Впервые об инкапсуляции вы узнали на примере функций в разделе «Что такое инкапсуляция» главы 6. Там я показал, что инкапсулированная функция скрывает детали своего внутреннего устройства и работы от той части программы, которая

ее вызывает (это так называемый клиент функции). Если функция реализована удачно, то клиент взаимодействует с ней только путем передачи параметров и возвращения значений. С объектами, в сущности, дело обстоит точно так же. Клиентский код должен взаимодействовать с объектами, передавая параметры их методам и получая возвращенные значения. Притом в клиентском коде не принято напрямую изменять значения атрибутов объекта.

Как всегда, поясню на конкретном примере. Пусть у вас есть объект `Checking_Account` (банковский счет) с атрибутом `balance`. Пусть, кроме того, ваша программа должна уметь снимать деньги со счета; при снятии денег значение атрибута `balance` должно уменьшиться на какое-либо число. Чтобы выполнить искомую операцию, клиентский код мог бы просто вычесть из значения `balance` снятую сумму. При всем удобстве для клиентской стороны программы этот способ доступа проблематичен. Может быть, например, выполнено такое вычитание, после которого баланс счета станет отрицательным, а это неприемлемо, в особенности с точки зрения банка. Гораздо лучше иметь метод `withdraw()`, которым клиентский код может пользоваться для снятия денег, передавая для этого один параметр — запрашиваемую сумму. Тогда возникающие ошибки сумеет обработать сам объект: если, к примеру, сумма слишком велика, то объект может отвергнуть транзакцию. Непрямой доступ к атрибутам через методы — залог безопасности объекта.

Применение закрытых атрибутов и методов

По умолчанию все атрибуты и методы объекта *открыты* (`public`): клиентский код может непосредственно извлекать значения таких атрибутов и вызывать такие методы. Но для более последовательной инкапсуляции можно объявлять атрибуты и методы как *закрытые* (`private`). Это будет значить, что лишь другие методы того же объекта будут иметь доступ к таким атрибутам и методам.

Знакомство с программой «Закрытая зверюшка»

В программе «Закрытая зверюшка» создается объект, у которого есть как открытые, так и закрытые атрибуты и методы. Пробный пуск программы показан на рис. 8.8.

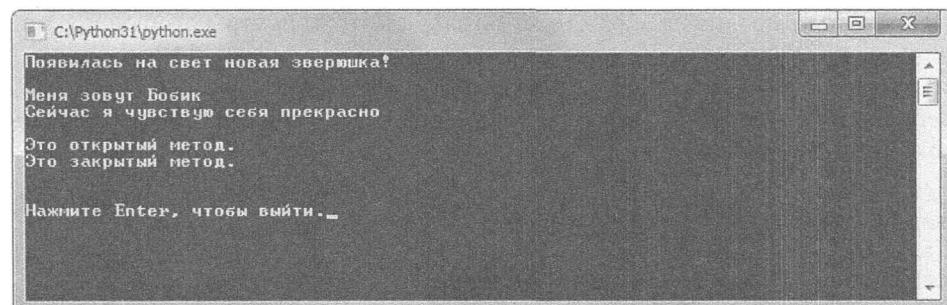


Рис. 8.8. Доступ к закрытому атрибуту и закрытому методу объекта осуществляется косвенно

Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 8. Файл называется `private_critter.py`.

Создание закрытых атрибутов

Чтобы ограничить прямой доступ клиентского кода к атрибутам объекта, можно сделать атрибуты закрытыми. В следующем примере метод-конструктор создает два атрибута: один открытый и один закрытый:

```
# Закрытая зверюшка
# Демонстрирует закрытые переменные и методы
class Critter(object):
    """Виртуальный питомец"""
    def __init__(self, name, mood):
        print("Появилась на свет новая зверюшка!")
        self.name = name # открытый атрибут
        self.__mood = mood # закрытый атрибут
```

Два символа подчеркивания, с которых начинается имя второго атрибута, заставляют интерпретатор рассматривать этот атрибут как закрытый. Чтобы создать свой собственный закрытый атрибут, просто начните его имя с двух символов подчеркивания.

ПОДСКАЗКА

Это верно и для атрибутов класса.

Доступ к закрытым атрибутам

Внутри объявления класса легко и удобно выполнять доступ к закрытым атрибутам будущего объекта (не забывайте, что закрытые атрибуты «закрыты» лишь от попыток обращения со стороны клиентского кода). В методе `talk()` я пользуюсь значением атрибута `__mood`:

```
def talk(self):
    print("\nМеня зовут", self.name)
    print("Сейчас я чувствую себя". __mood. "\n")
```

Этот метод выводит на экран текст из закрытого атрибута, который представляет текущее самочувствие зверюшки.

Если бы я попробовал осуществить доступ к тому же самому атрибуту вне объявления класса `Critter`, возникла бы проблема. Вот интерактивная сессия, из которой ясно, что я имею в виду:

```
>>> crit = Critter(name = "Бобик", mood = "прекрасно")
Появилась на свет новая зверюшка!
>>> print(crit.mood)
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
    print(crit.mood)
AttributeError: 'Critter' object has no attribute 'mood'
```

Генерируя исключение `AttributeError`, Python говорит нам, что у объекта `crit` нет атрибута `mood`. Если попробовать перехитрить интерпретатор и ввести имя атрибута с двумя символами подчеркивания, опять ничего не получится. Моя вторая попытка отражена далее в интерактивной сессии:

```
>>> print(crit.__mood)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(crit.__mood)
AttributeError: 'Critter' object has no attribute '__mood'
```

Снова было сгенерировано исключение `AttributeError`: система настаивает на том, что атрибута не существует. Значит ли это, что значение закрытого атрибута недоступно напрямую где бы то ни было, помимо объявления класса? Надо признать, нет. Python скрывает атрибут за особым именем; «достучаться» до атрибута, обратившись к нему по этому имени, все же возможно. Вот что я сделаю сейчас в интерактивной сессии:

```
>>> print(crit._Critter__mood)
прекрасно
```

Этот код выводит на экран значение неуловимого атрибута `mood`, которое в данном случае равно "прекрасно".

«Поскольку доступ к закрытым атрибутам извне класса технически возможен, какой же смысл в них?» — можете подумать вы. В Python закрытость — показатель того, что атрибут или метод предназначен лишь для внутреннего использования в объекте. Кроме того, не удастся по неосторожности, случайно воспользоваться таким атрибутом или методом.

ПОДСКАЗКА

Никогда не выполняйте прямой доступ к закрытым атрибутам и методам извне кода класса, в котором они объявлены.

Создание закрытых методов

Закрытый метод можно создать тем же самым несложным способом, что и закрытый атрибут: добавить в начало его имени два символа подчеркивания. Это я и сделал, объявляя в классе очередной метод:

```
def __private_method(self):
    print("Это закрытый метод.")
```

Этот закрытый метод доступен для остальных методов класса. Предполагается, что закрытые методы (как и закрытые атрибуты) найдут применение лишь внутри методов самого объекта.

Доступ к закрытым методам

Как и в случае с закрытыми атрибутами, доступ к закрытым методам объекта внутри объявления класса выполнить несложно. Я создал метод `public_method()`, в котором это иллюстрируется:

```
def public_method(self):
    print("Это открытый метод.")
    self._private_method()
```

Данный метод печатает на экране слова Это открытый метод. и затем вызывает закрытый метод объекта.

Не предполагается, что клиентский код будет осуществлять прямой доступ к закрытым методам. В новой интерактивной сессии я попытаюсь вызвать метод `private_method`:

```
>>> crit.private_method
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    crit.private_method
AttributeError: 'Critter' object has no attribute 'private_method'
```

Эта попытка генерирует уже знакомое нам исключение `AttributeError`: Python говорит, что метода с таким именем у объекта `crit` нет (хотя на самом деле всего лишь скрывает его за специальным именем). Если добавить два начальных символа подчеркивания, то сообщение об ошибке останется прежним:

```
>>> crit.__private_method()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    crit.__private_method()
AttributeError: 'Critter' object has no attribute '__private_method'
```

Однако все же остается техническая возможность вызвать закрытый метод (как и получить значение закрытого атрибута) из любой части программы. В доказательство привожу конец моей интерактивной сессии:

```
>>> crit._Critter__private_method()
Это закрытый метод.
```

Впрочем, как вы уже знаете, клиенту никогда не следует пытаться выполнять прямой доступ к закрытым методам объекта.

ПОДСКАЗКА

Закрытый статический метод создается таким же образом: достаточно добавить два знака подчеркивания в начало его имени.

Соблюдаем приватность

В основной части программы я веду себя как примерный ученик и не вмешиваюсь в частную жизнь объектов. Создав объект, я вызываю два его открытых метода:

```
# основная часть
crit = Critter(name = "Бобик", mood = "прекрасно")
crit.talk()
crit.public_method()
input("\n\nНажмите Enter, чтобы выйти.")
```

Метод `__init__()` объектов класса `Critter`, вызываемый автоматически при создании каждого такого объекта, оповещает пользователя о том, что на свет появилась зверюшка. Метод `talk()` объекта `crit` рассказывает о самочувствии зверюшки. Наконец, метод `public_method()` объекта `crit` выводит на экран строку `Это открытый метод.` и вызывает закрытый метод того же объекта, печатающий на экране строку `Это закрытый метод..` На этом программа заканчивается.

В каких случаях нужны закрытые атрибуты и методы

Вы научились «закрывать» методы и атрибуты. Значит ли это, что каждый атрибут любого класса надо обязательно закрыть от грубого и жестокого внешнего мира? Конечно, нет. Чтобы сделать код лучше, следует экономно «приправить» его закрытыми методами и атрибутами, хорошо подумав о том, что именно сделать недоступным из клиентской части программы. Многие программисты на Python убеждены, что клиентский код должен не напрямую менять значения атрибутов, а обращаться к ним через методы.

ПОДСКАЗКА

При реализации класса придерживайтесь двух правил: создавайте методы, существование которых сделает ненужным прямой доступ из клиентского кода к атрибутам объекта, и закрывайте лишь те атрибуты и методы, которые обслуживают внутренние операции объекта.

В работе с объектом полезно следовать еще двум правилам: избегайте непосредственного чтения и изменения значений атрибутов и никогда не пытайтесь прямо обращаться к закрытым атрибутам и методам объекта.

Управление доступом к атрибутам

Иногда вместо того, чтобы закрыть доступ к атрибуту, целесообразно только ограничить его. В некоторых случаях полезно, например, иметь атрибут, который можно будет прочесть из клиентского кода, но не изменить. В Python для этого есть кое-какие инструменты, в первую очередь *свойства*. Они позволяют весьма тонко управлять доступом к значениям атрибутов.

Знакомство с программой «Зверюшка со свойствами»

В программе «Зверюшка со свойствами» клиентскому коду удается выполнить чтение из атрибута, представляющего имя зверюшки – объекта класса `Critter`. Однако при попытке изменить значение, на которое ссылается этот атрибут, – в частности, заменить пустой строкой – программа «жалуется» и не позволяет выполнить правку. Результаты работы программы показаны на рис. 8.9.

Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 8. Файл называется `property_critter.py`.

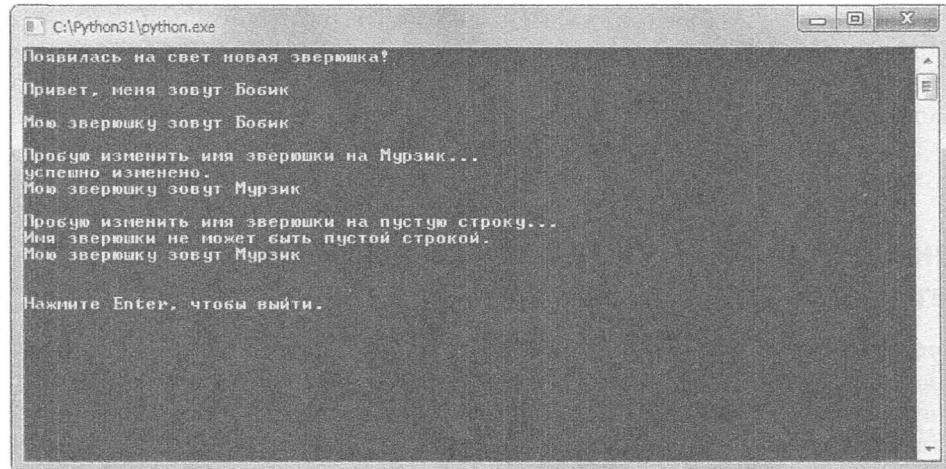


Рис. 8.9. Свойство управляет доступом к атрибуту name в экземпляре класса Critter

Создание свойств

Для управления доступом к закрытому атрибуту можно использовать *свойство* — объект с методами, которые позволяют косвенно обращаться к атрибутам и зачастую в чем-либо ограничивают такой косвенный доступ. Создав метод-конструктор класса *Critter*, я создаю свойство под названием *name*, через которое будет возможен непрямой доступ к закрытому атрибуту *_name*:

```

# Зверюшка со свойствами
# Демонстрирует свойства
class Critter(object):
    """Виртуальный питомец"""
    def __init__(self, name):
        print("Появилась на свет новая зверюшка!")
        self._name = name
    @property
    def name(self):
        return self._name

```

Чтобы создать свойство, я пишу метод, который возвращает интересующее меня значение (в данном случае *_name*), и предваряю объявление метода декоратором *@property*. Свойство одноименно методу; в нашем примере оно называется *name*. Теперь через свойство *name* любого объекта класса *Critter* можно узнавать значение закрытого атрибута *_name* этого объекта — неважно, внутри или вне объявления класса. Для этого применяется уже знакомая вам точечная нотация (примеры использования свойства *name* приведены в следующем подразделе «Доступ к свойствам»).

Вообще, чтобы создать свойство, надо написать метод, который будет возвращать интересующее значение, и предварить объявление метода декоратором *@property*. Имя свойства будет совпадать с именем метода. Если запрашиваемый

атрибут — закрытый, то принято называть свойство так же, как этот атрибут, но без начальных символов подчеркивания (как я и поступил).

Создавая свойство, мы «приоткрываем» закрытый атрибут и делаем его доступным для чтения, но не обязательно только для чтения. Можно разрешить запись в закрытый атрибут и даже установить ограничения на запись. Через свойство `name` я делаю закрытый атрибут `_name` доступным для записи с некоторыми оговорками:

```
@name.setter
def name(self, new_name):
    if new_name == "":
        print("Имя зверюшки не может быть пустой строкой.")
    else:
        self._name = new_name
        print("Имя успешно изменено.")
```

Этот код начинается с декоратора `@name.setter`. Обращаясь к атрибуту `setter` свойства `name`, я тем самым говорю, что метод, приводимый далее, устанавливает новое значение свойства `name`¹. Чтобы создать собственный декоратор для установки нового значения свойства, следуйте моему образцу: сначала символ @, потом имя свойства, потом точка и, наконец, слово `setter`.

За декоратором в моем коде следует метод `name`, который вызывается из клиентского кода при попытках присвоить новое значение скрытому атрибуту через свойство. Заметьте, что метод называется точно так же, как и свойство. Это общее условие: метод-сеттер должен носить то же имя, что и свойство.

При вызове метода параметру `new_name` передается значение, которое пользователь хочет сделать новым именем зверюшки. Если это будет пустая строка, то атрибут `_name` не изменится и программа известит пользователя о том, что попытка закончилась неудачей. В противном случае метод присваивает закрытому атрибуту `_name` значение `new_name` и сообщает, что зверюшка была успешно переименована. Как и в этом образце, у метода-сеттера обязательно должен быть параметр, в который клиентский код будет передавать новое значение.

Доступ к свойствам

После того как я создал свойство `name`, я могу узнать имя зверюшки, воспользовавшись точечной нотацией. Это и демонстрирует следующий фрагмент программы:

```
def talk(self):
    print("\nПривет, меня зовут". self.name)
# основная часть
crit = Critter("Бобик")
crit.talk()
```

Команда `self.name` запрашивает свойство `name` и таким образом вызывает метод, который возвращает атрибут `_name`. В данном случае будет возвращена строка

¹ Такие методы принято называть сеттерами не только в Python, но и в некоторых других языках программирования. — Примеч. пер.

"Бобик". Впрочем, свойством `name` можно пользоваться не только внутри объявления класса, но и в клиентском коде, как показано далее:

```
print("\nМою зверюшку зовут", end= " ")
print(crit.name)
```

Этот код находится вне класса `Critter`, но происходит, в сущности, то же самое: `crit.name` обращается к свойству `name` объекта `Critter` и косвенно вызывает метод, который возвратит значение `_name` (в данном случае по-прежнему "Бобик").

Теперь поменяем имя зверюшки:

```
print("\nПробую изменить имя зверюшки на Мурзик...")
crit.name = "Мурзик"
```

Конструкция с присвоением `crit.name = "Мурзик"` обращается к свойству `name` объекта и косвенно вызывает метод, который пробует установить новое значение атрибута `_name`. В данном случае методу передается ссылка на строку "Мурзик". Это не пустая строка, значит, атрибут `_name` примет значение "Мурзик", а на экране появятся слова Имя успешно изменено..

Теперь снова выведу имя зверюшки на экран с помощью свойства `name`:

```
print("Мою зверюшку зовут", end= " ")
print(crit.name)
```

Появится текст "Мою зверюшку зовут Мурзик".

Теперь предприму попытку заменить имя зверюшки на пустую строку:

```
print("\nПробую изменить имя зверюшки на пустую строку...")
crit.name = ""
```

Как и ранее, конструкция с присвоением обращается к свойству `name` объекта и косвенно вызывает метод, который пробует установить значение `_name`. Но поскольку параметру `new_name` этого метода передана ссылка на пустую строку, то метод только возвращает текст: Имя зверюшки не может быть пустой строкой., а значение атрибута `_name` объекта `crit` остается прежним. Чтобы показать, что все осталось на своих местах, выведу `name` в последний раз:

```
print("Мою зверюшку зовут", end= " ")
print(crit.name)
```

```
input("\n\nНажмите Enter, чтобы выйти.")
```

Компьютер скажет нам, что имя зверька – все еще Мурзик.

Вернемся к программе «Моя зверюшка»

В конечном варианте программы «Моя зверюшка» сочетаются фрагменты классов, уже изученных вами ранее в этой главе. Программа также имеет меню, посредством которого пользователь может взаимодействовать со своим компьютерным зверьком. Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 8. Файл называется `critter_caretaker.py`.

Класс Critter

Как вы помните, класс Critter — своего рода чертеж объекта, представляющего зверюшку. Этот класс несложен и в основном знаком вам, но его код занимает сравнительно много места и рассматривать его мы будем по частям.

Метод-конструктор

Метод-конструктор класса инициализирует три открытых атрибута объекта Critter: name, hunger и boredom. Заметьте, что у hunger и boredom устанавливаются значения по умолчанию, равные 0, так что сначала зверюшка находится в отличном настроении.

```
# Моя зверюшка
# Виртуальный питомец, о котором пользователь может заботиться
class Critter(object):
    """Виртуальный питомец"""
    def __init__(self, name, hunger = 0, boredom = 0):
        self.name = name
        self.hunger = hunger
        self.boredom = boredom
```

В этом методе я повел себя довольно легкомысленно и оставил атрибуты открытыми. Но я намерен реализовать и все те методы, которые могут понадобиться клиентскому коду для непрямого взаимодействия с экземпляром класса Critter.

Метод __pass_time()

Метод __pass_time() — закрытый метод, работа которого увеличивает уровень голода (hunger) и уныния (boredom) зверюшки. Этот метод вызывается каждый раз после того, как зверюшка ест, играет или беседует с хозяином; его задача — изобразить течение времени. Метод сделан закрытым потому, что его, согласно моему плану, должны вызывать лишь другие методы класса, ведь время в жизни зверюшки заметно только на каком-либо событийном фоне: кормление, игра, разговор.

```
def __pass_time(self):
    self.hunger += 1
    self.boredom += 1
```

Свойство mood

Свойство mood отражает самочувствие зверюшки. Это свойство вычисляется следующим образом: специальный метод находит сумму значений атрибутов hunger и boredom объекта Critter, после чего, исходя из получившегося числа, возвращает строку "прекрасно", "неплохо", "не сказать чтобы хорошо" или "ужасно".

Что интересно, свойство mood не дает доступа к закрытому атрибуту, потому что строка, выражаящая самочувствие зверюшки, не является частью экземпляра Critter, а формируется на лету. Свойство mood — инструмент доступа к значению, возвращаемому методом.

```
@property
def mood(self):
```

```

unhappiness = self.hunger + self.boredom
if unhappiness < 5:
    m = "прекрасно"
elif 5 <= unhappiness <= 10:
    m = "неплохо"
elif 11 <= unhappiness <= 15:
    m = "не сказать чтобы хорошо"
else:
    m = "ужасно"
return m Метод talk()

```

Метод `talk()` узнает значение свойства `mood` объекта класса `Critter` и сообщает о самочувствии зверюшки. После этого вызывается метод `__pass_time()`.

```

def talk(self):
    print("Меня зовут", self.name, ", и сейчас я чувствую себя", self.mood, "now.\n")
    self.__pass_time()

```

Метод eat()

Метод `eat()` уменьшает уровень голода зверюшки на величину, переданную параметру `food`. Если не передавать никакого значения, то по умолчанию `food` окажется равным 4. Программа следит за уровнем голода, не позволяя ему оказаться отрицательным числом. В конце, как обычно, вызывается `__pass_time()`.

```

def eat(self, food = 4):
    print("Мррр... Спасибо!")
    self.hunger -= food
    if self.hunger < 0:
        self.hunger = 0
    self.__pass_time()

```

Метод play()

Метод `play()` снижает уровень уныния зверюшки на величину, переданную параметру `fun`. Если не передавать никакого значения, то по умолчанию `fun` окажется равным 4. Программа следит за уровнем уныния зверюшки, не позволяя ему оказаться отрицательным числом. В конце, как обычно, вызывается `__pass_time()`.

```

def play(self, fun = 4):
    print("Уиии!")
    self.boredom -= fun
    if self.boredom < 0:
        self.boredom = 0
    self.__pass_time()

```

Создание зверюшки

Основную часть программы я вынес в самостоятельную функцию `main()`. В начале работы программы получает из пользовательского ввода имя зверюшки и инстанцирует класс `Critter`. Поскольку начальные значения `hunger` и `boredom` не присва-

иваются, при инициализации эти атрибуты становятся равными 0 и зверюшка появляется на свет в счастье и довольстве.

```
def main():
    crit_name = input("Как вы назовете свою зверюшку? ")
    crit = Critter(crit_name)
```

Создание меню

Теперь я создам уже знакомое вам меню. Если пользователь выбирает в нем вариант 0, программа завершает работу. Если введено 1, то к объекту применяется метод talk(). Если введено 2, то к объекту применяется метод eat(). Если же пользователь введет 3, то будет вызван метод play(). Наконец, любой другой пользовательский ввод рассматривается как некорректный.

```
choice = None
while choice != "0":
    print \
        """
    Моя зверюшка
    0 - Выйти
    1 - Узнать о самочувствии зверюшки
    2 - Покормить зверюшку
    3 - Поиграть со зверюшкой
    """
    choice = input("Ваш выбор: ")
    print()
    # выход
    if choice == "0":
        print("До свидания.")
    # беседа со зверюшкой
    elif choice == "1":
        crit.talk()
    # кормление зверюшки
    elif choice == "2":
        crit.eat()
    # игра со зверюшкой
    elif choice == "3":
        crit.play()
    # непонятный пользовательский ввод
    else:
        print("Извините, в меню нет пункта", choice)
```

Запуск программы

Следующая строка кода вызывает функцию main() и запускает программу. Наконец, последняя строка ожидает, пока пользователь выйдет из программы.

```
main()
input("\n\nНажмите Enter, чтобы выйти.")
```

Резюме

Эта глава познакомила вас с особой техникой программирования, которая пользуется программными объектами. Вы узнали, что в объекте могут сочетаться данные (называемые атрибутами) и функции (методы), что позволяет объектам успешно подражать реалиям внешнего мира. Вы научились писать классы — «чертежи» объектов. Вы получили представление о специальных методах-конструкторах, которые вызываются при создании новых объектов. Кроме того, вы увидели, как с помощью конструктора создавать и инициализировать атрибуты объекта. Вам было показано, как создавать атрибуты класса и статические методы, актуальные для целого класса, а не его отдельного экземпляра. Вы освоили технику инкапсуляции объектов — применение закрытых атрибутов и свойств. Кроме того, вы узнали, что хорошо устроен именно тот объект, который последовательно инкапсулирован. В конце главы вы применили все эти новые идеи и создали интерактивного домашнего зверька, который требует от пользователя непрестанного внимания и заботы.

ЗАДАЧИ

- Доработайте программу «Моя зверюшка» так, чтобы пользователь мог сам решить, сколько еды скормить зверюшке и сколько времени потратить на игру с ней (в зависимости от передаваемых величин зверюшка должна неодинаково быстро насыщаться и веселеть).
- Создайте программу, имитирующую телевизор как объект. У пользователя должна быть возможность вводить номер канала, а также увеличивать и уменьшать громкость. Программа должна следить за тем, чтобы номер канала и уровень громкости оставались в допустимых пределах.
- Создайте в программе «Моя зверюшка» своего рода «черный ход» — способ увидеть точные значения числовых атрибутов объекта. Сделайте в меню секретный пункт, который подсказка не будет отражать, и, если пользователь его выберет, выводите объект на экран (для этого в классе Critter должен быть реализован специальный метод `__str__()`).
- Напишите программу «Зооферма», в которой будет создано несколько объектов класса Critter, а манипулировать ими всеми можно будет с помощью списка. Теперь пользователь должен заботиться не об одной зверюшке, а обо всех обитателях зоофермы. Выбирая пункт в меню, пользователь выбирает действие, которое хотел бы выполнить со всеми зверюшками: покормить их, поиграть с ними или узнать об их самочувствии. Чтобы программа была интереснее, при создании каждой зверюшки следует назначать ей случайно выбранные уровни голода и уныния.

9 Объектно-ориентированное программирование. Игра «Блек-джек»

В предыдущей главе вы узнали о программных объектах. Хотя бы по одному объекту используется почти во всех программах, разбор которых дан в книге. Таким образом, вы успешно начали постигать премудрости ООП, но всю мощь этого подхода оценит лишь тот, кто увидит, как объекты взаимодействуют друг с другом. Эта глава научит вас создавать множественные объекты и устанавливать отношения между ними.

Вам предстоит узнать, как делать следующее:

- создавать объекты разных классов в составе одной программы;
- разрешать объектам взаимодействовать друг с другом;
- на основе более простых объектов создавать более сложные;
- производить новые классы от существующих;
- расширять объявления существующих классов;
- переопределять методы в существующих классах.

Знакомство с игрой «Блек-джек»

Проект, над разработкой которого мы потрудимся в этой главе, представляет собой упрощенную версию карточной игры «Блек-джек»¹. Игровой процесс идет так: участники получают карты, с которыми связаны определенные числовые значения — очки, и каждый участник стремится набрать 21 очко, но не больше. Количество очков, соответствующих карте с цифрой, равно ее номиналу; валет, дама и король идут за 10 очков, а туз — за 1 или 11 (в зависимости от того, как выгоднее для игрока).

Компьютер сдает карты² и играет против нескольких игроков, от одного до семи. В начале каждого раунда компьютер передает каждому из участников, в том числе

¹ Вариант этой игры известен у нас под названием «Очко». — Примеч. пер.

² В «Блек-джеке» принято называть сдающего дилером. — Примеч. пер.

и себе, по две карты. Игрокам видны карты друг друга, и даже автоматически подсчитывается текущая сумма очков на руках у каждого. Впрочем, из двух карт, которые дилер сдал себе, одна до поры до времени лежит рубашкой вверх.

Затем каждому игроку по очереди предоставляется возможность тянуть дополнительные карты. Игрок может брать их из перемешанной колоды до тех пор, пока ему угодно и пока сумма очков на руках у него не превысила 21. При превышении, которое называется перебором, участник проигрывает. Если все перебрали, то компьютер выводит свою вторую карту и начинает новый раунд. Если же один или несколько участников остались в игре, то раунд еще не закончен. Дилер открывает свою вторую карту и, по общему правилу «Блек-джека», тянет дополнительные карты для себя до тех пор, пока сумма его очков не будет равна 17 или больше. Если дилер, в нашем случае — компьютер, совершил перебор, то победу одерживают все участники, оставшиеся в игре. Если нет, то сумма очков каждого из участников сравнивается с очками, которые набрал компьютер. Набравший больше очков (человек или компьютер) побеждает. При одинаковой сумме очков объявляется ничья между компьютером и одним или несколькими участниками.

Игровой процесс отражен на рис. 9.1.

```
C:\Python31\python.exe
Сколько всего игроков? <1 - ?>: 2
Введите имя игрока: Larry
Введите имя игрока: Jerry
Larry: 8d      2d      <10>
Jerry: Qh      Js      <20>
Dealer: Kh      Ks

Larry, будете брать ещё карты? <Y/N>: y
Larry: 8d      2d      Jc      <20>

Larry, будете брать ещё карты? <Y/N>: y
Larry: 8d      2d      Js      10s    <30>
Larry перебрал.
Larry проиграл.

Jerry, будете брать ещё карты? <Y/N>: n
Dealer: 2c      Ks      <12>
Dealer: 2c      Ks      Kd      <22>
Dealer перебрал.
Jerry выиграл.

Хотите сыграть ещё раз? -
```

Рис. 9.1. Один из игроков выиграл этот раунд, а остальным не повезло

Отправка и прием сообщений

Объектно-ориентированная программа — это своего рода экологическая система, в которой объекты — живые организмы. Чтобы поддерживать биоценоз в порядке, его обитатели должны взаимодействовать; так же происходит и в ООП. Программа не может быть полезна, если объекты в ней не взаимодействуют каким-либо удачно заданным способом. В терминах ООП такое взаимодействие называется *отправкой сообщений*. На практике объекты всего лишь вызывают методы друг друга. Это хотя и не совсем вежливо, но, во всяком случае, лучше, чем если бы объектам пришлось напрямую обращаться к атрибутам друг друга.

Знакомство с программой «Гибель пришельца»

Программа изображает ситуацию из компьютерной игры, в которой игрок стреляет в инопланетного агрессора. Все буквально так и происходит: игрок стреляет, а пришелец умирает (но, правда, успевает напоследок произнести несколько высокопарных слов). Это реализовано с помощью посылки сообщения от одного объекта другому. Результат работы программы отражен на рис. 9.2.

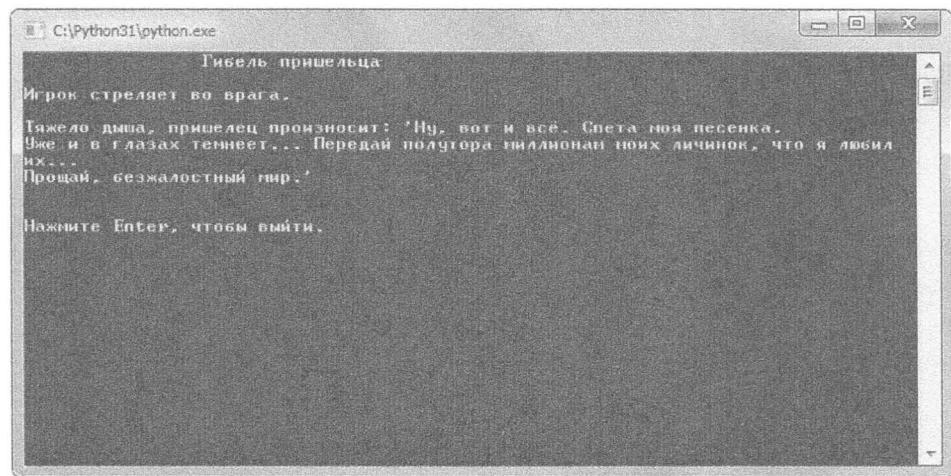


Рис. 9.2. Объекты обмениваются сообщениями, а на экране идет бой

Говоря технически, программа создает `hero` — экземпляр класса `Player` и `invader` — экземпляр класса `Alien`. При вызове метода `blast()` объекта `hero` с аргументом `invader` этот объект вызывает метод `die()` объекта `invader`. Другими словами, когда герой стреляет в пришельца, это значит, что объект «герой» посыпает объекту «пришелец» сообщение с требованием умереть. Этот обмен сообщениями показан на рис. 9.3.

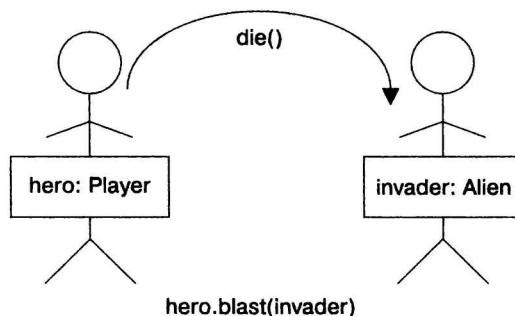


Рис. 9.3. Объект `hero` класса `Player` посыпает сообщение объекту `invader` класса `Alien`

НА САМОМ ДЕЛЕ

Диаграмма, которую я создал, чтобы проиллюстрировать обмен сообщениями между двумя объектами, очень проста. При большом количестве объектов и отношений между ними диаграммам свойственно значительно усложняться. Для графического представления IT-проектов есть множество математизированных методов. Один из наиболее популярных инструментов — UML (Унифицированный язык моделирования), язык записи диаграмм, который особенно полезен для наглядного представления объектно-ориентированных систем.

Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 9. Файл называется alien_blaster.py:

```
# Гибель пришельца
# Демонстрирует взаимодействие объектов
class Player(object):
    """ Игрок в экшен-игре. """
    def blast(self, enemy):
        print("Игрок стреляет во врага.\n")
        enemy.die()
class Alien(object):
    """ Враждебный пришелец-инопланетянин в экшен-игре. """
    def die(self):
        print("Тяжело дыша, пришелец произносит: 'Ну, вот и все. Спата моя песенка.\n'")
        print("Уже и в глазах темнеет... Передай полутора миллионам моих личинок, что я любил их...\n")
        print("Прощай, безжалостный мир.'")
# Основная часть программы
print("\ttГибель пришельца\n")
hero = Player()
invader = Alien()
hero.blast(invader)
input("\n\nНажмите Enter, чтобы выйти.")
```

Отправка сообщения

Прежде чем один объект сможет послать сообщение другому, надо, чтобы объектов было два! Столько их и создано в основной части моей программы: сначала объект класса `Player`, с которым связывается переменная `hero`, а потом объект класса `Alien`, с которым связывается переменная `invader`.

Интересное происходит в строке кода, следующей за этими двумя. Командой `hero.blast(invader)` язываю метод `blast()` объекта `hero` и передаю ему как аргумент объект `invader`. Изучив объявление `blast()`, вы можете увидеть, что этот метод принимает аргумент в параметр `enemy`. Поэтому при вызове `blast()` его внутренняя переменная `enemy` ссылается на объект класса `Alien`. Выведя на экран текст, метод `blast()` командой `enemy.die()` вызывает метод `die()` объекта `Alien`. Таким образом, по существу, экземпляр класса `Player` посыпает экземпляру класса `Alien` сообщение, которым вызывает его метод `die()`.

Прием сообщения

Объект `invader` принимает сообщение от объекта `hero` закономерным образом: вызывается метод `die()` и пришелец умирает, сказав самому себе душераздирающее надгробное слово.

Сочетание объектов

Обычно в жизни сложные вещи строятся из более простых. Так, гоночную машину можно рассматривать как единый объект, который, однако, составлен из других, более простых объектов: корпуса, двигателя, колес и т. д. Иногда встречается важный частный случай: объекты, представляющие собой наборы других объектов. Таков, например, зоопарк, который можно представить как набор животных. Эти типы отношений возможны и между программными объектами в ООП. К примеру, ничто не мешает написать класс `Drag_Racer`, представляющий гоночный автомобиль; у объектов этого класса будет атрибут `engine`, ссылающийся на объект `Race_Engine` (двигатель). Можно написать и класс `Zoo`, представляющий зоопарк, у объектов которого будет атрибут `animals` — список животных (объектов класса `Animal`). Сочетание объектов, как в этих примерах, позволяет строить сложные объекты из простых.

Знакомство с программой «Карты»

В программе «Карты» объекты представляют отдельные игральные карты, которыми можно воспользоваться для любой из игр от «Блек-джека» до «Подкидного дурака» (в зависимости от того, каковы ваши вкусы и денежные активы). Далее в той же программе строится объект «рука» (`Hand`), представляющий набор карт одного игрока; это не что иное, как список объектов-карт. Результат работы программы показан на рис. 9.4.

```
C:\Python31\python.exe
Вывожу на экран объект-карту:
Ас

Вывожу еще четыре карты:
2с
3с
4с
5с

Печатаю карты, которые у меня на руках до раздачи:
<пусто>

Печатаю пять карт, которые появились у меня на руках:
Ас 2с 3с 4с 5с

Первые две из моих карт я передал вам.
Теперь у вас на руках:
Ас 2с

У меня на руках:
3с 4с 5с

У меня на руках после того, как я сбросил все карты:
<пусто>

Нажмите Enter, чтобы выйти.
```

Рис. 9.4. Каждый объект класса `Hand` — коллекция объектов класса `Card`

Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 9. Файл называется `playing_cards.py`.

Создание класса Card

Первым делом в этой программе я создаю класс `Card`, объекты которого будут представлять игральные карты.

```
# Карты
# Демонстрирует сочетание объектов
class Card(object):
    """ Одна игральная карта. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

У каждого объекта класса `Card` есть атрибут `rank`, значение которого — достоинство карты. Атрибут класса `RANKS` содержит все возможные значения: туз ("A"), карты с номиналами от 2 до 10, валет ("J"), дама ("Q") и король ("K"). У каждой карты есть также атрибут `suit`, представляющий масть карты. Все его возможные значения содержит атрибут класса `SUITS`: "c" (clubs) — трефы, "d" (diamonds) — бубны, "h" (hearts) — червы и, наконец, "s" (spades) — пики.

Например, объект со значением `rank`, равным "A", и значением `suit`, равным "d", представляет бубновый туз. Значения этих двух атрибутов, соединенных в единую строку, возвращает для вывода на печать специальный метод `__str__()`.

Создание класса Hand

Следующее, что я должен сделать в программе, — создать класс `Hand`, экземпляры которого будут представлять наборы объектов-карт:

```
class Hand(object):
    """ 'Рука': набор карт на руках у одного игрока. """
    def __init__(self):
        self.cards = []
    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + " "
        else:
            rep = "<пусто>"
        return rep
```

```
def clear(self):
    self.cards = []
def add(self, card):
    self.cards.append(card)
def give(self, card, other_hand):
    self.cards.remove(card)
    other_hand.add(card)
```

У нового объекта класса `Hand` появляется атрибут `cards`, представляющий собой список карт. Таким образом, атрибут единичного объекта — список, который может содержать сколь угодно много других объектов.

Специальный метод `__str__()` возвращает одной строкой всю «руку». Метод последовательно берет все объекты класса `Card` и соединяет их строковые представления. Если в составе объекта `Hand` нет ни одной карты, то будет возвращена строка <пусто>.

Метод `clear()` очищает список карт: атрибут «руки» `cards` приравнивается к пустому списку. Метод `add()` добавляет объект к списку `cards`. Метод `give()` удаляет объект из списка `cards`, принадлежащего данной «руке», и добавляет тот же объект в набор карт другого объекта класса `Hand` (для этого вызывается его метод `add()`). Иными словами, первый объект `Hand` посыпает второму объекту `Hand` сообщение: добавить в атрибут `cards` данный объект `Card`.

Применение объектов-карт

В основной части программы я создаю и вывожу на экран пять объектов класса `Card`:

```
# основная часть
card1 = Card(rank = "A", suit = "c")
print("Вывожу на экран объект-карту:")
print(card1)
card2 = Card(rank = "2", suit = "c")
card3 = Card(rank = "3", suit = "c")
card4 = Card(rank = "4", suit = "c")
card5 = Card(rank = "5", suit = "c")
print("\nВывожу еще четыре карты:")
print(card2)
print(card3)
print(card4)
print(card5)
```

У первого из созданных экземпляров класса `Card` атрибут `rank` равен "A", а атрибут `suit` — "c" (туз треф). На экране этот объект отображается в виде Ac; вид других карт аналогичен.

Сочетание объектов-карт в объекте `Hand`

Теперь я создам экземпляр класса `Hand`, свяжу его с переменной `my_hand` и выведу информацию о нем на экран:

```
my_hand = Hand()
print("\nПечатаю карты, которые у меня на руках до раздачи:")
print(my_hand)
```

Поскольку атрибут cards этого объекта пока равен пустому списку, на экране будет напечатано <пусто>.

Добавлю в my_hand пять объектов класса Card и снова выведу объект на экран:

```
my_hand.add(card1)
my_hand.add(card2)
my_hand.add(card3)
my_hand.add(card4)
my_hand.add(card5)
print("\nПечатаю пять карт, которые появились у меня на руках:")
print(my_hand)
```

На экране отобразится текст Ac 2c 3c 4c 5c.

А сейчас я создам еще один экземпляр класса Hand под названием your_hand. Применив к my_hand метод give(), передам из «своей руки» в «вашу руку» две карты и затем выведу содержимое обеих «рук» на экран:

```
your_hand = Hand()
my_hand.give(card1, your_hand)
my_hand.give(card2, your_hand)
print("\nПервые две из моих карт я передал вам.")
print("Теперь у вас на руках:")
print(your_hand)
print("А у меня на руках:")
print(my_hand)
```

Как и следовало ожидать, your_hand имеет вид Ac 2c, а my_hand — 3c 4c 5c.

В конце программы я вызову метод clear() объекта my_hand и напечатаю его на экране еще раз:

```
my_hand.clear()
print("\nУ меня на руках после того, как я сбросил все карты:")
print(my_hand)
input("\n\nНажмите Enter, чтобы выйти.")
```

Закономерно отображается <пусто>.

Создание новых классов с помощью наследования

Одна из ключевых идей ООП — *наследование*. Благодаря ему можно основать вновь создаваемый класс на существующем, причем новый класс автоматически получит (унаследует) все методы и атрибуты старого. Это как если бы за весь тот объем работы, который ушел на создание первоначального класса, не пришлось платить!

ЛОВУШКА

В Python можно создавать новый класс, непосредственно основанный на нескольких классах; это так называемое множественное наследование. Но при множественном наследовании возникают кое-какие трудности, и начинающему программисту лучше пока без него обойтись.

Расширение класса через наследование

Наследование особенно полезно тогда, когда программист хочет создать более узкую и специализированную версию готового класса. Как вы уже знаете, наследующий класс приобретает все методы и атрибуты родительского. Но ведь ничто не мешает добавить собственные методы и атрибуты для расширения функциональности объектов этого нового класса.

Вообразим, например, что ваш класс `Drag_Racer` задает объект — гоночный автомобиль с методами `stop()` и `go()`. Можно на его основе создать новый класс, описывающий особый тип гоночных машин с функцией очистки ветрового стекла (на скорости до 400 км/ч о стекло будут все время биться насекомые). Этот новый класс автоматически унаследует методы `stop()` и `go()` от класса `Drag_Racer`. Значит, вам останется объявить всего один новый метод для очистки ветрового стекла, и класс будет готов.

Знакомство с программой «Карты 2.0»

Программа «Карты 2.0» — новая версия программы «Карты», где для описания колоды игральных карт вводится новый класс `Deck` (колода). В отличие от других классов, которые вам приходилось ранее создавать, `Deck` основан на уже существующем классе `Hand`. В результате `Deck` автоматически наследует все методы `Hand`. Я реализовал `Deck` таким образом потому, что колода — это, в сущности, специализированный набор карт на руках, «рука», но с дополнительными функциями. Колоде свойственно все, что свойственно «руке»: это набор карт, который может передавать карту любой «руке» и т. д. Однако колода также обладает такими формами поведения, которые неизвестны «руке». Во-первых, ее можно перемешивать, а во-вторых, из нее можно раздать карты на руки нескольким лицам. В «Картах 2.0» будет создана колода, из которой карты раздаются в две «руки».

Результаты работы программы показаны на рис. 9.5. Попробовав запустить программу, вы, скорее всего, увидите на экране другую комбинацию карт, потому что тасование колоды выполняется случайным образом.

Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 9. Файл называется `playing_cards2.py`.

Создание базового класса

Новая программа начинается с того же, с чего и ее более ранняя версия. Первые два класса — `Card` и `Hand` — приведены здесь почти без отличий:

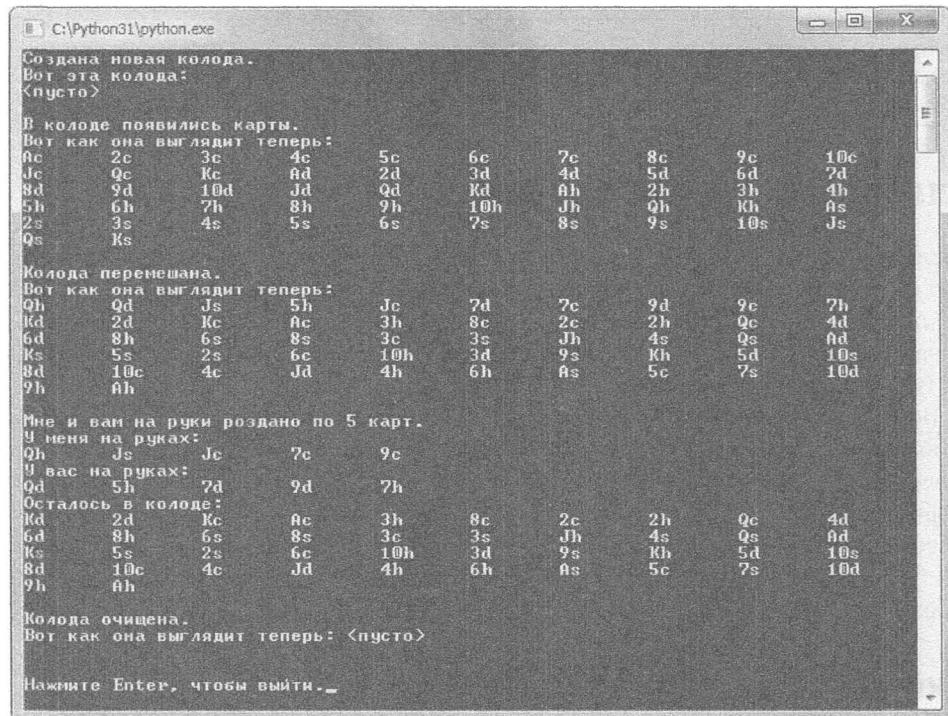


Рис. 9.5. Класс Deck наследует все методы класса Hand

```
# Карты 2.0
# Демонстрирует расширение класса через наследование
class Card(object):
    """ Одна игральная карта. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
    def __str__(self):
        rep = self.rank + self.suit
        return rep
class Hand(object):
    """ 'Рука': набор карт на руках у одного игрока. """
    def __init__(self):
        self.cards = []
    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = "<пусто>"
        return rep
```

```

    return rep
def clear(self):
    self.cards = []
def add(self, card):
    self.cards.append(card)
def give(self, card, other_hand):
    self.cards.remove(card)
    other_hand.add(card)

```

Наследование от базового класса

Следующим шагом в этой программе я создам класс Deck. Из заголовка класса можно видеть, что этот класс основан на Hand:

```
class Deck(Hand):
```

В данном случае Hand является *базовым классом*, а Deck — *производным*, так как часть его объявления «произведена» от объявления Hand. Вследствие таких взаимоотношений Deck наследует все методы Hand. Значит, даже если бы я не объявил в этом классе больше ни одного нового метода, экземплярам класса Deck были бы по-прежнему доступны следующие методы «руки»:

- `__init__();`
- `__str__();`
- `clear();`
- `add();`
- `give().`

Можно даже вообразить, что благодаря наследованию все методы Hand как бы скопированы и вставлены в объявление класса Deck.

Расширение производного класса

Производный класс можно расширять, задав в нем дополнительные методы. Это я и делаю в объявлении Deck:

```

""" Колода игральных карт. """
def populate(self):
    self.cards = []
    for suit in Card.SUITS:
        for rank in Card.RANKS:
            self.add(Card(rank, suit))
def shuffle(self):
    import random
    random.shuffle(self.cards)
def deal(self, hands, per_hand = 1):
    for rounds in range(per_hand):
        for hand in hands:
            if self.cards:
                top_card = self.cards[0]

```

```

    self.give(top_card, hand)
else:
    print("Не могу больше сдавать: карты закончились!")
)

```

Итак, в добавок ко всем методам, унаследованным колодой от «руки», реализованы еще и следующие новые методы:

- populate();
- shuffle();
- deal().

С точки зрения клиентского кода, все методы Deck совершенно равноправны: и унаследованные от Hand, и реализованные в самом классе Deck. Вызывать их по отношению к объекту класса Deck можно уже привычным для нас образом, с помощью точечной нотации.

Применение производного класса

В основной части программы я сначала создаю новый объект класса Deck:

```
# основная часть
deck1 = Deck()
```

Взглянув на код класса, вы заметите, что метод-конструктор для объектов Deck не был объявлен. Но Deck наследует конструктор из класса Hand, который и будет вызван по отношению к новому экземпляру Deck. Как следствие, у новой колоды атрибут cards инициализируется в виде пустого списка; так произошло бы и вообще с любой вновь созданной «рукой». В заключение этого фрагмента кода новый объект связывается с переменной deck1.

Теперь у меня есть новая колода, правда, пустая. Выведу ее на экран:

```
print("Создана новая колода.")
print("Вот эта колода:")
print(deck1)
```

Опять же специальный метод `__str__()` в классе Deck не объявлен, но унаследован от Hand. Поскольку колода не содержит ни одной карты, на экране появится текст `<пусто>`. Вплоть до сих пор колода по своему поведению ничем не отличалась от «руки». Но она ведь и есть специализированная «рука», обладающая всеми теми же возможностями плюс еще некоторыми.

Пустая колода — это неинтересно. Вызову метод `populate()` объекта `deck1`, который поместит в нашу колоду традиционные 52 карты:

```
deck1.populate()
```

Теперь в ход пошла функциональность, неизвестная «руке»: метод `populate()` — новый метод, объявленный в классе Deck. Этот метод перебирает все 52 возможных сочетания масти и номинала (элементов `Card.SUITS` и `Card.RANKS`); для каждого такого сочетания метод создает новый объект `Card`, карту соответствующей масти и номинала и добавляет в колоду.

Снова напечатаю колоду на экране:

```
print("\nВ колоде появились карты.")  
print("Вот как она выглядит теперь:")  
print(deck1)
```

На этот раз выводятся все 52 карты. Но если внимательно присмотреться, то станет ясно, что порядок карт — удручающе стройный. Чтобы стало интереснее, перемешаем колоду:

```
deck1.shuffle()
```

В классе Deck объявлен метод `shuffle()`, который импортирует модуль `random` и применяет метод `random.shuffle()` к списку `cards` данного экземпляра класса. Как несложно догадаться, метод `random.shuffle()` возвращает тот же список, элементы которого, однако, переставлены в случайном порядке. Таким образом, все карты в колоде перемешаны, — отлично.

После установления случайного порядка карт выведем нашу колоду на экран:

```
print("\nКолода перемешана.")  
print("Вот как она выглядит теперь:")  
print(deck1)
```

Вслед за этим создадим два объекта класса `Hand`, поместим их в список и сделаем этот список значением переменной `hands`:

```
my_hand = Hand()  
your_hand = Hand()  
hands = [my_hand, your_hand]
```

А сейчас раздадим в каждую «руку» по пять карт:

```
deck1.deal(hands, per_hand = 5)
```

Метод `deal()` — новый, объявленный в классе `Deck`. Он принимает два аргумента: список «рук» и количество карт, которые следует раздать в каждую «руку». Метод передает из колоды в каждую «руку» по одной карте. Если в колоде не хватает карт, метод сообщает: Не могу больше сдавать: карты закончились!. Этот цикл повторяется столько раз, по сколько карт должно быть сдано на руки каждому игроку. Значит, предшествующая строка кода передает по пять карт из колоды `deck1` в «руки» `my_hand` и `your_hand`.

Чтобы увидеть результаты сдачи, опять выведем на экран колоду и обе «руки»:

```
print("\nМне и вам на руки раздано по 5 карт.")  
print("У меня на руках:")  
print(my_hand)  
print("У вас на руках:")  
print(your_hand)  
print("Осталось в колоде:")  
print(deck1)
```

Взглянув на результаты, можно удостовериться, что в каждой «руке» по пять карт, а в колоде осталось 42.

Возвращу наконец колоду в ее первоначальное пустое состояние:

```
deck1.clear()
print("\nКолода очищена.")
```

И выведу ее в последний раз на экран:

```
print("Вот как она выглядит теперь:", deck1)
input("\n\nНажмите Enter, чтобы выйти.")
```

Переопределение унаследованных методов

Вы узнали, как можно расширить производный класс по сравнению с базовым, добавляя в него новые методы. Оказывается, в производном классе можно, кроме того, изменить работу унаследованного метода. Это так называемое *переопределение* метода. Переопределяя метод базового класса, вы стоите перед выбором: или создать метод с совершенно новой функциональностью, или основать новые функции на уже имеющихся.

Для примера вновь возьмем класс `Drag_Racer`. Пусть, например, при вызове метода `stop()` у автомобиля просто срабатывают тормоза. Чтобы создать новый класс, описывающий гоночные автомобили, которые останавливаются более эффективно — с помощью тормозного парашюта, — можно произвести от `Drag_Racer` класс `Parachute_Racer` с переопределенным методом `stop()`. Удобно написать новый метод `stop()`, который будет вызывать метод `stop()` базового класса `Drag_Racer` (и, значит, срабатывание тормозов), а вдобавок описывать выброс тормозного парашюта.

Знакомство с программой «Карты 3.0»

В программе «Карты 3.0» от класса `Card`, хорошо знакомого вам, будут произведены два новых класса игральных карт. Первый описывает карты, масти и номиналы которых нельзя выводить на экран. Говоря более точно, при попытке напечатать объект такого класса на экране появится текст <нельзя напечатать>. Другой класс будет описывать карты, лежащие рубашкой вверх и вниз. При выводе объектов этого класса на печать возможны два исхода. Если карта лежит лицевой стороной вверх, она будет отображена как любой экземпляр класса `Card`. Но если карта расположена лицевой стороной вниз, то вместо номинала и масти отобразится XX. Пробный пуск программы показан на рис. 9.6.

Я буду разбирать код небольшими порциями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 9. Файл называется `playing_cards3.py`.

Создание базового класса

Для того чтобы произвести класс, сначала нужно создать базовый класс. В данной программе в качестве базового используется класс `Card`, с которым вы, надо полагать, уже сроднились душой:

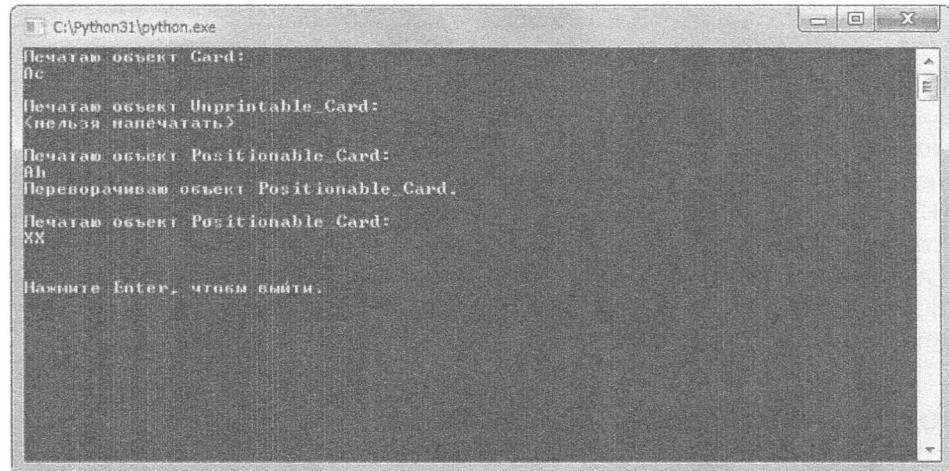


Рис. 9.6. В двух производных классах переопределен унаследованный метод `__str__()`, из-за чего объекты этих классов по-разному отображаются на экране

```
# Карты 3.0
# Демонстрирует наследование в части переопределения методов
class Card(object)1:
    """ Одна игральная карта. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
    def __str__(self):
        rep = self.rank + self.suit
        return rep
```

Переопределение методов базового класса

Я произведу от `Card` новый класс, объектами которого будут непечатаемые (нераскрытие) карты. Заголовок класса выглядит обычным образом:

```
class Unprintable_Card(Card):
```

Из этого заголовка видно, что класс `Unprintable_Card` наследует все методы класса `Card`. Но поведение унаследованного метода можно изменить, если дополнитель но объявить его и в производном классе. Именно так я и поступил далее в коде класса:

```
""" Карта, номинал и масть которой не могут быть выведены на экран. """
def __str__(self):
    return "<нельзя напечатать>"
```

¹ В Python 3 нет смысла наследоваться от `object`, как это было во 2-й версии. Все классы уже ведут себя так, как если бы были унаследованы от `object` во 2-й версии. По факту они и так наследуются от `object`. — Примеч. науч. ред.

Класс `Unprintable_Card` наследует метод `__str__()` от класса `Card`. Но в `Unprintable_Card` объявляется собственный метод `__str__()`, который переопределяет (в данном случае просто замещает) унаследованный метод. Вообще каждый раз, когда в производном классе объявляется метод с тем же именем, что и уже готовый, унаследованный, этот унаследованный метод переопределется. Вот почему при попытке напечатать объект `Unprintable_Card` на экране появится текст `<нельзя напечатать>`.

Изменения в производном классе никак не скажутся на базовом классе, которому совершенно все равно, унаследуют ли от него что-либо и будет ли унаследованное переопределено. Базовый класс будет функционировать по-прежнему. Иными словами, при выводе на экран игральной карты — экземпляра класса `Card` — она будет появляться как обычно.

Вызов методов базового класса

Иногда при переопределении метода базового класса хочется «встроить» его унаследованную функциональность в новую. К примеру, я хочу создать особый тип игральных карт на основе класса `Card`. Я решил, что у объектов этого класса должен быть атрибут, показывающий, вверх или вниз лицевой стороной лежит карта. Значит, мне нужно переопределить метод-конструктор, унаследованный от `Card`, так, чтобы новый метод создавал при инициализации объекта атрибут `is_face_up`. Вместе с тем я хотел бы, чтобы конструктор, как и ранее, устанавливал значения атрибутов `rank` и `suit`. Переписывать код из конструктора объекта `Card` не придется: этот метод можно вызвать из моего нового конструктора, и он позаботится о том, чтобы создать и инициализировать атрибуты `rank` и `suit` у объекта нового класса. После этого новый конструктор создаст атрибут, определяющий положение карты лицом или рубашкой вверх.

Именно так все сделано в классе `Positionable_Card`:

```
class Positionable_Card(Card):
    """ Карта, которую можно положить лицом или рубашкой вверх. """
    def __init__(self, rank, suit, face_up=True):
        super(Positionable_Card, self).__init__(rank, suit)
        self.is_face_up = face_up
```

Функция `super()` в этом конструкторе позволяет вызвать метод базового класса, называемого также *надклассом*. Код `super(Positionable_Card, self).__init__(rank, suit)`¹ вызывает метод `__init__()` из класса `Card`, который по отношению к `Positionable_Card` является надклассом. Дальнейшая часть этой команды — `__init__(rank, suit)` — предлагает интерпретатору вызвать метод-конструктор базового класса с теми аргументами, которые будут переданы в параметры `rank` и `suit`.

Следующий метод в классе `Positionable_Card` также переопределяет (и притом вызывает) метод, унаследованный от `Card`:

```
def __str__(self):
    if self.is_face_up:
        rep = super(Positionable_Card, self).__str__()
```

¹ Можно спокойно использовать `super()` без параметров. — Примеч. науч. ред.

```

else:
    rep = "XX"
return rep

```

Этот метод `__str__()` сначала проверяет на истинность атрибут объекта `is_face_up`. Если его значение равно `True`, значит, карта лежит лицевой стороной вверх. Тогда строковое представление карты приравнивается к строке, которую возвращает метод `__str__()` из класса `Card`, примененный к объекту производного класса `Positionable_Card`. Другими словами, если карта открыта, то ее масть и номинал можно вывести на экран, как масть и номинал любого экземпляра класса `Card`. Однако, если карта закрыта, на экране может быть показано только `XX`.

Последний метод в классе не переопределяет ни одного из унаследованных методов, а только расширяет новый класс:

```

def flip(self):
    self.is_face_up = not self.is_face_up

```

Этот метод переворачивает карту, изменяя значение атрибута `is_face_up` у объекта на противоположное. Если оно было равно `True`, то после вызова метода `flip()` окажется равным `False`, и наоборот, если было равным `False`, вызов метода `flip()` установит его равным `True`.

Применение производного класса

В основной части программы я создаю три объекта: один — класса `Card`, другой — класса `Unprintable_Card`, наконец, третий — класса `Positionable_Card`:

```

# основная часть
card1 = Card("A", "c")
card2 = Unprintable_Card("A", "d")
card3 = Positionable_Card("A", "h")

```

Теперь выведем на экран экземпляр класса `Card`:

```

print("Печатаю объект Card:")
print(card1)

```

Как и в предыдущей программе, будет выведен текст `Ac`.

Затем я пробую вывести на экран экземпляр класса `Unprintable_Card`:

```

print("\nПечатаю объект Unprintable_Card:")
print(card2)

```

Несмотря на то что здесь атрибут `rank` имеет значение `"A"`, а атрибут `suit` — значение `"d"`, попытка окажется неудачной: на экран будет выведено `<нельзя напечатать>`, потому что в классе `Unprintable_Card` переопределен унаследованный метод `__str__()` и теперь он всегда возвращает строку `"<нельзя напечатать>"`.

Следующие две строки кода посвящены экземпляру класса `Positionable_Card`:

```

print("\nПечатаю объект Positionable_Card:")
print(card3)

```

Поскольку по умолчанию карта лежит лицом вверх (`is_face_up = True`), метод `_str_()` данного объекта вызывает метод `_str_()` объекта `Card`, в результате чего на экране появится `Ah`.

Теперь я вызову метод `flip()` объекта `Positionable_Card`:

```
print("Переворачиваю объект Positionable_Card.")
card3.flip()
```

Как следствие, атрибут `is_face_up` окажется ложным. Следующие две строки вновь пытаются напечатать объект `Positionable_Card`:

```
print("\nПечатаю объект Positionable_Card:")
print(card3)
input("\n\nНажмите Enter, чтобы выйти.")
```

Но на этот раз отображается только `XX`, потому что карту перевернули рубашкой вверх.

Что такое полиморфизм

Когда предметы одной группы неодинаково реагируют на одну и ту же операцию, такое поведение называется *полиморфным*. В контексте ООП полиморфизм означает, что одно и то же сообщение можно посыпать объектам разных классов, один из которых наследуется от другого, и получать разное поведение у наследников и предков. Например, класс `Unprintable_Card` произведен от `Card`, но, если вызвать метод `_str_()` объекта `Unprintable_Card`, результат будет отнюдь не тем же, каким был бы при вызове метода `_str_()` объекта `Card`. Как следствие полиморфного поведения, объект можно выводить на экран, даже не зная наверняка, какого он класса: `Unprintable_Card` или просто `Card`. Вне зависимости от класса для печати на экране будет вызван метод `_str_()`, который отобразит правильное строковое представление.

В контексте ООП полиморфизм означает, что одно и то же сообщение можно посыпать объектам разных классов, один из которых наследуется от другого, и получать разное поведение у наследников и предков.

Создание модулей

О модулях вы впервые узнали в подразделе «Импорт модуля `random`» раздела «Вернемся к игре «Отгадай число»» главы 3. Особая мощь программирования на Python в том, что создавать, применять и даже распространять свои собственные модули ничуть не сложно. Создание собственных модулей даже дает несколько важных преимуществ.

Во-первых, создав свой модуль, вы сможете повторно задействовать код, что сэкономит вам время и усилия. Так, например, с использованием классов `Card`, `Hand` и `Deck` можно реализовать много разных карточных игр, в которых не придется все время изобретать заново базовую функциональность карты, колоды и «руки» игрока.

Во-вторых, если большая программа разбита на логические модули, то с ее кодом легче работать. До сих пор рассмотренные нами программы состояли из одного файла; это и неудивительно, ведь все это короткие программы. Но вообразите программу длиной в несколько тысяч или даже десятков тысяч строк (профессиональные проекты часто достигают таких размеров). Управлять работой всего этого кода, если он помещен в один большой файл, очень сложно. А вот если разбить код на модули, то каждый программист в IT-команде сможет взять на себя разработку или правку отдельной части проекта.

В-третьих, создавая модули, вы делитесь своим трудом и вдохновением. Разработав полезный модуль, вы можете, например, отправить его по электронной почте другу, который сможет пользоваться им с не меньшим удобством, чем любым встроенным модулем Python.

Знакомство с программой «Простая игра»

Эта программа, как подсказывает ее название, очень проста и незатейлива. Она сначала спрашивает, сколько всего игроков желает поучаствовать в игре, затем просит ввести имена игроков и, наконец, присваивает каждому игроку случайное количество очков, после чего выводит результат. Не слишком впечатляюще; но вся суть программы — не сама игра, а ее реализация и внутренняя механика. В программе используется новый модуль с функциями и классом, которые я специально написал. Результаты работы программы показаны на рис. 9.7.

```
C:\Python31\python.exe
Добро пожаловать в самую-самую простую игру!
Сколько игроков участвует? <2 - 5>: 2
Имя игрока: Fred
Имя игрока: Bartley
Вот результаты игры:
Fred: 26
Bartley: 20
Хотите сыграть еще раз? <y/n>: 
```

Рис. 9.7. Используемые в этой программе функции и класс позаимствованы из модуля, который разработал сам программист

Пишем модуль

Здесь следовало бы показать вам код программы «Простая игра», но этот раздел я посвящу разбору специально написанного модуля, которым программа, кстати говоря, пользуется.

Модуль создается точно так же, как и любая другая программа на Python. Впрочем, при написании модуля желательно собрать вместе несколько взаимосвязанных программных компонентов (функций, классов), чтобы единый файл, в котором они будут храниться, можно было импортировать в новые программы. Я создал базовый модуль под названием `games` с двумя функциями и одним классом, которые могут пригодиться в разработке игры. Код модуля вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке Chapter 9. Файл называется `games.py`.

```
# Игры
# Демонстрирует создание модуля
class Player(object):
    """ Участник игры. """
    def __init__(self, name, score = 0):
        self.name = name
        self.score = score
    def __str__(self):
        rep = self.name + "\t" + str(self.score)
        return rep
def ask_yes_no(question):
    """Задаёт вопрос с ответом 'да' или 'нет'."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
def ask_number(question, low, high):
    """Просит ввести число из заданного диапазона."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response
if __name__ == "__main__":
    print("Вы запустили этот модуль напрямую, а не импортировали его.")
    input("\n\nНажмите Enter, чтобы выйти.")
```

Этот модуль называется `games`, потому что я сохранил файл под названием `games.py`. Названия авторских модулей совпадают с именами файлов без расширения `.py`; под этими же именами их и импортируют.

Основная часть модуля очень проста. Класс `Player` описывает объект с двумя атрибутами: `name` и `score`, которые создает метод-конструктор. В классе, кроме него, задан еще метод `__str__()`, возвращающий строку с теми же двумя свойствами.

Две последующие функции — `ask_yes_no()` и `ask_number()` — вы уже видели в разделе «Вернемся к игре «Крестики-нолики»» главы 6.

Дальнейшая часть программы знакомит вас с новой конструкцией, характерной для модулей. В конструкции `if` условие `__name__ == "__main__"` истинно тогда и только тогда, когда программа запускается напрямую. Если она импортируется в качестве модуля, то условие ложно. Таким образом, при попытке непосредственного запуска файла `games.py` текст на экране уведомит пользователя о том, что этот файл следует импортировать, а не запускать.

Импорт модулей

Теперь, когда вы уже знакомы с модулем `games`, я покажу вам программу «Простая игра». Я буду разбирать код небольшими частями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 9. Файл называется `simple_game.py`.

```
# Простая игра
# Демонстрирует импорт модулей
import games, random
```

Команда `import` загружает модули, созданные самим программистом, точно так же, как и стандартные модули. По сути, можно импортировать несколько встроенных и пользовательских модулей одной строкой кода.

ЛОВУШКА

Если модуль, созданный программистом, находится не в той же самой папке, что и программа, которая его импортирует, то Python не сумеет найти модуль. Этую проблему можно решить. Можно, в частности, «инсталлировать» нестандартный модуль так, что он будет доступен для импорта, как и любой системный модуль, из любого места на компьютере. Но для этого нужно выполнить особую процедуру, описание которой выходит за рамки темы книги. Лучше просто содержите все свои импортируемые модули в той же папке, что и программы, которые ими пользуются.

Применение импортированных функций и классов

В остальной части кода «Простой игры» использованы оба импортированных модуля. Программа приветствует игроков и далее функционирует на основе простого цикла. В начале циклизированного фрагмента компьютер интересуется у пользователя, сколько игроков будет:

```
print("Добро пожаловать в самую-самую простую игру!\n")
again = None
while again != "n":
    players = []
    num = games.ask_number(question = "Сколько игроков участвует? (2 - 5): ", low = 2,
                           high = 5)
```

Чтобы узнать количество игроков, я вызываю функцию `ask_number()` из модуля `games`. Вызвать ее, как и любую другую функцию из импортированного модуля, можно с помощью точечной нотации: имя_модуля.имя_функции.

Затем программа узнает имя каждого участника и генерирует случайное число из диапазона от 1 до 100 (это делает функция `randrange()` из модуля `random`) как игровой результат этого участника. Зная имя и результат, можно создать экземпляр класса `Player`. Поскольку этот класс объявлен в модуле `games`, мне снова пригодится точечная нотация, на этот раз с именем класса после имени модуля. Новый объект-игрок добавляется в конец списка игроков.

```
for i in range(num):
    name = input("Имя игрока: ")
    score = random.randrange(100) + 1
    player = games.Player(name, score)
    players.append(player)
```

Осталось вывести на печать имя и достижение каждого участника:

```
print("\nБот результаты игры:")
for player in players:
    print(player)
```

В заключение с помощью функции `ask_yes_no()` из модуля `games` я интересуюсь, не желает ли пользователь сыграть еще раз:

```
again = games.ask_yes_no("\nХотите сыграть еще раз? (y/n): ")
input("\n\nНажмите Enter, чтобы выйти.")
```

Вернемся к игре «Блек-джек»

Нет оснований сомневаться, что вы научились виртуозно применять на практике классы Python, представляющие игральную карту, колоду и «руку» игрока. Пришло время воспользоваться навыками и основать на этих классах большую программу – настоящую, как в казино, карточную игру (правда, зеленое сукно с ней в комплекте не идет).

Модуль cards

Для создания игры «Блек-джек» я собрал из разных версий программы «Карты» модуль `cards`. В нем классы `Hand` и `Deck` реализованы точно так же, как в «Картах 2.0», а новый класс `Card` совпадает по функциональности с `Positionable_Card` из «Карт 3.0». Код этой программы вы можете найти на сайте-помощнике (courseptr.com/downloads) в папке `Chapter 9`. Файл называется `cards.py`.

```
# Модуль cards
# Набор базовых классов для карточной игры
class Card(object):
    """ Одна игральная карта. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]
    def __init__(self, rank, suit, face_up = True):
        self.rank = rank
        self.suit = suit
        self.is_face_up = face_up
    def __str__(self):
        if self.is_face_up:
            rep = self.rank + self.suit
        else:
            rep = "XX"
        return rep
    def flip(self):
        self.is_face_up = not self.is_face_up
class Hand(object):
    """ 'Рука': набор карт на руках у одного игрока. """
    def __init__(self):
        self.cards = []
```

```

def __str__(self):
    if self.cards:
        rep = ""
        for card in self.cards:
            rep += str(card) + "\t"
    else:
        rep = "<пусто>"
    return rep
def clear(self):
    self.cards = []
def add(self, card):
    self.cards.append(card)
def give(self, card, other_hand):
    self.cards.remove(card)
    other_hand.add(card)
class Deck(Hand):
    """ Колода игральных карт. """
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))
    def shuffle(self):
        import random
        random.shuffle(self.cards)
    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print("Не могу больше сдавать: карты закончились!")
if __name__ == "__main__":
    print("Это модуль, содержащий классы для карточных игр.")
    input("\n\nНажмите Enter, чтобы выйти.")

```

Продумаем систему классов

Прежде чем начать работу над проектом, в котором используется большое количество классов, будет полезно продумать эти классы и выписать на бумагу. Можно, например, составить список с названиями классов и их краткими описаниями. Моя первая попытка в этом направлении применительно к игре «Блек-джек» показана в табл. 9.1.

Таблица 9.1. Классы игры «Блек-джек»

Класс	Базовый класс	Описание
BJ_Card	cards.Card	Карта для игры в «Блек-джек». Дополнительно к атрибутам базового класса объявляет атрибут value — количество очков, соответствующее данной карте

Таблица 9.1 (продолжение)

Класс	Базовый класс	Описание
BJ_Deck	cards.Deck	Колода для игры в «Блек-джек». Представляет собой набор объектов BJ_Card
BJ_Hand	cards.Hand	«Рука» игрока в «Блек-джек». Объявляет атрибуты: name, представляющий имя игрока, и total, равный сумме очков на руках у этого игрока
BJ_Player	BJ_Hand	Игрок в «Блек-джек»
BJ_Dealer	BJ_Hand	Дилер (сдающий при игре в «Блек-джек»)
BJ_Game	object	Игра в «Блек-джек». Объявляет атрибуты: deck со значением — объектом BJ_Deck, dealer со значением — объектом BJ_Dealer и players, ссылающийся на список объектов BJ_Player

Старайтесь перечислить в такой таблице все классы, которые, как вам кажется, могут пригодиться, однако не заботьтесь о том, чтобы сделать их описания исчерпывающими: все равно не получится (у меня вот не получилось). Список поможет вам уяснить, объектами каких типов вам придется оперировать при работе над проектом.

Можно не только описать классы на словах, но и изобразить их «родословное древо», наглядно демонстрирующее все связи между ними. В моем случае такую диаграмму представляет рис. 9.8.

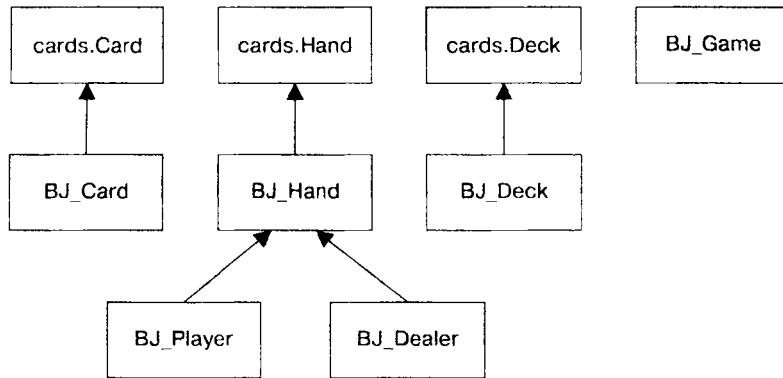


Рис. 9.8. Иерархия наследования классов в игре «Блек-джек»

Схема иерархии классов, наподобие той, что показана на рис. 9.8, даст вам общее понятие о наследовании в вашем проекте.

Напишем псевдокод для основного цикла игры

Далее при планировании игры я написал псевдокод, представляющий розыгрыш одного раунда. Как я полагал, это поможет мне осмыслить взаимодействие объектов. Вот что получилось в итоге:

Раздать каждому игроку и дилеру по 2 карты
Для каждого игрока

Пока игрок просит сдавать ему дополнительные карты и не перебрал количество очков

сдавать игроку по одной карте

Если в игре не осталось игроков

вывести на экран 2 карты дилера

Иначе

Пока дилер должен сдавать себе дополнительные карты и не перебрал количество очков

сдавать дилеру по одной карте

Если дилер перебрал

для каждого игрока, оставшегося в игре

этот игрок побеждает

Иначе

для каждого игрока, оставшегося в игре

Если сумма очков игрока больше суммы очков дилера

этот игрок побеждает

Иначе, если сумма очков игрока меньше суммы очков дилера

этот игрок проигрывает

Иначе

этот игрок сыграл с компьютером вничью

Импорт модулей cards и games

Итак, вы познакомились с планом, а теперь пора перейти к самому коду. В начале программы «Блек-джек» загружается два модуля: cards и games. Я буду разбирать код небольшими частями, однако вы можете ознакомиться и с целой программой на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 9. Файл называется blackjack.py.

```
# Блек-джек
# От 1 до 7 игроков против дилера
import cards, games
```

Модуль games, как вы помните, я реализовал ранее в этой главе, при создании «Простой игры».

Класс BJ_Card

Класс BJ_Card расширяет функциональность своего базового класса cards.Card. В BJ_Card создается новое свойство value, представляющее номинал карты:

```
class BJ_Card(cards.Card):
    """ Кarta для игры в Блек-джек. """
    ACE_VALUE = 1
    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
```

```

if v > 10:
    v = 10
else:
    v = None
return v

```

Метод `value` возвращает число из диапазона от 1 до 10 — количество очков, связанное с данной картой. Вычисления начинаются с доступа к величине `BJ_Card.RANKS[index(self.rank)]`. Эта конструкция берет атрибут `rank` объекта-карты, то есть ее истинный номинал (например, "6"), и находит порядковый номер этого номинала в списке `BJ_Card.RANKS` (для шестерки это 5). Затем к полученному значению прибавляется единица, потому что компьютер начинает считать с 0 (таким образом, от строкового "6" мы перейдем к целочисленному 6). Для валета, дамы и короля вычисленная величина будет превышать 10; на этот случай все значения больше 10 убавляются до этого числа. Если атрибут `is_face_up` объекта имеет значение `False`, то будет без вычислений возвращено значение `None`.

Класс BJ_Deck

Класс `BJ_Deck` предназначен для создания колоды из карт «Блек-джека». Он почти полностью совпадает со своим базовым классом `cards.Deck`. Единственное отличие в том, что я переопределил метод `cards.Deck's populate()`, и теперь новая колода класса `BJ_Deck` наполняется картами класса `BJ_Card`:

```

class BJ_Deck(cards.Deck):
    """ Колода для игры в "Блек-джек". """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))

```

Класс BJ_Hand

Экземпляры класса `BJ_Hand`, основанного на `cards.Hand`, — это «руки», произвольные наборы карт «Блек-джека». Я переопределил конструктор `cards.Hand`, добавив в него атрибут `name`, представляющий имя обладателя данной «руки»:

```

class BJ_Hand(cards.Hand):
    """ 'Рука': набор карт "Блек-джека" у одного игрока. """
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name

```

Затем я переопределил унаследованный метод `__str__()` так, что теперь он отображает сумму очков на руках у игрока:

```

def __str__(self):
    rep = self.name + ": \t" + super(BJ_Hand, self).__str__()
    if self.total:
        rep += "(" + str(self.total) + ")"
    return rep

```

Атрибут `name` объекта я соединяю со строкой, которую возвращает метод `cards.Hand __str__()` по отношению к тому же объекту. Если свойство `total` у объекта не равно `None`, то его строковое представление добавляется сюда же. Полученную таким образом строку метод возвращает.

Вслед за этим я создал свойство `total`, представляющее сумму очков на руках у игрока. Если в составе «руки» есть карта, перевернутая рубашкой вверх, то свойство приобретает значение `None`. В противном случае, чтобы вычислить его значение, суммируются очки, соответствующие всем картам в «руке».

```
@property
def total(self):
    # если у одной из карт value равно None, то и все свойство равно None
    for card in self.cards:
        if not card.value:
            return None
    # суммируем очки, считая каждый туз за 1 очко
    t = 0
    for card in self.cards:
        t += card.value
    # определяем, есть ли туз на руках у игрока
    contains_ace = False
    for card in self.cards:
        if card.value == BJ_Card.ACE_VALUE:
            contains_ace = True
    # если на руках есть туз и сумма очков не превышает 11, будем считать туз за 11
    # очков
    if contains_ace and t <= 11:
        # прибавить нужно лишь 10, потому что единица уже вошла в общую сумму
        t += 10
    return t
```

Первая часть этого метода проверяет, есть ли в составе «руки» хотя бы одна карта со свойством `value`, равным `None` (это означало бы, что карта скрыта). Если да, то метод возвратит `None`. Дальнейший код просто суммирует номиналы всех карт в «руке». Затем надо выяснить, есть ли на руках у участника хотя бы один туз. Если да, то следует решить (что и делает заключительная часть кода метода), каким должен быть эквивалент туза в очках: 1 или 11.

Последний из методов в классе `BJ_Hand` называется `is_busted()`. Он возвращает `True`, когда свойство `total` объекта принимает значение больше 21. В противном случае будет возвращено `False`.

```
def is_busted(self):
    return self.total > 21
```

Заметьте, что в этом методе я возвращаю логическое значение истинности условия `self.total > 21` вместо того, чтобы сохранить это значение в переменной и возвратить ее. Такого рода конструкции с `return` могут содержать в своем составе любое условие (выражение, поддающееся оценке) и зачастую делают методы элегантнее.

Чрезвычайно широко распространены методы, подобные этому, которые возвращают `True` или `False`. Часто их используют для сообщения о том, что объект находится в одном из двух возможных состояний («включен» и «выключен», например). Имена таких методов часто начинаются со слова `is` (в последнем примере я бы назвал метод `is_on()`).

Класс `BJ_Player`

Экземплярами класса `BJ_Player`, производного от `BJ_Hand`, являются игроки в «Блек-джек»:

```
class BJ_Player(BJ_Hand):
    """ Игрок в "Блек-джек". """
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name + ", будете брать еще карты? (Y/N): ")
        return response == "y"
    def bust(self):
        print(self.name, "перебрал.")
        self.lose()
    def lose(self):
        print(self.name, "проиграл.")
    def win(self):
        print(self.name, "выиграл.")
    def push(self):
        print(self.name, "сыграл с компьютером вничью.")
```

Первый метод — `is_hitting()` — возвращает `True` в том случае, если игроку угодно получить еще карту, и `False` — если нет. Метод `bust()` объявляет, что участник перебрал, и вызывает метод `lose()` данного объекта. В свою очередь, метод `lose()` объявляет, что участник проиграл, а метод `win()` — что участник выиграл. Наконец, метод `push()` заявляет о ничьей. Методы `bust()`, `lose()`, `win()` и `push()` так просто реализованы, что вы могли бы усомниться в их целесообразности. Я поместил их в класс потому, что они создают удобный «каркас» для обработки более сложных форм игры, например такой, в которой игрокам разрешено делать ставки (система ставок появится в игре уже совсем скоро — после того, как вы решите одну из задач в конце этой главы).

Класс `BJ_Dealer`

Экземпляром класса `BJ_Dealer`, производного от `BJ_Hand`, является дилер «Блек-джека»:

```
class BJ_Dealer(BJ_Hand):
    """ Дилер в игре "Блек-джек". """
    def is_hitting(self):
        return self.total < 17
    def bust(self):
        print(self.name, "перебрал.")
```

```
def flip_first_card(self):
    first_card = self.cards[0]
    first_card.flip()
```

Первый метод — `is_hitting()` — определяет, будет ли дилер брать дополнительные карты. Поскольку дилер, имея на руках не более 17 очков, обязан тянуть очередную карту, метод возвращает `True`, если свойство `total` объекта не превышает 17, а в противном случае возвращает `False`. Метод `bust()` объявляет, что дилер перебрал. Наконец, метод `flip_first_card()` переворачивает первую карту дилера лицевой стороной вниз.

Класс BJ_Game

Класс `BJ_Game` используется для создания объектов, которые будут представлять отдельные игры в «Блек-джек». В методе `play()` этого класса содержится код основного цикла игры. Впрочем, игровой процесс довольно сложен, так что мне пришлось создать вне этого метода несколько вспомогательных элементов, в том числе метод `__additional_cards()`, который позаботится о сдаче дополнительных карт на руки игроку, и свойство `still_playing`, которое будет возвращать список всех игроков, еще не перебравших очки в текущем раунде.

Метод `__init__()`

Конструктор принимает список имен и создает на каждое имя по игроку. Кроме того, будут созданы дилер и колода.

```
class BJ_Game(object):
    """ Игра в Блек-джек. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)
        self.dealer = BJ_Dealer("Dealer")
        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()
```

Свойство `still_playing`

Свойство `still_playing` возвращает список всех игроков, которые еще остались в игре (не перебрали количество очков в течение данного раунда):

```
@property
def still_playing(self):
    sp = []
    for player in self.players:
        if not player.is_busted():
            sp.append(player)
    return sp
```

Метод __additional_cards()

Метод `__additional_cards()` сдает игроку или дилеру дополнительные карты. Этот метод принимает в свой параметр `player` объект класса `BJ_Player` или `BJ_Dealer`. До тех пор пока метод `is_busted()` данного объекта возвращает `False`, а метод `is_hitting()` возвращает `True`, программа будет сдавать карты. Если метод `is_busted()` возвратит `True`, то по отношению к объекту будет вызван метод `bust()`.

```
def __additional_cards(self, player):
    while not player.is_busted() and player.is_hitting():
        self.deck.deal([player])
        print(player)
        if player.is_busted():
            player.bust()
```

Здесь можно в двух местах увидеть в деле полиморфизм. Во-первых, метод `player.is_hitting()` сработает вне зависимости от того, к какому классу относится объект `player`: `BJ_Player` или `BJ_Dealer`. Методу `__additional_cards()` незачем знать, с каким из двух типов объектов он имеет дело. Во-вторых, в классах `BJ_Player` и `BJ_Dealer` реализован метод `bust()`, так что команда `player.bust()` в обоих случаях выводит нужный нам результат.

Метод play()

Метод `play()` содержит основной цикл игры и отмечен поразительным сходством с псевдокодом:

```
def play(self):
    # сдача всем по 2 карты
    self.deck.deal(self.players + [self.dealer], per_hand = 2)
    self.dealer.flip_first_card() # первая из карт, сданных дилеру, переворачивается рубашкой вверх
    for player in self.players:
        print(player)
    print(self.dealer)
    # сдача дополнительных карт игрокам
    for player in self.players:
        self.__additional_cards(player)
    self.dealer.flip_first_card() # первая карта дилера раскрывается
    if not self.still_playing:
        # все игроки перебрали, покажем только "руку" дилера
        print(self.dealer)
    else:
        # сдача дополнительных карт дилеру
        print(self.dealer)
        self.__additional_cards(self.dealer)
        if self.dealer.is_busted():
            # выигрывают все, кто еще остался в игре
            for player in self.still_playing:
                player.win()
        else:
```

```

# сравниваем суммы очков у дилера и у игроков, оставшихся в игре
for player in self.players:
    if player.total > self.dealer.total:
        player.win()
    elif player.total < self.dealer.total:
        player.lose()
    else:
        player.push()

# удаление всех карт
for player in self.players:
    player.clear()
self.dealer.clear()

```

Каждый игрок и дилер получают по две начальные карты. Первая из карт, сданных дилеру, переворачивается рубашкой вверх, чтобы можно было скрыть ее значение. Затем на экран выводятся «руки» всех участников игры. После этого дилер будет сдавать каждому игроку карты до тех пор, пока тот просит это делать и не перебрал сумму очков. Если все игроки перебрали, то первая карта дилера переворачивается и на экран выводится его «рука». В противном случае игра продолжается. Дилер сдает себе карты до тех пор, пока сумма очков у него на руках не превышает 17. Если дилер перебрал, то все оставшиеся в игре участники выигрывают. В противном случае сравниваются суммы очков на руках у дилера и у каждого оставшегося игрока. Если игрок набрал больше очков, чем дилер, он побеждает, если меньше — проигрывает, если столько же — объявляется ничья.

Функция main()

Функция `main()` принимает имена всех игроков, организует их в виде списка и создает объект `BJ_Game`, которому этот список передается как аргумент. Затем функция вызывает метод `play()` данного объекта и будет делать это снова и снова до тех пор, пока игроки не изъявили желание прекратить игру.

```

def main():
    print("\t\tДобро пожаловать за игровой стол Блек-джека!\n")
    names = []
    number = games.ask_number("Сколько всего игроков? (1 - 7): ", low = 1, high
= 8)
    for i in range(number):
        name = input("Введите имя игрока: ")
        names.append(name)
        print()
    game = BJ_Game(names)
    again = None
    while again != "n":
        game.play()
        again = games.ask_yes_no("\nХотите сыграть еще раз? ")
    main()
    input("\n\nНажмите Enter, чтобы выйти.")

```

Резюме

Эта глава продолжила ваше знакомство с миром ООП. Вы увидели, как отправлять сообщение от одного объекта к другому. Вы научились создавать из более простых объектов более сложные. Вы познакомились с наследованием — созданием новых классов на основе существующих. Вы увидели, как расширить производный класс путем добавления в него новых методов и переопределения унаследованных методов. Вы освоили технику написания и импортирования собственных модулей. В этой главе был продемонстрирован набросок системы классов, с которого может начинаться работа над проектом. Применение всех этих свежих идей на практике вы увидели при разработке настоящей карточной игры для нескольких игроков, как в казино.

ЗАДАЧИ

- Добавьте в игру «Блек-джек» проверку на ошибки. Перед началом очередного раунда надо проверять, достаточно ли карт в колоде. Если нет, колоду следует вновь наполнить и перемешать. Найдите в программе и другие уязвимые места, которым не помешает проверка на ошибки или перехват исключений.
- Напишите однокарточную версию игры «Война», структура раунда в которой такова: все игроки тянут по одной карте, а выигрывает тот, у кого номинал карты оказывается наибольшим.
- Доработайте проект «Блек-джек» так, чтобы игроки могли делать ставки. Программа должна следить за капиталом каждого игрока и выводить из-за стола тех, у кого закончатся деньги.
- Создайте несложную объектно-ориентированную приключенческую игру, в которой игрок сможет менять свое местонахождение, перемещаясь каждый раз в одно из мест, ближайших к данному.

10

Разработка графических интерфейсов. Программа **«Сумасшедший сказочник»**

Все программы, код которых я вам показывал до сих пор, взаимодействовали с пользователем через обычный текст. Но есть и более хитроумные способы представления данных и обмена ими. Графический пользовательский интерфейс (GUI) — средство визуального взаимодействия между пользователем и компьютером. GUI применяется во всех популярных операционных системах на домашних компьютерах; он упрощает действия пользователя и делает их единообразными. В этой главе вы научитесь создавать GUI. Если говорить подробнее, то вы научитесь делать следующее:

- пользоваться набором инструментов для создания GUI;
- создавать и заполнять рамки;
- создавать и применять кнопки;
- создавать и применять текстовые поля и области;
- создавать и применять флажки;
- создавать и применять переключатели.

Знакомство с программой **«Сумасшедший сказочник»**

Программа «Сумасшедший сказочник», разработкой которой мы займемся в этой главе, просит пользователя помочь в составлении назидательного рассказа. Пользователь должен ввести имя человека, существительное во множественном числе и инфинитив глагола. Предоставлена также возможность указать одно или несколько прилагательных и выбрать одну часть тела. Получив все эти данные, программа создает рассказ. Ее работа показана на рис. 10.1–10.3. Как видите, «Сумасшедший сказочник» взаимодействует с пользователем через GUI.

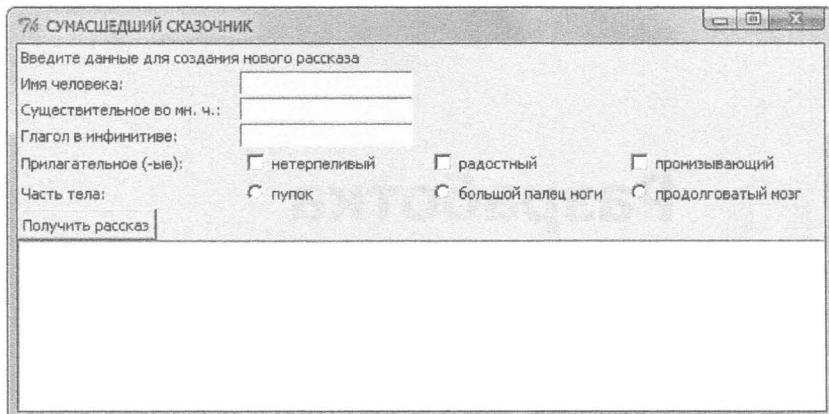


Рис. 10.1. Симпатичный GUI ожидает творческих идей пользователя

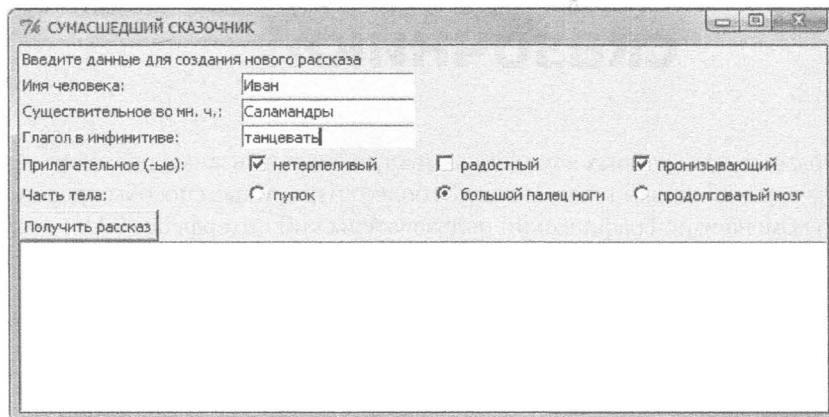


Рис. 10.2. Пользователь ввел все необходимые данные

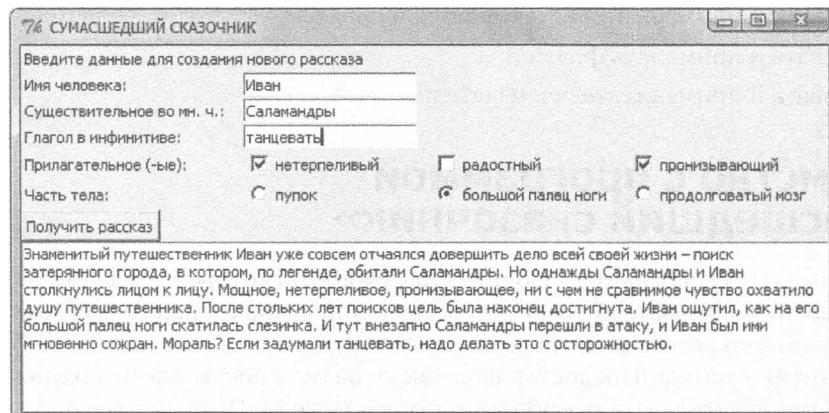


Рис. 10.3. После нажатия кнопки Получить рассказ в текстовой области появится автоматически созданный шедевр

GUI в подробностях

Прежде чем вы узнаете, как программировать GUI, я познакомлю вас со всеми элементами управления, которые вы встретите в графических интерфейсах в этой главе. На рис. 10.4 показано окно «Сумасшедшего сказочника» с подписанными элементами.

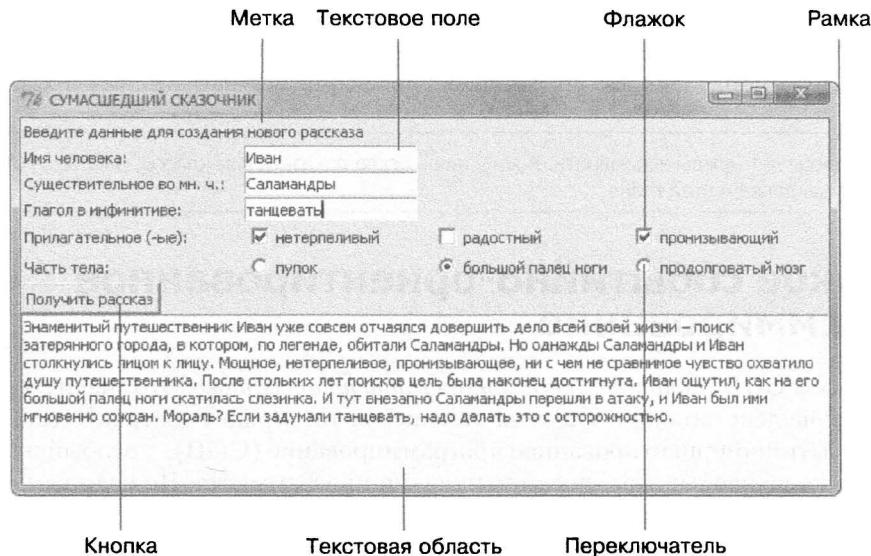


Рис. 10.4. Вы научитесь создавать все эти графические элементы управления

Чтобы создать GUI с помощью Python, нужен набор инструментов для разработки GUI. Таковых есть несколько на выбор; в этой главе я буду пользоваться популярным кроссплатформенным пакетом Tkinter.

ПОДСКАЗКА

Если на вашем компьютере установлена не операционная система Windows, а какая-либо другая, то для применения пакета Tkinter вам, возможно, понадобится скачать и установить дополнительное ПО. Подробнее об этом читайте на странице Tkinter на сайте Python: <http://www.python.org/topics/tkinter>.

Элементы GUI – это экземпляры классов модуля `tkinter`, который входит в состав пакета Tkinter. В табл. 10.1 описаны все элементы GUI, перечисленные на рис. 10.4, и названы соответствующие им классы `tkinter`.

Таблица 10.1. Избранные элементы GUI

Элемент	Класс <code>tkinter</code>	Описание класса
Рамка	Frame	Служит базой для других элементов GUI
Метка	Label	Отображает значок или текст, который нельзя прописать

Продолжение ↗

Таблица 10.1 (продолжение)

Элемент	Класс tkinter	Описание класса
Кнопка	Button	После пользовательского нажатия выполняет какое-либо действие
Текстовое поле	Entry	Принимает и отображает одну строку текста
Текстовая область	Text	Принимает и отображает несколько строк текста
Флажок	Checkbutton	Позволяет пользователю включить или выключить какой-либо параметр
Переключатель	Radiobutton	Позволяет пользователю выбрать один параметр из нескольких сгруппированных

ПОДСКАЗКА

Все эти классы нетрудно запомнить. Я хочу лишь бегло показать вам классы, работе с которыми вы обучитесь далее в этой главе.

Что такое событийно-ориентированное программирование

Программы GUI, как правило, *событийно-ориентированные*. Это значит, что они отвечают на действия пользователя независимо от порядка совершаемых операций. Событийно-ориентированное программирование (СОП) — особый путь написания кода, необычный способ мышления программиста. Но если вам когда-нибудь приходилось пользоваться программами с GUI (например, браузером), то работать с событийно-ориентированными системами вы уже умеете.

Чтобы прояснить суть СОП, рассмотрим с внешней стороны программу «Сумасшедший сказочник». Если бы вы решили создать нечто подобное, пользуясь только нынешними навыками программирования на Python, то ваша программа, вероятно, задавала бы пользователю серию вопросов, ответы на которые получала бы функцией `input()`. Компьютер просил бы ввести сначала имя человека, потом существительное во множественном числе, потом глагол и т. д. Все эти сведения пользователь должен был вводить последовательно, по порядку. Напротив, событийно-ориентированная программа, например программа с GUI, позволит вводить информацию в любом порядке. Выбрать момент вывода конечного текста рассказа на экран тоже сможет пользователь.

В СОП *события* (то, что может произойти с объектами программы) связываются с *обработчиками* (кодом, запускаемым при соответствующих событиях). Вот конкретный пример. Когда в программе «Сумасшедший сказочник» пользователь нажимает кнопку **Получить рассказ** (это событие), программа вызывает метод, который выводит рассказ на экран (это обработчик события). Чтобы все названное произошло, надо связать нажатие кнопки с методом, выводящим текст.

Задавая объекты, события и обработчики событий, вы задаете порядок работы программы. Потом программу надо запустить, создав *событийный цикл*, внутри которого она будет ожидать описанных вами событий. Когда любое из этих событий происходит, программа обрабатывает его предписанным вами образом.

Не беспокойтесь, если пока не до конца понимаете этот новый для вас взгляд на программирование. Усвоив логику работы нескольких образцов, вы поймете, как создавать собственные событийно-ориентированные приложения.

Базовое окно

Отправная точка любой программы с графическим интерфейсом — *базовое*, или *корневое*, окно, поверх которого размещаются все остальные элементы GUI. Если представить себе GUI в виде дерева, то это будет корень. Ваше дерево может «ветвиться» в разных направлениях, но каждая его часть прямо или косвенно будет быть возведена к корню.

Знакомство с программой «Простейший GUI»

В программе «Простейший GUI» создается, пожалуй, самый бесхитростный графический интерфейс: только окно. Результат работы программы отражен на рис. 10.5.

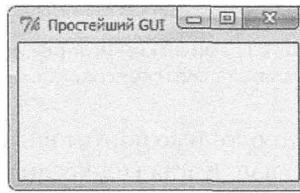


Рис. 10.5. Эта программа создает на экране одно-единственное пустое окно.
Но ведь с чего-то надо начинать, правда?

ЛОВУШКА

Если запустить программу на основе Tkinter прямо из IDLE, то либо программа, либо IDLE зависнет. Проще всего решить эту проблему, непосредственно запуская программу; в Windows для этого достаточно дважды щелкнуть на ее значке.

ХИТРОСТЬ

Хотя после двойного щелчка кнопкой мыши оконное приложение и начнет работать, возникнет другая сложность: если код программы содержит ошибку, то консольное окно закроется прежде, чем вы успеете прочесть описание этой ошибки. При работе в Windows оптимально создать пакетный файл, который будет запускать программу и дожидаться окончания ее работы, так что окно консоли останется открытым и сообщения об ошибках будут видны. Если, например, ваша программа называется `simple_gui.py`, то запускающий ее пакетный файл должен состоять всего из двух строк:

```
simple_gui.py  
pause
```

Этот файл надо вызвать на исполнение, дважды щелкнув на его значке.

Чтобы создать пакетный файл, откройте простой текстовый редактор, такой как Блокнот (но не Word или WordPad). Затем введите код. Сохраните файл с расширением `.bat` (например, `simple_gui.bat`). Убедитесь, что за `.bat` не следует расширение `.txt`.

Я создал запускающие пакетные файлы ко всем программам этой главы. Вместе с кодом программ вы можете найти их на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10.

Кроме того окна, что изображено на рис. 10.5, программа «Простейший GUI» может (не во всех операционных системах) вызвать к жизни еще одно окно — хорошо знакомую нам консоль, текущий вид которой можно лицезреть на рис. 10.6.

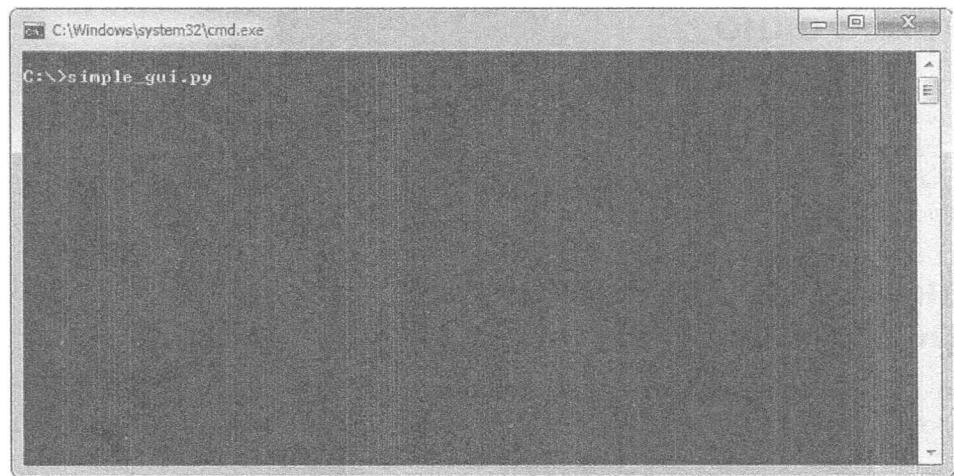


Рис. 10.6. Программа с графическим интерфейсом может создать на экране также и консольное окно

Вы можете счесть, что консоль только портит вид вашего замечательного GUI, но пренебрегать ею все же не надо. Когда (или если) в вашей программе с интерфейсом Tkinter произойдет ошибка, в окне консоли появятся подробные сведения о ней. Консоль не следует закрывать еще и потому, что в этом случае GUI тоже прекратит работу.

ХИТРОСТЬ

Добавившись безошибочной работы своего GUI-приложения, вы, возможно, пожелаете убрать второе окно. Чтобы сделать это на компьютере с операционной системой Windows, проще всего изменить расширение файла с .py на .pwy.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется simple_gui.py.

Импорт модуля tkinter

Вот и настало время погрузиться в код! Первым делом в программе «Простейший GUI» я загрузил модуль tkinter:

```
# Простейший GUI
# Демонстрирует создание окна
from tkinter import *
```

Эта строка непосредственно переносит все содержимое tkinter в глобальную область видимости программы. Обычно этого принято избегать, но некоторые мо-

дули, в частности `tkinter`, импортируются именно таким образом. Из дальнейшего кода вы поймете, чем это удобно.

Создание базового окна

Чтобы создать базовое окно, я инстанцировал класс `Tk` из модуля `tkinter`:

```
# создание базового окна
root = Tk()
```

Обратите внимание на то, что префикс `tkinter` к названию класса `Tk` не добавляется. В сущности, любая часть модуля `tkinter` доступна без префикса. Поскольку обычно в программах на основе этого пакета содержится много ссылок на классы и константы внутри модуля, программист таким образом освобождается от массы ненужного труда, а код становится легче читать.

ЛОВУШКА

В программах с интерфейсом Tkinter может быть только одно базовое окно. Если создать их два экземпляра, то программа перестанет отвечать, так как окна будут спорить за приоритет.

Изменение вида базового окна

Теперь я воспользуюсь двумя методами базового окна, чтобы изменить его внешний вид:

```
# изменение окна
root.title("Простейший GUI")
root.geometry("200x100")
```

Метод `title()` назначает заголовок окна. Достаточно передать ему строку, которую вы хотите отобразить в заголовке. В моем примере самая верхняя строка окна получает текст "Простейший GUI".

Метод `geometry()` устанавливает размеры базового окна в пикселях. Этот метод принимает строковый (а не целочисленный) аргумент, в котором ширина и высота окна должны быть разделены символом `x`. Я назначил ширину окна равной 200 пикселям, а высоту — 100 пикселям.

Запуск событийного цикла базового окна

Теперь наконец можно вызвать метод `mainloop()`, чтобы начать событийный цикл базового окна:

```
# старт событийного цикла
root.mainloop()
```

Как результат, окно остается открытым и ожидает событий для дальнейшей обработки. Поскольку я не описал ни одного события, функциональность полученного окна невелика. Однако это уже абсолютно полноценное окно, которое можно изменить в размерах, свернуть или закрыть. Не бойтесь поэкспериментировать с ним; для этого щелкните дважды на значке пакетного файла `simple_gui.bat`.

Применение меток

В состав GUI входят различные элементы управления. Пожалуй, нет другого столь же простого элемента управления, как *метка* — текст и/или изображение, которые нельзя редактировать. В соответствии со своим названием метка «помечает» какую-либо часть интерфейса, например другой элемент. В отличие от большинства элементов управления, метки не интерактивны: если пользователь щелкнет на метке, то система никак не отреагирует. Впрочем, это не мешает меткам быть по-своему полезными. Думаю, при создании любого графического интерфейса вы по крайней мере однажды воспользуетесь меткой.

Знакомство с программой «Это я, метка»

В программе внутри базового окна создается элемент `Label`, который попросту объявляет о своем существовании. Внешний вид программы показан на рис. 10.7.

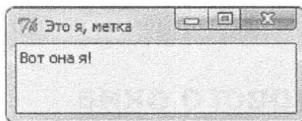


Рис. 10.7. В тексте метки может присутствовать информация о GUI

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `labeler.py`.

Настройка программы

Сначала я выполнил настройку программы: импортировал модуль `tkinter` и создал базовое окно.

```
# Это я, метка
# Демонстрирует применение меток
from tkinter import *
# создание базового окна
root = Tk()
root.title("Это я, метка")
root.geometry("200x50")
```

Создание рамки

Рамка — это такой элемент управления, внутри которого могут содержаться другие элементы, например метки. Функционально рамка схожа с пробковым покрытием офисной доски, на которую булавками крепятся разные мелочи.

Итак, я создаю новую рамку:

```
# внутри окна создается рамка для размещения других элементов
app = Frame(root)
```

Каждый раз при создании очередного элемента управления вы должны передавать конструктору нового объекта его *родительский элемент*, то есть тот элемент, внутри которого он находится. В данном случае методу-конструктору Frame было передано root. Как следствие, внутри базового окна появляется новая рамка.

Теперь я вызову метод grid() нового объекта:

```
app.grid()
```

Метод grid() есть у всех элементов управления. Он связан с *менеджером размещения*, с помощью которого можно управлять расположением элементов в окне. Чтобы не усложнять все преждевременно, мы отложим беседу о менеджерах размещения.

Создание метки

Для создания метки я инстанцировал класс Label:

```
# создание метки внутри рамки  
lbl = Label(app, text = "Вот она я!")
```

Конструктору объекта Label я передал аргумент app. Тем самым я велел интерпретатору считать, что рамка app — родительский элемент относительно метки. Вот почему метка будет помещена внутрь рамки.

У элементов управления есть параметры, значения которых может выбирать программист. Многие из этих параметров определяют внешний вид элемента. Так, например, я передал параметру text текст "Вот она я!", и при отображении метки на экране появятся слова Вот она я!.

Теперь я вызову метод grid() объекта-метки:

```
lbl.grid()
```

Благодаря этому метка наверняка будет отображена в составе GUI.

Запуск событийного цикла базового окна

И вот наконец начинает работу событийный цикл базового окна, а вместе с ним и весь GUI:

```
# старт событийного цикла  
root.mainloop()
```

Применение кнопок

Кнопку, элемент управления класса Button, пользователь может зачем-либо активизировать — нажать. Поскольку вам уже известно, как создавать метки, процедура создания кнопок не должна вызвать у вас трудностей.

Знакомство с программой «Бесполезные кнопки»

В программе «Бесполезные кнопки» создается несколько кнопок, которые, если их нажимать, никак не отвечают. Поместить на графический интерфейс такие кнопки — все равно что прикрутить к потолку люстру без лампочек. Она уже на своем месте, но еще не функциональна. Вид окна программы показан на рис. 10.8.

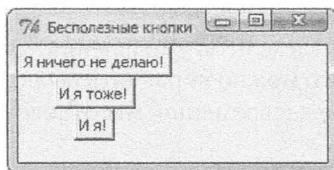


Рис. 10.8. Можете сколько угодно нажимать эти бесполезные кнопки: они не отзовутся

Код программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `lazy_buttons.py`.

Настройка программы

Сначала, как обычно, я импортирую `tkinter`, создам базовое окно и рамку в нем:

```
# Бесполезные кнопки
# Демонстрирует создание кнопок
from tkinter import *
# создание базового окна
root = Tk()
root.title("Бесполезные кнопки")
root.geometry("200x85")
# внутри окна создается рамка для размещения других элементов
app = Frame(root)
app.grid()
```

Создание кнопок

Чтобы создать кнопку, я инстанцировал класс `Button`. Это и выполняют следующие строки кода:

```
# создание кнопки внутри рамки
bttn1 = Button(app, text = "Я ничего не делаю!")
bttn1.grid()
```

Здесь создается новая кнопка с надписью `Я ничего не делаю!`. Ее родительский элемент — ранее созданная рамка; таким образом, кнопка будет размещена внутри рамки.

Когда дело доходит до создания, описания и изменения внешнего вида элементов управления, модуль `tkinter` проявляет немало гибкости. Так, можно создать элемент

и определить все его параметры одной строкой кода, как это сделано у меня. Можно, напротив, сначала создать элемент, а потом задать или изменить его вид. На примере следующей кнопки вы поймете, что я имею в виду.

Сначала создается новая кнопка:

```
# создание второй кнопки внутри рамки  
bttn2 = Button(app)  
bttn2.grid()
```

Заметьте, что конструктору объекта я передаю только значение app — имя родительского элемента кнопки. Итак, пока я только добавил внутрь рамки пустую кнопку. Но это легко исправить. Вид кнопки можно изменять после создания, для чего существует метод объекта `configure()`:

```
bttn2.configure(text = "И я тоже!")
```

Эта строка присваивает параметру `text` данной кнопки значение "И я тоже!", так что соответствующий текст появляется на кнопке. Метод `configure()` позволяет сконфигурировать любой параметр у элемента какого угодно типа. С его помощью можно даже назначить новое значение уже установленному параметру.

Теперь я создам еще одну кнопку:

```
# создание третьей кнопки внутри рамки  
bttn3 = Button(app)  
bttn3.grid()
```

В этом случае значение параметра `text` будет выбрано иным способом:

```
bttn3["text"] = "И я!"
```

Как видите, я осуществил доступ к параметру `text` через интерфейс, подобный словарю. Установлено значение "И я!", так что на кнопке будет отображаться текст И я!. Чтобы присваивать значения в таком стиле («словарном»), надо указывать в качестве ключей имена параметров, взятые в кавычки.

Запуск событийного цикла базового окна

Как всегда, для запуска GUI нужно начать работу событийного цикла базового окна:

```
# старт событийного цикла  
root.mainloop()
```

Создание GUI с помощью класса

Как вы убедились в предшествующих главах, представление кода в виде классов намного облегчает жизнь программисту. При разработке больших приложений с графическим интерфейсом классы используются часто. Итак, посмотрим, как написать GUI-программу, организовав ее код в класс.

Знакомство с программой «Бесполезные кнопки — 2»

Программа «Бесполезные кнопки — 2» — это «Бесполезные кнопки», переписанные в объектном стиле. С точки зрения пользователя, все осталось прежним, но внутренняя структура программы все же претерпела некоторые изменения.

Уже знакомое вам окно показано на рис. 10.9.

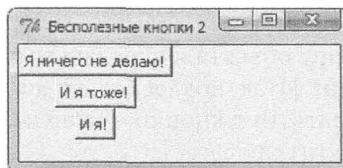


Рис. 10.9. Все по-прежнему. Окно программы выглядит и работает точно так же, как и раньше, хотя код стал существенно иным

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `lazy_buttons2.py`.

Импорт модуля `tkinter`

Несмотря на значительные нововведения в структуре программы, модуль графического интерфейса по-прежнему загружается в самом начале:

```
# Бесполезные кнопки – 2
# Демонстрирует создание класса в оконном приложении на основе tkinter
from tkinter import *
```

Объявление класса `Application`

Теперь я создам новый класс `Application`, производный от `Frame`:

```
class Application(Frame):
    """ GUI-приложение с тремя кнопками. """
```

Вместо того чтобы создавать экземпляр класса `Frame`, я буду инстанцировать класс `Application`, объект которого уже будет содержать все кнопки. Такой подход удобен потому, что объект `Application` — это всего лишь особого рода рамка.

Объявление метода-конструктора

В классе `Application` я объявлю конструктор:

```
def __init__(self, master):
    """ Инициализирует рамку. """
    super(Application, self).__init__(master)
    self.grid()
    self.create_widgets()
```

В этом коде я первым делом вызываю конструктор надкласса. Объекту `Application` передается имя родительского элемента (окна); в этих условиях объект ведет

себя точно так же, как обычная рамка. Вслед за тем я вызываю метод объекта `create_widgets()`, объявленный далее.

Объявление метода, создающего элементы управления

Вот метод `create_widgets()`, который создает все три кнопки на рамке:

```
def create_widgets(self):
    """ Создает три бесполезные кнопки. """
    # первая кнопка
    self.bttn1 = Button(self, text = "Я ничего не делаю!")
    self.bttn1.grid()
    # вторая кнопка
    self.bttn2 = Button(self)
    self.bttn2.grid()
    self.bttn2.configure(text = "И я тоже!")
    # третья кнопка
    self.bttn3 = Button(self)
    self.bttn3.grid()
    self.bttn3["text"] = "И я!"
```

Этот код очень похож на те инструкции, с помощью которых мы создавали кнопки в первой версии программы «Бесполезные кнопки». Важное отличие состоит в том, что `bttn1`, `bttn2` и `bttn3` – атрибуты объекта `Application`. Еще одно отличие таково: в качестве родительского элемента для кнопок я назначил `self`, следовательно, они будут «прикреплены» к объекту `Application`.

Создание объекта класса Application

В основной части программы я создаю базовое окно, присваиваю ему заголовок и назначаю подходящие размеры:

```
# основная часть
root = Tk()
root.title("Бесполезные кнопки - 2")
root.geometry("200x85")
```

Вслед за этим я инстанцирую класс `Application` с родительским элементом `root`:

```
app = Application(root)
```

Данный код создает объект `Application` в окне `root`. Метод-конструктор этого объекта вызывает метод `create_widgets()`, который, в свою очередь, создает внутри объекта `Application` три кнопки.

В конце, как обычно, запускается событийный цикл, начинающий работу GUI:

```
root.mainloop()
```

Связывание элементов управления с обработчиками событий

GUI-приложения, рассмотренные ранее в этой главе, практически бесполезны. Это потому, что в них на активизацию пользователем элементов управления не реагирует никакой код. Итак, если раньше наши элементы были похожи на обесточенные люстры без лампочек, то теперь «подключим» их и заставим эффективно работать. В терминах GUI-программирования мы сейчас займемся тем, что будем писать обработчики событий и связывать их с событиями.

Знакомство с программой «Счетчик щелчков»

В программе «Счетчик щелчков» есть работающая кнопка: на ней отображается, сколько раз пользователь ее нажал. Говоря технически, связанный с этой кнопкой обработчик каждый раз при событии (щелчке) увеличивает количество нажатий на единицу и меняет текст на кнопке. Работу программы иллюстрирует рис. 10.10.

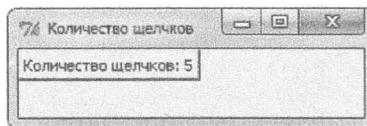


Рис. 10.10. Обработчик события, связанный с кнопкой, увеличивает счет нажатий на единицу каждый раз, когда пользователь нажимает кнопку

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `click_counter.py`.

Настройка программы

Традиционный первый шаг — загрузить модуль графического интерфейса:

```
# Счетчик щелчков
# Демонстрирует связывание событий с обработчиками
from tkinter import *
```

Вслед за тем объявляется Application:

```
class Application(Frame):
    """ GUI-приложение, которое подсчитывает количество нажатий кнопки. """
    def __init__(self, master):
        """ Инициализирует рамку. """
        super(Application, self).__init__(master)
        self.grid()
        self.bttm_clicks = 0 # количество нажатий
        self.create_widget()
```

По большей части этот код вам уже знаком. Из нового в нем только одна строка, `self.bttm_clicks = 0`, которая создает у объекта атрибут, следящий за количеством нажатий кнопки.

Связывание обработчика с событием

В методе `create_widget()` создам единственную кнопку:

```
def create_widget(self):
    """ Создает кнопку, на которой отображается количество совершенных нажатий. """
    self.btn = Button(self)
    self.btn["text"] = "Количество щелчков: 0"
    self.btn["command"] = self.update_count
    self.btn.grid()
```

Параметр `command` элемента ссылается на метод `update_count()`. Поэтому, когда пользователь нажимает кнопку, вызывается данный метод. Как говорят программисты, я «связал» событие (нажатие кнопки) с обработчиком события (методом `update_count()`). Параметр `command` любого элемента управления вообще предназначен для того, чтобы при его активизации вызывать метод-обработчик.

Создание обработчика события

Теперь я напишу метод `update_count()`, который будет обрабатывать нажатие кнопки:

```
def update_count(self):
    """ Увеличивает количество нажатий на единицу и отображает его. """
    self.btn_clicks += 1
    self.btn["text"] = "Количество щелчков: " + str(self.btn_clicks)
```

Этот метод увеличивает общее количество совершенных пользователем нажатий кнопки и затем изменяет текст на кнопке так, чтобы отображалось это новое число. Простая механика, но благодаря ей кнопка стала хоть чуть-чуть полезной.

Обертка программы

Основная часть кода вам уже известна:

```
# основная часть
root = Tk()
root.title("Количество щелчков")
root.geometry("200x50")
app = Application(root)
root.mainloop()
```

Я создал базовое окно `root`, назначил его заголовок и установил размеры. Затем я создал экземпляр класса `Application` внутри родительского элемента `root`. Наконец, запуск событийного цикла базового окна выводит GUI на экран.

Текстовые поля и области. Менеджер размещения Grid

В GUI-программировании нередко возникает потребность сделать так, чтобы пользователь мог ввести или прочесть какой-либо текст. Для обоих случаев предназначены

текстовые элементы управления. Я познакомлю вас с двумя из них: *поле* удобно для одной строки текста, а *область* может содержать блок текста, состоящий из нескольких строк. Программа способна «читать» содержимое элементов обоих типов, чтобы таким образом получать пользовательский ввод. Кроме того, как средство обратной связи с пользователем эти элементы управления можно вставлять текст.

Поскольку мы научились помещать на рамку столько разных элементов, надо научиться грамотно располагать их. В предшествующих программах я применял менеджер размещения Grid, но лишь самым обычным образом. Между тем Grid предоставляет широкие возможности управления внешним видом GUI. Менеджер покрывает рамку невидимой координатной сеткой, на которой можно позиционировать элементы удобным для вас образом.

Знакомство с программой «Долгожитель»

Программа «Долгожитель» по секрету сообщит, как дожить до 100 лет, но только если ввести чрезвычайно надежный и стойкий к взлому пароль secret. Пользователь должен вписать пароль в текстовое поле и затем нажать кнопку Узнать секрет. Если пароль верен, то в текстовой области, расположенной ниже, программа поделится своей тайной. Работу программы иллюстрируют рис. 10.11 и 10.12.

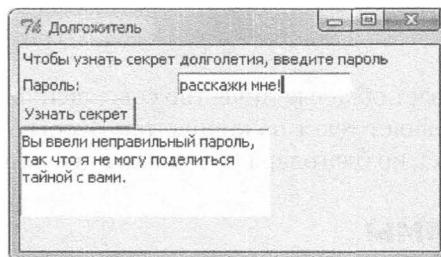


Рис. 10.11. Если пользователь не сумеет ввести правильный пароль, программа вежливо откажется разглашать секрет

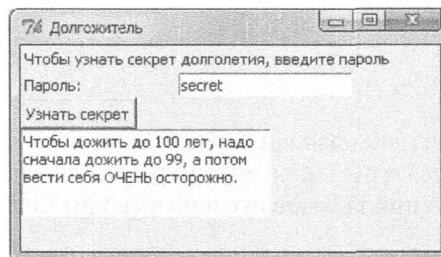


Рис. 10.12. Если пароль окажется верным, то бесценное знание, ключ к долгим годам жизни, перекочует от компьютера к пользователю

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `longevity.py`.

Настройка программы

В начале программы я проделал все то же, что и обычно:

```
# Долгожитель
# Демонстрирует текстовое поле, текстовую область и менеджер размещения Grid
from tkinter import *
class Application(Frame):
    """ GUI-приложение, владеющее секретом долголетия. """
    def __init__(self, master):
        """ Инициализирует рамку. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

После загрузки модуля `tkinter` происходит объявление класса `Application`. Его метод-конструктор инициализирует новый объект `Application`, удостоверяет, что объект будет видимым на экране, и вызывает метод `create_widgets()`.

Размещение элементов управления с помощью менеджера Grid

Теперь я объявлю метод `create_widgets()` и в нем первым делом создам инструктирующую пользователя текстовую метку:

```
def create_widgets(self):
    """ Создает кнопку, текстовое поле и текстовую область. """
    # метка-инструкция
    self.inst_lbl = Label(self, text = "Чтобы узнать секрет долголетия, введите пароль")
```

Пока ничего нового, правда? Однако уже в следующей строке я даю менеджеру размещения `Grid` явные инструкции на предмет того, как расположить метку:

```
self.inst_lbl.grid(row = 0, column = 0, columnspan = 2, sticky = W)
```

Метод `grid()` объекта может принимать значения для разных параметров. Я пользуюсь только четырьмя из них: `row`, `column`, `columnspan` и `sticky`.

Параметры `row` и `column` принимают целочисленные значения и определяют положение объекта относительно родительского элемента управления. Можно представить себе, что наша рамка в базовом окне представляет собой сетку или таблицу со строками и столбцами. Каждое пересечение строки и столбца образует «ячейку», в которую можно поместить элемент. На рис. 10.13 показано размещение девяти кнопок в девяти ячейках, которые однозначно описываются номерами строк и столбцов.

В случае с моей текстовой меткой параметрам `row` и `column` было передано значение 0. Тем самым метка была помещена в левый верхний угол рамки.

Если элемент очень широк (пример тот же: длинная инструктирующая метка в этой программе), то ему можно позволить занять больше одной ячейки, чтобы при этом не нарушилось расположение остальных элементов. Параметр `columnspan`

«растягивает» элемент по горизонтали, помещая его более чем в один столбец. Я передал этому параметру значение 2, чтобы метка расположилась в двух соседних ячейках: координаты одной — строка номер 0, столбец номер 0, а координаты второй — строка номер 0, столбец номер 1 (аналогичным образом параметр rowspan позволяет «растянуть» элемент по вертикали).

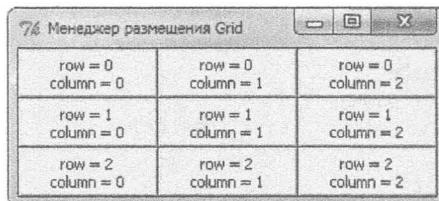


Рис. 10.13. Каждой кнопке отведена своя ячейка, «адрес» которой задают параметры row и column

Гибкость системы параметров не исчерпывается выбором места для элемента управления — ячейки или ячеек на сетке менеджера размещения. Можно также выровнять его внутри ячейки (или ячеек), пользуясь параметром sticky. Значения, которые принимает этот параметр, — направления: вверх (N), вниз (S), вправо (E) и влево (W). Элемент управления будет передвинут в ту долю ячейки (или ячеек), которая соответствует выбранному направлению. Поскольку я передал W параметру sticky при позиционировании объекта-метки, то последний выравнивается по «западному» — левому краю.

Я создам еще одну метку, которая будет размещена ниже на сетке и выровнена по левому краю:

```
# метка около поля ввода пароля
self.pw_lbl = Label(self, text = "Пароль: ")
self.pw_lbl.grid(row = 1, column = 0, sticky = W)
```

Создание текстового поля

Теперь следует создать элемент нового типа — Entry:

```
# текстовое поле для ввода пароля
self.pw_ent = Entry(self)
```

Этот код создает текстовое поле, в которое пользователь сможет вписать пароль. Элемент управления позиционируется так, чтобы поле ввода соседствовало справа с соответствующей меткой:

```
self.pw_ent.grid(row = 1, column = 1, sticky = W)
```

Следующая на очереди — кнопка, нажатием которой пользователь будет передавать программе свой вариант пароля:

```
# кнопка отправки значения
self.submit_btn = Button(self, text = "Узнать секрет", command = self.reveal)
```

Активизацию кнопки я связал с методом `reveal()`, открывающим пользователю секрет долголетия, если тот введет верный пароль.

Эта кнопка размещается на сетке еще ниже с левой стороны:

```
self.submit_btn.grid(row = 2, column = 0, sticky = W)
```

Создание текстовой области

Теперь я создам новый для вас элемент управления — `Text`:

```
# создание текстовой области, в которую будет выведен ответ
self.secret_txt = Text(self, width = 35, height = 5, wrap = WORD)
```

Значения, переданные параметрам `width` и `height`, определяют ширину и высоту текстовой области. Параметр `wrap` определяет механизм переноса текста, напечатанного внутри: доступные значения — `WORD`, `CHAR` и `NONE`. Примененное мной в данном случае значение `WORD` указывает, что на новую строку будут переноситься целые слова, заходящие за правую кромку текстовой области. Значение `CHAR` переносит текст посимвольно: при достижении края очередная буква появится уже на новой строке. Наконец, `NONE` означает отсутствие переноса и возможность помещать текст лишь на первой строке в текстовой области.

Теперь я сделаю так, чтобы текстовая область разместилась на следующей «строке» в менеджере и охватила два «столбца»:

```
self.secret_txt.grid(row = 3, column = 0, columnspan = 2, sticky = W)
```

Текстовые элементы: извлечение и вставка данных

Сейчас я реализую метод `reveal()`, который проверяет, введен ли пользователем правильный пароль. Если пароль введен, то программа сообщает секрет долголетия, а если нет, то заявляет, что пароль ошибочен.

Я получу текст, вписанный пользователем в поле ввода. Для этого надо вызвать метод `get()` экземпляра `Entry`:

```
def reveal(self):
    """ В зависимости от введенного пароля отвечает разными сообщениями. """
    contents = self.pw_ent.get()
```

Метод `get()` возвращает текстовое содержимое элемента. Он есть и у объекта `Entry`, и у `Text`. Я проверяю, равен ли полученный текст слову `secret`. Если да, то в ответном сообщении (строке, на которую ссылается переменная `message`) компьютер откроет секрет, как дожить до 100 лет. В противном случае пользователь должен быть извещен о том, что он ввел неправильный пароль.

```
if contents == "secret":
    message = "Чтобы дожить до 100 лет, надо сначала дожить до 99, " \
              "а потом вести себя ОЧЕНЬ осторожно."
else:
    message = "Вы ввели неправильный пароль, так что я не могу " \
              "поделиться тайной с вами."
```

Итак, готова строка, которой мы хотим поделиться с пользователем. Осталось вставить ее в текстовую область. Для этого я удаляю любое содержимое, которое могло ранее существовать внутри элемента Text. Это сделает метод `delete()`:

```
self.secret_txt.delete(0.0, END)
```

Метод удаляет текст из текстовых элементов. Он может принимать один или два параметра: одиночный индекс или точки начала и конца. Пары «координат», то есть номеров условных строк и столбцов внутри текстового поля или области, представляются десятичными дробями: слева от десятичной точки — номер строки, справа от нее — номер столбца. Так, например, в предшествующей строке я передал методу начальное значение `0.0`, имея в виду, что метод должен удалять текст, начиная с `0`-го по горизонтали и `0`-го по вертикали символа — с абсолютного начала текстовой области.

В `tkinter` есть несколько констант, упрощающих работу с данным методом, например константа `END`, означающая конец текста. Итак, показанная выше строка кода удаляет из текстовой области весь текст с начала и до конца. Метод `delete()` есть у элементов обоих типов: и `Text`, и `Entry`.

Теперь я вставлю строку, которую решил отобразить, в текстовую область:

```
self.secret_txt.insert(0.0, message)
```

Метод `insert()` вставляет строку в текстовый элемент. Он принимает два параметра: начальную позицию для вставки и имя переменной. Здесь я предлагаю методу вставить текст, начиная с «координат» `0.0`, то есть с самого начала. В качестве второго аргумента было передано `message`, так что в текстовой области отобразится подобающий ответ компьютера. Метод `insert()` также есть у обоих элементов.

ЛОВУШКА

Метод `insert()` не заменяет текст в текстовых элементах, а только добавляет его. Если вам угодно заменить существующий текст новым, то каждый раз вызывайте сначала метод `delete()`.

Обертка программы

В завершение программы я создам базовое окно, установлю его размеры и заголовок. Потом я создам новый объект `Application`, подчиненный базовому окну. Наконец, запущу событийный цикл окна, начиная работу программы.

```
# основная часть
root = Tk()
root.title("Долгожитель")
root.geometry("300x150")
app = Application(root)
root.mainloop()
```

Применение флагков

Флагки позволяют пользователю выбрать из группы альтернатив одну или несколько, вплоть до всех. Это очень гибкий механизм и вместе с тем удобный для

программиста, потому что пользователю разрешается выбирать лишь из закрытого списка.

Знакомство с программой «Киноман»

Программа «Киноман» позволяет пользователю выбрать один или несколько любимых жанров кино из трехэлементного списка: комедия, драма, мелодрама. Поскольку в программе использованы флажки, пользователь может указать столько жанров, сколько пожелает. Сделанный выбор будет отображен в области с текстом. Вид окна программы приводится на рис. 10.14.

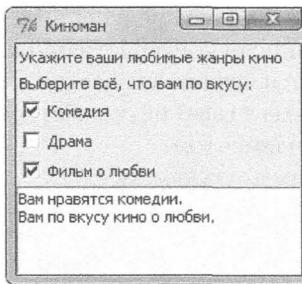


Рис. 10.14. Сделанный пользователем выбор отображается в текстовой области

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `movie_chooser.py`.

Настройка программы

В начале кода программы «Киноман» импортируется модуль `tkinter` и начинается объявление класса `Application`:

```
# Киноман
# Демонстрирует применение флажков
from tkinter import *
class Application(Frame):
    """ GUI-приложение, позволяющее выбрать любимые жанры кино. """
    def __init__(self, master):
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

Ссылка только на родительский объект элемента управления

Теперь я создам метку, описывающую назначение программы:

```
def create_widgets(self):
    """ Создает элементы, с помощью которых пользователь будет выбирать. """
```

```
# метка-описание
Label(self,
      text = "Укажите ваши любимые жанры кино"
      ).grid(row = 0, column = 0, sticky = W)
```

Между этой текстовой меткой и теми, что вы видели ранее, есть важное различие: с созданным объектом `Label` не связана никакая переменная. На первый взгляд это большая ошибка, потому что в данном случае прорисовка объекта, как может показаться, произойдет независимо от всего остального в программе. Однако в `tkinter` объект `Label`, как и все остальные элементы управления, связан с программой через родительский элемент. Это значит, что если я не нахожу необходимым прямо обращаться к элементу, то и связывать его с переменной мне тоже не нужно. Основной козырь такого подхода — сокращение и прояснение кода.

До сих пор я проявлял завидный консерватизм и каждый раз связывал элемент с переменной. Но здесь-то мне наверняка известно, что запрашивать эту метку ни о чем я не буду. Поэтому на объект `Label` не ссылается ни одна переменная, а он сам связан лишь с родительским элементом.

Подобным образом я создам другую метку:

```
# метка-инструкция
Label(self,
      text = "Выберите все, что вам по вкусу:"
      ).grid(row = 1, column = 0, sticky = W)
```

Эта метка сообщает пользователю, что тот может не ограничивать свои симпатии одной альтернативой и установить несколько флагжков.

Создание флагжков

Далее я создам три флагжка — по одному на каждый популярный жанр. Сначала разберемся с тем из флагжков, к которому будет тянуться рука любителя кинокомедий.

Каждый флагжок нуждается в специальном объекте, связанном с ним, который бы автоматически отображал текущий статус флагжка. Этот специальный объект должен быть экземпляром класса `BooleanVar` из модуля `tkinter`. Поэтому еще до самого флагжка «Комедия» я создам объект `BooleanVar` и свяжу его с атрибутом `likes_comedy`, также вновь созданным:

```
# флагжок "Комедия"
self.likes_comedy = BooleanVar()
```

НА САМОМ ДЕЛЕ

Логические (булевы) переменные — это переменные особого рода, которые могут быть только истинными или ложными. Программисты иногда называют их булевами. По-английски это слово пишется с заглавной буквы, потому что происходит от фамилии математика Джорджа Буля.

А вот и сам флагжок:

```
Checkbutton(self,
            text = "Комедия",
            variable = self.likes_comedy,
```

```
        command = self.update_text
    ).grid(row = 2, column = 0, sticky = W)
```

Этот код создает новый флагжок, рядом с которым написано **Комедия**. Передав `self.likes_comedy` параметру `variable`, я связал статус флагжка («установлен» или «снят») с атрибутом `likes_comedy`. А поскольку параметру `command` передано `self.update_text()`, при активизации (или деактивизации) флагжка пользователем будет вызываться метод `update_text()`. Этот флагжок будет расположен слева внизу.

Заметьте, что полученный объект `Checkbutton` не связан ни с какой переменной. Нас будет интересовать только статус флагжка, который можно узнать через атрибут `likes_comedy`.

Таким же образом реализованы еще два флагжка:

```
# флагжок "Драма"
self.likes_drama = BooleanVar()
Checkbutton(self,
            text = "Драма",
            variable = self.likes_drama,
            command = self.update_text
).grid(row = 3, column = 0, sticky = W)

# флагжок "Фильм о любви"
self.likes_romance = BooleanVar()
Checkbutton(self,
            text = "Фильм о любви",
            variable = self.likes_romance,
            command = self.update_text
).grid(row = 4, column = 0, sticky = W)
```

Каждый раз, когда пользователь устанавливает или снимает флагжки **Драма** и **Фильм о любви**, к ним применяется метод `update_text()`. Несмотря на то что эти объекты `Checkbutton` не связаны с какими-либо переменными, текущий статус флагжка **Драма** доступен в атрибуте `likes_drama`, а флагжка **Фильм о любви** — в атрибуте `likes_romance`.

Кроме того, я создам текстовую область, в которую буду выводить пользовательский выбор:

```
# текстовая область с результатами
self.results_txt = Text(self, width = 40, height = 5, wrap = WORD)
self.results_txt.grid(row = 5, column = 0, columnspan = 3)
```

Получение статуса флагжка

Теперь надо реализовать метод `update_text()`, который будет менять содержимое текстовой области в зависимости от того, какие флагжки выбрал пользователь:

```
def update_text(self):
    """ Обновляет текстовый элемент по мере того, как пользователь выбирает свои
    любимые киножанры. """
    likes = ""
    if self.likes_comedy.get():
        likes += "Комедия"
    if self.likes_drama.get():
        likes += "Драма"
    if self.likes_romance.get():
        likes += "Фильм о любви"
    self.results_txt.delete(1.0, END)
    self.results_txt.insert(1.0, likes)
```

```

    likes += "Вам нравятся комедии.\n"
if self.likes_drama.get():
    likes += "Вас привлекает жанр драмы.\n"
if self.likes_romance.get():
    likes += "Вам по вкусу кино о любви."
self.results_txt.delete(0.0, END)
self.results_txt.insert(0.0, likes)

```

Значения объектов BooleanVar недоступны напрямую: надо вызывать метод `get()` соответствующих объектов. В показанном выше коде метод `get()` объекта `BooleanVar`, на который ссылается атрибут `likes_comedy`, использован, чтобы узнать статус флагшка. Если значение окажется равным `True`, то флагшок **Комедия** выбран и строку "Вам нравятся комедии.\n" можно добавить в конструируемый текст, который затем отобразится в текстовой области. Подобным же образом опрашиваются флагшки **Драма** и **Кино о любви**. Наконец, я удаляю все содержимое текстовой области и вставляю новое.

Обертка программы

Программа завершает знакомый фрагмент, в котором создается базовое окно и дочерний объект `Application`, после чего запускается событийный цикл окна.

```

# основная часть
root = Tk()
root.title("Киноман")
app = Application(root)
root.mainloop()

```

Применение переключателей

Переключатели отличаются от флагков только тем, что выбрать можно не более чем одно положение из группы. Это удобно, если мы хотим, чтобы пользователь совершил единственный выбор из нескольких альтернатив. Ввиду сходства с флагками переключатели не представляют для нас труда как объект изучения.

Знакомство с программой «Киноман-2»

Программа «Киноман-2» во всем похожа на свою предшественницу. Пользователю так же предлагаются три жанра фильмов на выбор, но теперь, поскольку вместо флагков используется переключатель, выбрать можно лишь один вариант. Чтобы подстроиться под ситуацию, программа просит пользователя назвать один любимый жанр.

Окно программы показано на рис. 10.15.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `movie_chooser2.py`.

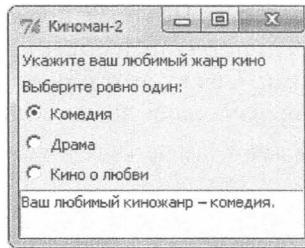


Рис. 10.15. Теперь можно выбрать только один кинематографический жанр

Настройка программы

В начале программы импортируется модуль `tkinter`:

```
# Киноман-2
# Демонстрирует переключатель
from tkinter import *
```

Затем создается класс `Application`. Я объявляю его конструктор, который инициализирует новый объект `Application`:

```
class Application(Frame):
    """ GUI-приложение, позволяющее выбрать один любимый жанр кино. """
    def __init__(self, master):
        """ Инициализирует рамку. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

Теперь я создаю метки, по которым пользователь будет ориентироваться при работе с программой:

```
def create_widgets(self):
    """ Создает элементы, с помощью которых пользователь будет выбирать. """
    # метка-описание
    Label(self,
          text = "Укажите ваш любимый жанр кино"
          ).grid(row = 0, column = 0, sticky = W)
    # метка-инструкция
    Label(self,
          text = "Выберите ровно один:"
          ).grid(row = 1, column = 0, sticky = W)
```

Создание переключателя

Из нескольких положений переключателя пользователь может выбрать одноединственное, а значит, нецелесообразно хранить состояния положений переключателя, образующих группу, в отдельных переменных, как у флагков. Вместо этого переключателю ставится в соответствие специальный объект — экземпляр класса `StringVar` из модуля `tkinter`. В таком объекте можно сохранить строковое

значение и потом запрашивать о нем. Поэтому, прежде чем создать сами положения переключателя, я создам единственный объект `StringVar`, ссылка на который будет распределена между ними. Свяжу этот объект с атрибутом `favorite` и с помощью метода `set()` установлю начальное значение `None`:

```
# переменная для хранения сведений о единственном любимом жанре
self.favorite = StringVar()
self.favorite.set(None)
```

Вот теперь я создам положение **Комедия** переключателя:

```
# положение "Комедия" переключателя
Radiobutton(self,
            text = "Комедия",
            variable = self.favorite,
            value = "комедия.",
            command = self.update_text
            ).grid(row = 2, column = 0, sticky = W)
```

Параметр `variable` ссылается на нашу специальную переменную `self.favorite`, а параметр `value` показывает, какое значение будет сохранено в этой переменной, когда пользователь установит переключатель в данное положение. Значение `value` равно `"комедия."`, а следовательно, когда пользователь выберет положение **Комедия**, в переменной `StringVar`, на которую ссылается `self.favorite`, тоже окажется строковое значение `"комедия."`.

Аналогично создаются два других положения переключателя:

```
# положение "Драма" переключателя
Radiobutton(self,
            text = "Драма",
            variable = self.favorite,
            value = "драма.",
            command = self.update_text
            ).grid(row = 3, column = 0, sticky = W)

# положение "Кино о любви" переключателя
Radiobutton(self,
            text = "Кино о любви",
            variable = self.favorite,
            value = "кино о любви.",
            command = self.update_text
            ).grid(row = 4, column = 0, sticky = W)
```

Параметр `variable` для положения **Драма** переключателя я сделал равным `self.favorite`, а параметр `value` — равным `"драма."`. Это значит, что при выборе этого положения объект `StringVar`, на который ссылается `self.favorite`, окажется равным строке `"драма."` Совершенно те же соображения действуют в случае с положением **Кино о любви** переключателя.

Необходимо также создать текстовую область, в которой будет отображаться пользовательский выбор:

```
# текстовая область с результатами
self.results_txt = Text(self, width = 40, height = 5, wrap = WORD)
self.results_txt.grid(row = 5, column = 0, columnspan = 3)
```

Доступ к значениям в переключателе

Чтобы получить значение из переключателя, достаточно вызвать метод `get()` единого для всех его положений объекта `StringVar`:

```
def update_text(self):
    """ Обновляя текстовую область, вписывает в нее любимый жанр. """
    message = "Ваш любимый киножанр - "
    message += self.favorite.get()
```

Если выбрано положение **Комедия** переключателя, то `self.favorite.get()` возвратит строку "комедия.", если **Драма** — "драма.". Закономерно, что если выбрана альтернатива **Кино о любви**, то `self.favorite.get()` возвратит "кино о любви.".

Осталось удалить текст, который, возможно, ранее был в текстовой области, и добавить туда вновь созданную строку — информацию о кинематографическом вкусе пользователя:

```
self.results_txt.delete(0.0, END)
self.results_txt.insert(0.0, message)
```

Обертка программы

Как всегда, напоследок я создаю базовое окно и новый объект `Application`, а потом запускаю событийный цикл окна, чтобы начать работу GUI.

```
# основная часть
root = Tk()
root.title("Киноман-2")
app = Application(root)
root.mainloop()
```

Вернемся к программе «Сумасшедший сказочник»

Теперь, познакомившись с множеством разных элементов управления по отдельности, вы наверняка сумеете применить их и вместе, в составе одного большого GUI. В программе «Сумасшедший сказочник» не вводится никаких новых понятий, поэтому ее код я почти не комментировал. Код вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 10. Файл называется `mad_lib.py`.

Импорт модуля `tkinter`

Вы уже, наверное, привыкли, что прежде, чем пользоваться модулем `tkinter`, его надо импортировать:

```
# Сумасшедший сказочник
# Создает рассказ на основе пользовательского ввода
from tkinter import *
```

Метод-конструктор класса Application

Как и все остальные конструкторы аналогичных классов, этот метод инициализирует вновь созданный объект Application и вызывает его метод create_widgets():

```
class Application(Frame):
    """ GUI-приложение, создающее рассказ на основе пользовательского ввода. """
    def __init__(self, master):
        """ Инициализирует рамку. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

Метод create_widgets() класса Application

Этот метод создает внутри GUI все необходимые элементы управления. Новое здесь только то, что три положения переключателя создаются с помощью цикла, который, перебирая список строк, устанавливает значения атрибутов text и value:

```
def create_widgets(self):
    """ Создает элементы управления, с помощью которых пользователь будет вводить исходные данные и получать готовый рассказ. """
    # метка с текстом-инструкцией
    Label(self,
          text = "Введите данные для создания нового рассказа"
          ).grid(row = 0, column = 0, columnspan = 2, sticky = W)
    # метка и поле ввода для имени человека
    Label(self,
          text = "Имя человека: "
          ).grid(row = 1, column = 0, sticky = W)
    self.person_ent = Entry(self)
    self.person_ent.grid(row = 1, column = 1, sticky = W)
    # метка и поле ввода для существительного
    Label(self,
          text = "Существительное во мн. ч.:"
          ).grid(row = 2, column = 0, sticky = W)
    self.noun_ent = Entry(self)
    self.noun_ent.grid(row = 2, column = 1, sticky = W)
    # метка и поле ввода для глагола
    Label(self,
          text = "Глагол в инфинитиве:"
          ).grid(row = 3, column = 0, sticky = W)
    self.verb_ent = Entry(self)
    self.verb_ent.grid(row = 3, column = 1, sticky = W)
    # метка при группе флажков с прилагательными
    Label(self,
          text = "Прилагательное (-ые):"
          ).grid(row = 4, column = 0, sticky = W)
    # флажок со словом "недерпеливый"
    self.is_itchy = BooleanVar()
    Checkbutton(self,
```

```

text = "нетерпеливый",
variable = self.is_itchy
).grid(row = 4, column = 1, sticky = W)
# флажок со словом "радостный"
self.is_joyous = BooleanVar()
Checkbutton(self,
            text = "радостный",
            variable = self.is_joyous
            ).grid(row = 4, column = 2, sticky = W)
# флажок со словом "пронизывающий"
self.is_electric = BooleanVar()
Checkbutton(self,
            text = "пронизывающий",
            variable = self.is_electric
            ).grid(row = 4, column = 3, sticky = W)
# метка при переключателе с названиями частей тела
Label(self,
       text = "Body Part:")
       ).grid(row = 5, column = 0, sticky = W)
# переменная, содержащая название одной из частей тела
self.body_part = StringVar()
self.body_part.set(None)
# переключатель с названиями частей тела
body_parts = ["пупок", "большой палец ноги", "продолговатый мозг"]
column = 1
for part in body_parts:
    Radiobutton(self,
                text = part,
                variable = self.body_part,
                value = part
                ).grid(row = 5, column = column, sticky = W)
    column += 1
# кнопка отсылки данных
Button(self,
       text = "Получить рассказ",
       command = self.tell_story
       ).grid(row = 6, column = 0, sticky = W)
self.story_txt = Text(self, width = 75, height = 10, wrap = WORD)
self.story_txt.grid(row = 7, column = 0, columnspan = 4)

```

Метод tell_story() класса Application

Этот метод извлекает в переменные те значения, которые ввел пользователь, и на их основе создает одну длинную строку — рассказ. Потом из текстовой области удаляется текст, который, возможно, существовал там ранее, а созданный пользователем рассказ вставляется на его место.

```

def tell_story(self):
    """ Заполняет текстовую область очередной историей на основе пользовательского
    ввода. """

```

```

# get values from the GUI
person = self.person_ent.get()
noun = self.noun_ent.get()
verb = self.verb_ent.get()
adjectives = ""
if self.is_itchy.get():
    adjectives += "нетерпеливое, "
if self.is_joyous.get():
    adjectives += "радостное, "
if self.is_electric.get():
    adjectives += "пронизывающее, "
body_part = self.body_part.get()
# создание рассказа
story = "Знаменитый путешественник "
story += person
story += " уже совсем отчаялся довершить дело всей своей жизни - поиск затерян-
ного города, в котором, по легенде, обитали "
story += noun.title()
story += ". Но однажды "
story += noun
story += " и "
story += person + " столкнулись лицом к лицу. "
story += "Мощное. "
story += adjectives
story += "ни с чем не сравнимое чувство охватило душу путешественника. "
story += "После стольких лет поисков цель была наконец достигнута. "
story += person
story += " ощущил, как на его " + body_part + " скатилась слезинка. "
story += "И тут внезапно "
story += noun
story += " перешли в атаку, и "
story += person + " был ими мгновенно сожран. "
story += "Мораль? Если задумали "
story += verb
story += ". надо делать это с осторожностью."
# вывод рассказа на экран
self.story_txt.delete(0.0, END)
self.story_txt.insert(0.0, story)

```

Основная часть программы

Этот код вы видели уже много раз. Создаются базовое окно и экземпляр класса Application, после чего весь графический интерфейс вступает в работу при вызове метода mainloop() базового окна.

```

# основная часть
root = Tk()
root.title("Сумасшедший сказочник")
app = Application(root)
root.mainloop()

```

Резюме

В этой главе вы научились создавать графические интерфейсы. Сначала вы познакомились с событийно-ориентированным программированием — новой для вас парадигмой разработки кода. Затем вы увидели в деле несколько графических элементов управления, в том числе рамки, кнопки, поля и области текстового ввода, флагшки и переключатели. Вы узнали, как настраивать элементы управления для своих нужд и как располагать их внутри рамки, пользуясь менеджером размещения Grid. Вы научились связывать события с обработчиками, благодаря которым элементы могут как-то реагировать при активизации. В заключение я показал вам, как может быть реализована забавная компьютерная программа-повествователь со сложным GUI.

ЗАДАЧИ

- Напишите собственную версию «Сумасшедшего сказочника», в которой система элементов управления внутри окна будет другой.
- Перепишите игру «Отгадай число» из главы 3: создайте для нее графический интерфейс.
- Напишите программу «Счет, пожалуйста!». Она должна показать пользователю несложное ресторальное меню с блюдами и ценами, принять его заказ и вывести на экран сумму счета.

11 Графика. Игра «Паника в пиццерии»

Большинство программ, с которыми вы знакомились ранее, преподносило пользователю только текст. Но от всех современных приложений избалованный пользователь ждет богатой графики. Из этой главы вы узнаете, как создавать графику с помощью нескольких мультимедийных модулей, предназначенных для разработки игр в Python. Конкретно вы научитесь делать следующее:

- создавать графическое окно;
- создавать спрайты и манипулировать ими;
- отображать текст в графическом окне;
- отслеживать столкновения между спрайтами;
- обрабатывать пользовательский ввод, осуществляемый с помощью мыши;
- управлять действиями компьютерного противника.

Знакомство с игрой «Паника в пиццерии»

Проект, разработка которого посвящена эта глава, представляет собой игру «Паника в пиццерии». По сюжету игры в одной пиццерии случилась неприятность: шеф-повара до последнего градуса ярости довел наглых привередливых посетителей и он взобрался на крышу здания и стал оттуда разбрасывать готовую пиццу. Конечно, нельзя допустить такого перевода продуктов: пицца должна быть спасена. С помощью мыши игрок может передвигать по экрану глубокую сковородку и с ее помощью ловить падающую пиццу. С каждой следующей пойманной пицци количество очков на счету у игрока увеличивается, но как только пицца касается земли, игра заканчивается. Игровой процесс отражен на рис. 11.1 и 11.2.

Знакомство с пакетами `pygame` и `livewires`

Под названиями `pygame` и `livewires` скрываются наборы модулей (так называемые *пакеты*), дающие программисту на Python доступ к широкому диапазону мультимедийных классов. С помощью этих классов можно создавать программы с графикой, анимацией, звуковыми эффектами и музыкой. Названные пакеты также

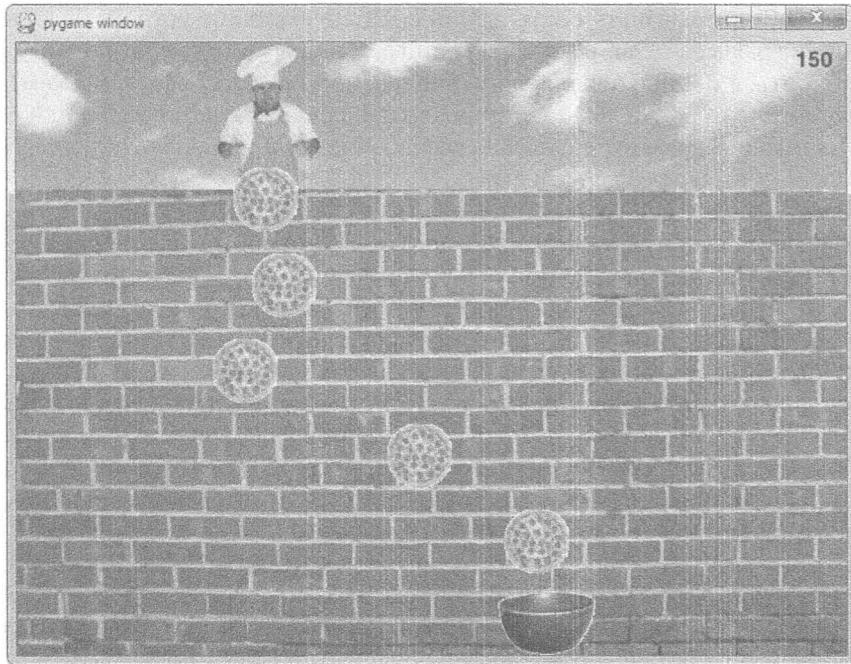


Рис. 11.1. Игрок должен ловить падающие круги пиццы

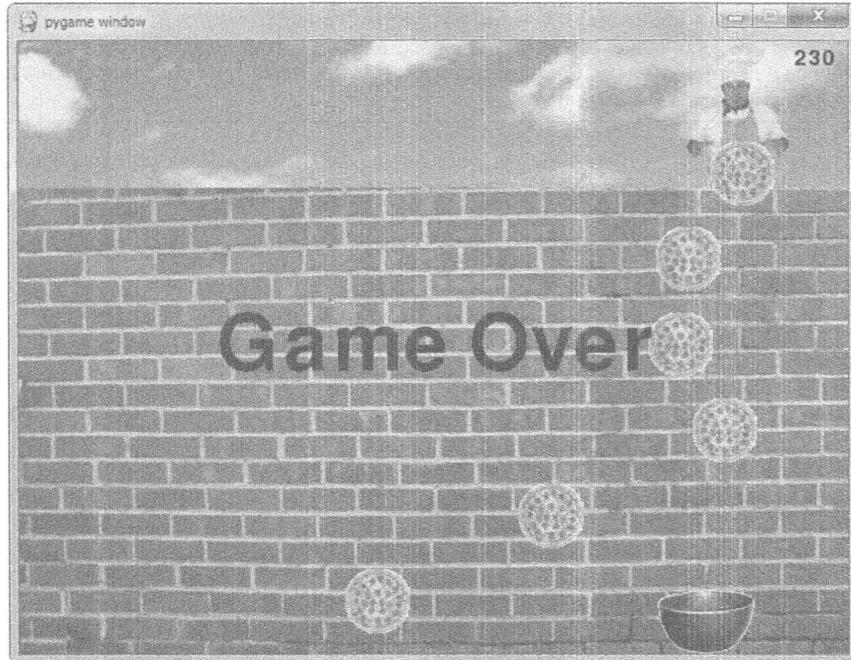


Рис. 11.2. Не поймал — игра закончилась

позволяют принимать пользовательский ввод с нескольких устройств: клавиатуры, мыши и др. Имея в своем распоряжении `pygame` и `livewires`, программист может не беспокоиться о «низкоуровневой» аппаратной составляющей, скажем, о том, какая на компьютере у игрока установлена видеокарта и установлена ли она вообще. Достаточно сосредоточиться на реализации внутренней логики программы. Это позволяет быстро создавать игры.

Пакет `pygame`, детище Пита Шиннерса (Pete Shinners), — оружие, которое достойно украсить ваш мультимедийный арсенал. С его помощью вы сможете создавать на Python впечатляющие приложения с графикой и звуком. Мощь этого пакета такова, что начинающих программистов она зачастую ошеломляет.

В свою очередь, `livewires` — пакет, разработанный в Англии группой преподавателей программирования, предназначен как раз для того, чтобы, не поступаясь функциональностью `pygame`, упростить работу с мультимедиа для программиста. Пакет `livewires` позволяет без особых трудностей начать программировать игры с графикой и звуком. Несмотря на то что в программах на основе `livewires` не осуществляется прямого доступа к `pygame`, этот последний все же неявно задействован в них и выполняет основную часть работы.

Прежде чем запускать программы, описанные в этой главе, вы должны установить `pygame` и `livewires`. Пригодные к использованию версии обоих пакетов можно найти на сайте-помощнике (www.courseptr.com/downloads) в папке **Software**. Следуйте инструкциям по установке, которыми оснащены оба пакета.

ЛОВУШКА

Вам будет полезно ознакомиться с сайтом организации LiveWires (<http://www.livewires.org.uk>), но помните, что размещенный там оригиналный продукт LiveWires отличается от того пакета `livewires`, который есть на сайте-помощнике. Я немного доработал пакет, чтобы начинающему программисту было еще проще пользоваться им. Соответственно, и документация к пакету тоже изменилась; она содержится в приложении В к этой книге.

Чтобы ближе познакомиться с пакетом `pygame`, посетите сайт <http://www.pygame.org>.

Создание графического окна

Прежде чем выводить графику на экран, надо создать графическое окно¹ — пустой «холст», на котором будут отображаться картинки и текст.

Знакомство с программой «Новое графическое окно»

Нет ничего сложного в том, чтобы создать графическое окно с помощью пакета `livewires`. Программа «Новое графическое окно», которая решает эту задачу, состоит всего из нескольких строк кода. Результат работы программы показан на рис. 11.3.

¹ Здесь и далее автор пользуется терминами «графическое окно» и «графический экран» как синонимичными. — Примеч. пер.



Рис. 11.3. Мое первое графическое окно. Пустое, но мое

ЛОВУШКА

Как и программы, в которых для создания окна привлекается модуль Tkinter, программы на основе livewires не следует запускать из IDLE. Если вы пользователь Windows, создайте пакетный файл, который будет запускать вашу программу и затем делать паузу. Как написать такой пакетный файл, вам напомнит раздел «Знакомство с программой “Простейший GUI”» главы 10.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется new_graphics_window.py.

Импорт модуля games

Пакет livewires состоит из нескольких важных модулей. В их числе модуль games, содержащий группу объектов и классов для программирования игр. Чтобы импортировать из пакета какой-либо определенный модуль, воспользуйтесь служебным словом from. Надо написать так: from, затем имя пакета, import и имя модуля (или список имен, разделенных запятыми).

Первым делом в демонстрируемой программе я загружаю модуль games из пакета livewires:

```
# Новое графическое окно
# Демонстрирует создание графического окна
from livewires import games
```

Теперь я могу пользоваться `games` точно так же, как и любым другим импортированным модулем.

Содержимое модуля `games` описано в табл. 11.1 и 11.2. Там перечислены соответственно объекты и классы, которые пригодятся вам в программировании игр.

Таблица 11.1. Полезные объекты модуля `games`

Объект	Описание
<code>screen</code>	Предоставляет доступ к «графическому экрану» (<code>graphics screen</code>) — области, в которой помещаются, движутся и взаимодействуют графические объекты
<code>mouse</code>	Предоставляет доступ к мыши
<code>keyboard</code>	Предоставляет доступ к клавиатуре

Таблица 11.2. Полезные классы модуля `games`

Класс	Описание
<code>Sprite</code>	Класс для создания спрайтов — объектов, которые могут отображаться на графическом экране
<code>Text</code>	Подкласс <code>Sprite</code> . Для текстовых объектов, отображаемых на графическом экране
<code>Message</code>	Подкласс <code>Text</code> . Для текстовых объектов, отображаемых на графическом экране в течение фиксированного промежутка времени

Инициализация графического экрана

Следующим шагом я инициализирую графический экран:

```
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Каждый раз при вызове функции `init()` из модуля `games` создается новый графический экран. Его ширину определяет значение, переданное параметру `screen_width`, а высоту, как нетрудно догадаться, значение `screen_height`. Размеры графического экрана измеряются в пикселях, то есть отдельных точках. Значение, которое принимает параметр `fps` (сокращение от frames per second — «кадров в секунду»), — это количество обновлений графического экрана в течение одной секунды.

Запуск основного цикла

Вот заключительная строка в коде программы:

```
games.screen.mainloop()
```

Здесь `screen` — это объект из модуля `games`, представляющий графический экран. Вызов метода `mainloop()` заставляет окно обновляться `fps` раз каждую секунду. Итак, заключительная строка кода ответственна за то, чтобы графическое окно оставалось открытым, а его содержимое перерисовывалось 50 раз в секунду. Некоторые свойства объекта `screen` перечислены в табл. 11.3, а некоторые его методы — в табл. 11.4.

Таблица 11.3. Полезные свойства объекта screen

Свойство	Описание
width	Ширина графического экрана
height	Высота экрана
fps	Частота обновления экрана (количество раз в секунду)
background	Фоновое изображение экрана
all objects	Список всех спрайтов, содержащихся на экране
event_grab	Булева переменная, которая показывает, перенаправляется ли пользовательский ввод на экран. Если да, то равна True, если нет — False

Таблица 11.4. Полезные методы объекта screen

Метод	Описание
add(sprite)	Добавляет объект класса Sprite (или одного из его подклассов) на графический экран
clear()	Удаляет все спрайты с экрана
mainloop()	Запускает основной цикл работы экрана
quit()	Закрывает графическое окно

Назначение фоновой картинки

Пустой графический экран хороший разве что как кандидат на звание самой бесполезной программы в мире. Надо бы чем-нибудь его заполнить. Для этого у объекта screen есть свойство, позволяющее назначить фоновое изображение.

Знакомство с программой «Фоновая картинка»

«Фоновая картинка» — это частная версия программы «Новое графическое окно». Как можно видеть на рис. 11.4, на графический экран было добавлено фоновое изображение.

Чтобы создать программу «Фоновая картинка», я доработал «Новое графическое окно», добавив всего две строки кода перед вызовом mainloop().

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется background_image.py.

```
# Фоновая картинка
# Демонстрирует назначение фоновой картинки для графического экрана
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
wall_image = games.load_image("wall.jpg", transparent = False)
games.screen.background = wall_image
games.screen.mainloop()
```

Загрузка изображения

Прежде чем как-либо манипулировать изображением, например назначать его в качестве фона для графического экрана, надо это изображение загрузить в память

и создать объект соответствующего типа. Я загрузил картинку с помощью следующего кода, который идет сразу за инициализацией графического окна:

```
wall_image = games.load_image("wall.jpg", transparent = False)
```

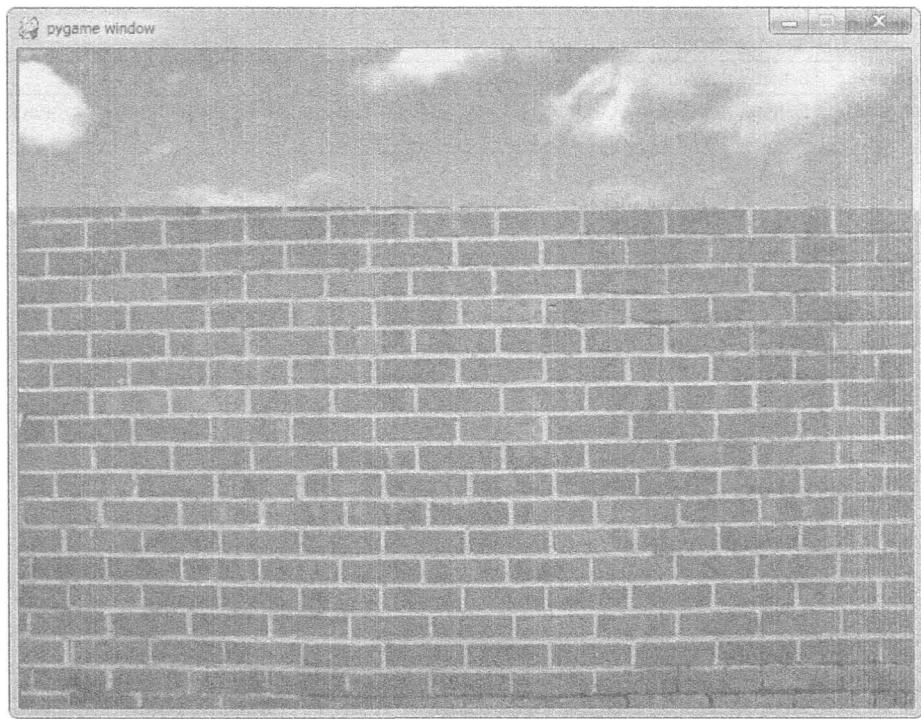


Рис. 11.4. С помощью свойства `background` объекта `screen` я оснастил пустое графическое окно фоновой картинкой

Таким образом вызывается функция `load_image()` модуля `games`, которая загружает в память картинку из файла `wall.jpg` и связывает полученный объект-изображение с переменной `wall_image`.

ЛОВУШКА

Убедитесь, что путь к файлу, которым вы хотите воспользоваться в программе, указан полно и правильно, как описано в разделе «Чтение текстового файла» главы 7. Проще всего, как я и поступил, хранить файлы картинок в той же папке, что и код программы, которая к ним обращается. Если следовать этой системе, то о путях к файлам можно вообще не беспокоиться.

Функция `load_image()` принимает два аргумента: первый (строковый) — имя файла, а второй (булев) — значение параметра `transparent`. Чуть дальше в этой главе я объясню, что, собственно, значит `transparent`. Пока вам будет достаточно запомнить такое правило: фоновые изображения всегда загружаются с аргументом `transparent = False`.

Вы, наверное, заметили, что в качестве фоновой в этой программе я загрузил картинку с расширением JPEG. Для функции `load_image()` это не единственно воз-

можный выбор. Она способна загружать также файлы многих других типов, например BMP, GIF, PNG, PCX и TGA.

Установка фона

Для того чтобы сделать какой-либо объект-изображение фоновой картинкой графического экрана, надо воспользоваться свойством `background` объекта `screen`. Вот почему после загрузки изображения я выполняю следующее:

```
games.screen.background = wall_image
```

В результате работы этого кода фоном графического экрана становится объект-изображение `wall_image`.

Когда программа доходит до команды `mainloop()`, графическое окно открывается уже с фоновой картинкой.

Что такое система координат графики

Итак, мы создали уже несколько графических экранов, но все шириной 640 и высотой 480 пикселов; никаких нововведений в этой части не предпринималось. Теперь ближе рассмотрим систему координат графического экрана.

Графический экран можно представить себе как сетку или решетку на 640 столбцов по ширине и 480 строк по высоте. Каждое пересечение строки со столбцом — это точка на экране, один пиксель. Вот почему, описывая местоположение чего-либо на экране, пользуются двумя координатами: x (номер столбца) и y (номер строки). Начало отсчета координат — левый верхний угол графического экрана. Поэтому крайняя слева и сверху точка экрана обладает координатами x и y , равными нулю, что может быть записано в виде пары $(0; 0)$. При движении вправо возрастают значения x , при движении вниз — значения y . Таким образом, противоположный край графического экрана, правый нижний угол, будет иметь координаты $(639; 479)$. На рис. 11.5 наглядно показана система координат графического экрана.

Система координат позволяет позиционировать на экране графические объекты, например изображение пиццы или крупную надпись *Game Over*, выделенную красным шрифтом: центр графического объекта связывается с определенной парой координат. Как это работает на практике, вы увидите в следующей программе.

Отображение спрайта

Фоновая картинка, бесспорно, способна украсить мрачный и пустой графический экран, но, какой бы восхитительной она ни была, не стоит забывать, что это всего лишь статическое изображение. Графический экран с одним только фоном похож на пустую сцену, которой недостает актеров. Вызовем же на сцену спрайты!



Рис. 11.5. Точки на графическом экране описываются парами чисел: абсциссой и ординатой

Спрайтом называется графический объект с картинкой в своем составе, буквально «оживляющий» программу. Спрайты применяются в играх, развлекательных приложениях, презентациях и на различных страницах в Интернете. В сущности, «Паника в пиццерии» уже познакомила вас со спрайтами: тронувшийся умом повар, сковородка и летающие круги пиццы — это все они.

НА САМОМ ДЕЛЕ

Спрайты предназначены не только для игр. В десятках приложений отнюдь не развлекательной направленности ими пользуются к месту и не к месту. Пожалуй, самый печально известный спрайт в истории IT-индустрии вам наверняка знаком: это Скрепыш — анимированная канцелярская скрепка, которую создатели Microsoft Office предназначали в помощь пользователям. Увы, многим Скрепыш пришелся не по нраву за его экстравагантный вид и назойливость. В одном влиятельном блоге даже появилась статья под названием «Убить Скрепыша!». Наконец разработчики Microsoft сообразили, что к чему, и в Office 2007 в числе помощников Скрепыша не стало. Из этого примера вы должны извлечь такой урок: графика украсит вашу программу, но только если возможности спрайтов обращены во благо, а не во зло.

Вместо того чтобы сразу создать много спрайтов, которые будут красиво летать и сталкиваться, начну с малого: выведу на экран один неподвижный спрайт.

Знакомство с программой «Спрайт-пицца»

В программе «Спрайт-пицца», как и в предшествующих примерах, создается графическое окно и назначается фоновая картинка, однако вслед за этим ровно посередине экрана создается спрайт с изображением круга пиццы. Результаты работы программы показаны на рис. 11.6.

«Спрайт-пицца» — модификация «Фоновой картинки», к коду которой непосредственно перед вызовом `mainloop()` были добавлены всего три строки, призванные поместить спрайт на экран. Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется `pizza_sprite.py`.

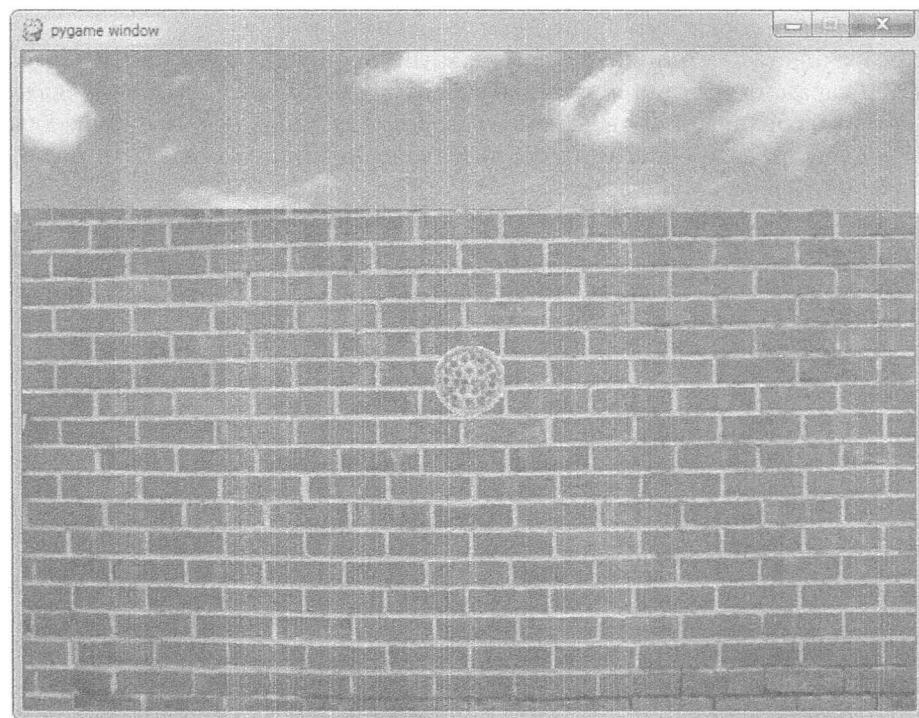


Рис. 11.6. Изображение пиццы — не часть фона, а самостоятельный объект, экземпляр класса `Sprite`

```
# Спрайт-пицца
# Демонстрирует создание спрайта
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
wall_image = games.load_image("wall.jpg", transparent = False)
games.screen.background = wall_image
pizza_image = games.load_image("pizza.bmp")
pizza = games.Sprite(image = pizza_image, x = 320, y = 240)
games.screen.add(pizza)
games.screen.mainloop()
```

Загрузка изображения для спрайта

Сначала я загружу картинку с пиццей в оперативную память, чтобы создать объект-изображение:

```
pizza_image = games.load_image("pizza.bmp")
```

Вы должны были заметить одно маленькое отличие в способе загрузки: на этот раз я не установил значение параметра `transparent`. По умолчанию он равен `True`, так что картинка загружается прозрачной. В этом случае через прозрачные участки изображения, помещенного на графический экран, просвечивает фон. Для спрайтов неправильной формы — непрямоугольных и «дырявых» — это очень удобно: например, представьте спрайт с изображением головки сыра.

Прозрачность тех или иных частей картинки определяется их цветом. Если включен параметр `transparent`, то «прозрачным» цветом в изображении окажется тот, который обнаружится в верхнем левом углу картинки. Все участки изображения, заполненные этим цветом, будут визуально проницаемыми: сквозь них будет виден фон графического экрана.

На рис. 11.7 показан кусок швейцарского сыра на сплошном белом фоне.

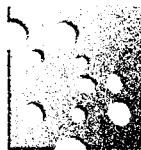


Рис. 11.7. Спрайт с куском сыра на непрозрачном белом фоне. Скоро в игру вступит прозрачность

Если загрузить эту картинку с включенным параметром `transparent`, то все ее участки, залитые белым цветом (тем же, что и в верхнем левом углу), окажутся прозрачными при отображении спрайта на графическом экране; через них будет просвечивать фоновый рисунок. На рис. 11.8 показано, как выглядит картинка, загруженная в «прозрачном» и «непрозрачном» режимах. Как правило, если вы пользуетесь спрайтами, фоновая картинка должна быть непрозрачной и не совпадающей по цвету с рисунками, которые вы будете помещать поверх нее.

ЛОВУШКА

Убедитесь, что собственно рисунок, предназначенный вами для спрайта, не содержит «прозрачного» цвета. Иначе закрашенные этим цветом участки тоже станут прозрачными, и у пользователя сложится впечатление, что это дырки или прорези, сквозь которые виден фон.

Создание спрайта

Теперь я наконец создам спрайт с изображением пиццы:

```
pizza = games.Sprite(image = pizza_image, x = 320, y = 240)
```

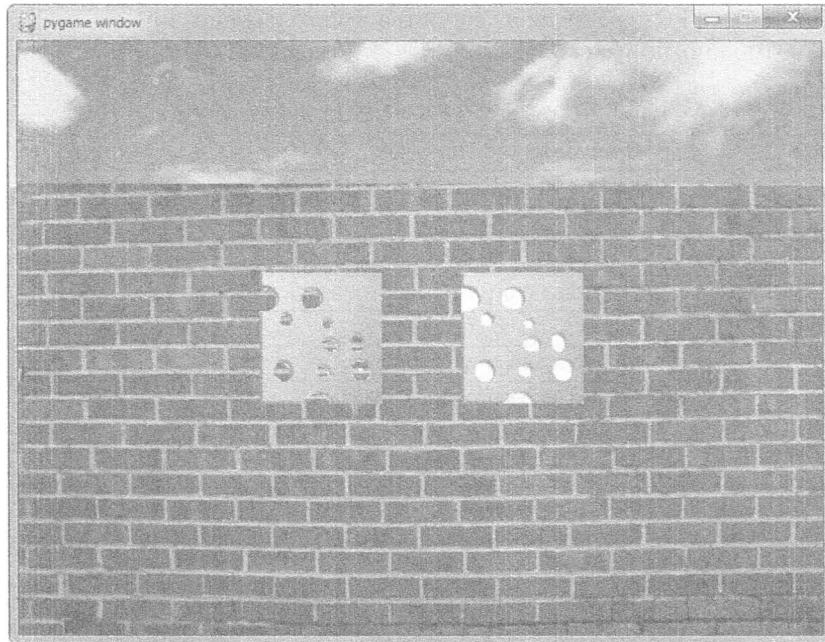


Рис. 11.8. Слева — изображение с параметром transparent, равным True, справа — равным False

Создан новый объект Sprite с изображением круга пиццы и парой координат (320; 240), которая точно соответствует центру графического окна. С этим новым объектом связана переменная pizza. Создавая объект Sprite, вы должны передать конструктору класса по крайней мере исходное изображение, абсциссу (x) и ординату (y).

Добавление спрайта на экран

Создав спрайт, вы должны добавить его на экран, чтобы он был виден и все время прорисовывался. Это и делает следующая строка кода:

```
games.screen.add(pizza)
```

Метод add() просто помещает спрайт на графический экран.

ХИТРОСТЬ

Чтобы создавать графику для игр, не надо быть дизайнером. Как вы уже могли заметить, в этой главе полное отсутствие у меня художественных способностей замаскировала технологическая новинка — фотоаппарат. Добавлять картинки в свои проекты может любой владелец цифрового фотоаппарата. По сути, именно этим способом создана вся графика в игре «Паника в пиццерии». Кирпичную стену я снял, находясь в гостях у одного своего знакомого. Чтобы получить изображение пиццы, однажды вечером я заказал себе порцию. Наконец, в роли шеф-повара выступил мой ничуть не стеснительный друг Дэйв.

Это, конечно, прекрасный способ, но помните, что не всякий снимок человека или предмета автоматически переходит в вашу собственность: возможно, заснятый объект защищен авторским правом или относится к торговой марке. Впрочем, если осмотрительно пользоваться цифровой камерой и запечатлевать лишь «обобщенные» виды, ничто не помешает вам украсить программы неповторимым элементом фотореализма.

В табл. 11.5 и 11.6 перечислены свойства и методы, доступные в классе `Sprite`, которые, возможно, пригодятся вам.

Таблица 11.5. Полезные свойства экземпляров класса `Sprite`

Свойство	Описание
<code>angle</code>	Наклон в градусах
<code>x</code>	Абсцисса
<code>y</code>	Ордината
<code>dx</code>	Скорость по оси абсцисс
<code>dy</code>	Скорость по оси ординат
<code>left</code>	Абсцисса левого угла спрайта
<code>right</code>	Абсцисса правого угла спрайта
<code>top</code>	Ордината верхнего угла спрайта
<code>bottom</code>	Ордината нижнего угла спрайта
<code>image</code>	Объект-изображение, на основе которого создан спрайт
<code>overlapping_sprites</code>	Список всех объектов, которые на экране перекрываются с данным спрайтом
<code>is_collideable</code>	Булево свойство, определяющее, будет ли спрайт «сталкиваться», то есть должны ли регистрироваться его перекрытия с другими объектами на экране как столкновения

Таблица 11.6. Полезные методы экземпляров класса `Sprite`

Метод	Описание
<code>update()</code>	Обновляет внешний вид спрайта. Автоматически вызывается при каждой итерации цикла <code>mainloop()</code>
<code>destroy()</code>	Удаляет спрайт с экрана

Отображение текста

Чтобы вывести на экран счет — неважно, означает он количество заработанных долларов или погубленных инопланетян, — вам придется найти способ отобразить на графическом экране текст. Такой способ есть: в модуле `games` для этого создан специализированный класс `Text`.

Знакомство с программой «Ничего себе результат!»

Чтобы отобразить текст на графическом экране, надо всего лишь создать объект класса `Text`. В программе «Ничего себе результат!», которая представляет собой не более чем очередную модификацию «Фоновой картинки», счет выводится на экран, как и во многих классических аркадных играх, в правом верхнем углу. Как это выглядит, показано на рис. 11.9.

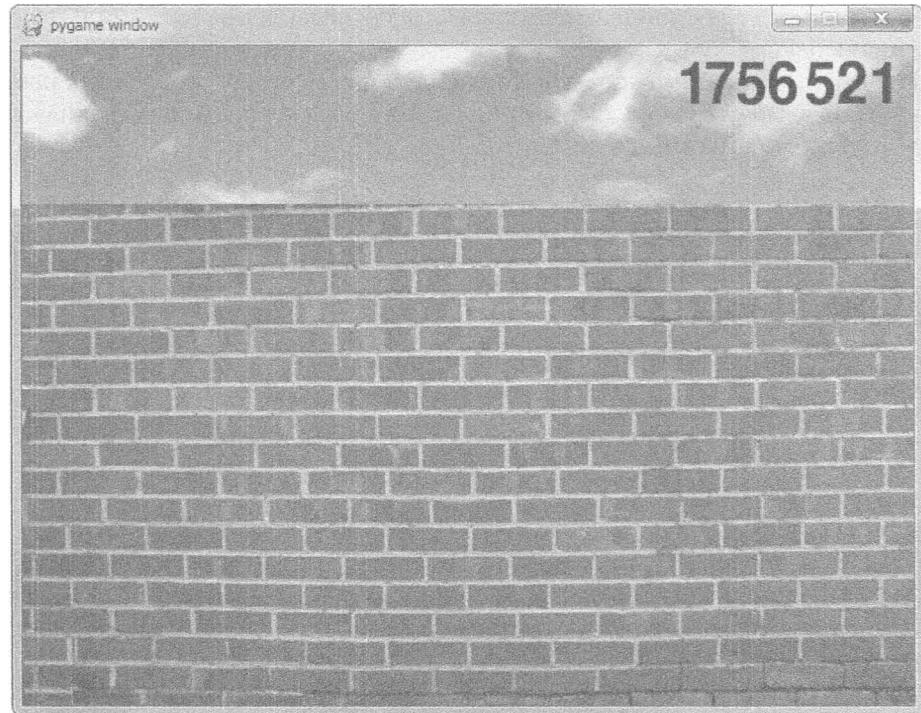


Рис. 11.9. После создания объекта Text на экране отображается впечатляющий счет

Код программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется big_score.py.

```
# Ничего себе результат!
# Демонстрирует отображение текста на графическом экране
from livewires import games, color
games.init(screen_width = 640, screen_height = 480, fps = 50)
wall_image = games.load_image("wall.jpg", transparent = False)
games.screen.background = wall_image
score = games.Text(value = 1756521,
                    size = 60,
                    color = color.black,
                    x = 550,
                    y = 30)
games.screen.add(score)
games.screen.mainloop()
```

Импорт модуля color

В пакете `livewires`, кроме `games`, содержится модуль `color`, в котором задан набор констант, представляющих различные цвета. Этими цветами можно пользоваться в некоторых графических объектах, например экземплярах классов `Text` и `Message`.

(полный список предустановленных цветов перечислен в приложении В, в документации к пакету `livewires`).

Чтобы получить возможность выбрать один из заранее определенных цветов, я импортирую модуль `color`. Для этого надо немного скорректировать директиву `import` в начале программы:

```
from livewires import games, color
```

Теперь из пакета `livewires` загружаются модули `color` и `games`.

Создание объекта Text

Объектом `Text` может быть представлена надпись на графическом экране. Прежде чем вызвать функцию `mainloop()`, я создал объект `Text` и связал его с переменной `score`:

```
score = games.Text(value = 1756521,
                    size = 60,
                    color = color.black,
                    x = 550,
                    y = 30)
```

Методу-конструктору класса `Text` надо передать по крайней мере пять аргументов: значение, которое будет отображаться, размер шрифта, его цвет, абсциссу и ординату надписи.

Параметру `value` я передал целое число 17 56521. Оно отобразится на экране, но как строка (вообще, какое бы значение ни было передано параметру `value`, объект `Text` преобразует его в строку). Далее параметру `size` было передано 60 — это высота текста в пикселях. Шрифт будет крупным и красивым, не хуже самого результата. Параметр `color` получил значение константы `color.black`, извлеченной из модуля `color`. Это значит, что цвет шрифта будет черным. Наконец, переменные `x` и `y` получают значения соответственно 550 и 30, так что центр текстового объекта будет помещен в точку (550; 30) — поблизости от правого верхнего угла графического окна.

Добавление объекта Text на экран

Следующая строка кода добавляет новый объект на графический экран, где он и будет отображаться:

```
games.screen.add(score)
```

При вызове `mainloop()` на мониторе появится графическое окно, в составе которого теперь будет и результат `score`.

Поскольку `Text` — подкласс `Sprite`, объекты этого класса наследуют все свойства, атрибуты и методы спрайтов. Два собственных свойства, объявленных в классе `Text`, приводятся в табл. 11.7.

Таблица 11.7. Добавочные свойства экземпляров класса `Text`

Свойство	Описание
<code>value</code>	Значение, отображаемое как текст
<code>color</code>	Цвет текста

Вывод сообщения

Может случиться, что какой-либо текст вам надо будет вывести на экран лишь на короткое время. Например, иногда пользователя надо извещать о том, что «Список рекордов обновлен», а иногда о том, что «Уровень 7 успешно пройден». Для таких временных сообщений идеально подходит класс `Message`.

Знакомство с программой «Победа»

Эта программа — также вариация на тему «Фоновой картинки». Экземпляр класса `Message` создается непосредственно перед запуском `mainloop()`; на экране будет большими красными буквами написано: `Победа!`. Примерно пять секунд это сообщение отображается, а затем исчезает, и программа прекращает работу. Результат показан на рис. 11.10.

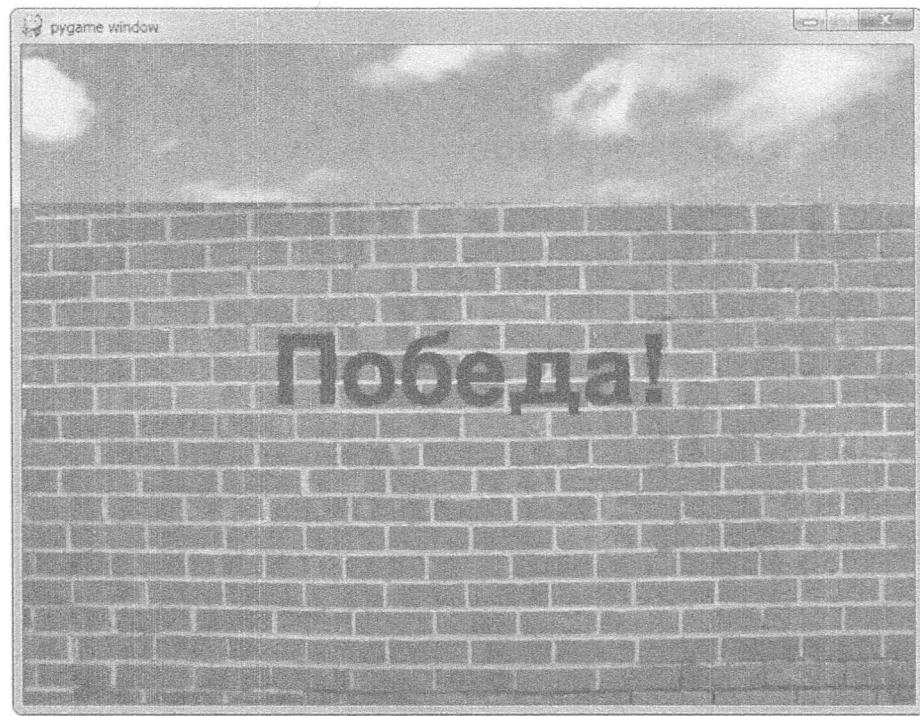


Рис. 11.10. Как сладок миг победы!

Код программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке **Chapter 11**. Файл называется `you_won.py`.

```
# Победа
# Демонстрирует вывод сообщения на экран
from livewires import games, color
```

```

games.init(screen_width = 640, screen_height = 480, fps = 50)
wall_image = games.load_image("wall.jpg", transparent = False)
games.screen.background = wall_image
won_message = games.Message(value = "Победа!",
                             size = 100,
                             color = color.red,
                             x = games.screen.width/2,
                             y = games.screen.height/2,
                             lifetime = 250,
                             after_death = games.screen.quit)
games.screen.add(won_message)
games.screen.mainloop()

```

Импорт модуля color

Экземпляры класса Message, как и объекты Text, располагают свойством color. Чтобы можно было выбрать из множества доступных цветов, я уже знакомым образом импортировал в добавок к games модуль color:

```
from livewires import games, color
```

Создание объекта Message

Сообщения создаются как экземпляры класса Message, входящего в модуль games. Сообщение – это особый подтип объекта Text: текст, самопроизвольно исчезающий с экрана по истечении фиксированного промежутка времени. Можно даже назначить метод или функцию, которая будет вызвана сразу после разрушения объекта.

Метод-конструктор Message принимает все те же значения, что и в случае с Text, но допускает еще два: lifetime и after_death. Параметр lifetime принимает целое число, означающее количество циклов mainloop(), в течение которых сообщение будет висеть на экране. Параметру after_death можно передать имя метода или функции, вызываемой после того, как объект Message исчезнет с экрана. По умолчанию after_death имеет значение None, так что этому параметру не обязательно что-либо передавать.

Объект Message я создал прямо перед вызовом mainloop():

```
won_message = games.Message(value = "Победа!",
                             size = 100,
                             color = color.red,
                             x = games.screen.width/2,
                             y = games.screen.height/2,
                             lifetime = 250,
                             after_death = games.screen.quit)
```

Как результат исполнения этого кода, в центре графического окна будет примерно пять секунд пребывать набранное крупным красным шрифтом слово Победа!, а потом программа закончит работу. Данный код создает новый объект Message с атрибутом lifetime, равным 250, поэтому «время жизни» объекта на экране и составляет с 5 секундами, ведь каждую секунду mainloop() запускается 50-кратно.

По истечении назначенного времени будет вызван метод `games.screen.quit()`, имя которого передано параметру `after_death`. После этого графический экран и все связанные с ним объекты прекратят существование и программа завершится.

ПОДСКАЗКА

Параметру `after_death` следует передавать только имя функции или метода, без следующих за ним скобок.

Ширина и высота графического экрана

У объекта `screen` есть свойство `width`, представляющее ширину графического окна, и свойство `height`, указывающее его высоту. Иногда при размещении объектов на экране удобнее отталкиваться от этих свойств, чем пользоваться буквальными целочисленными координатами.

Так, например, передавая новому объекту `Message` его координаты, я установил `x` равной `games.screen.width/2` и `y` равной `games.screen.height/2`. Приравняв абсциссу к половине ширины графического экрана, а ординату — к половине его высоты, я разместил объект точно по центру окна. Пользуйтесь этим приемом, чтобы помещать объект в середину графического окна независимо от его размеров в пикселях.

Добавление объекта `Message` на экран

Следующая строка кода добавляет новый объект на графический экран для его дальнейшего отображения там:

```
games.screen.add(won_message)
```

Поскольку `Message` — подкласс `Text`, объекты этого класса наследуют все свойства, атрибуты и методы текстовых объектов. Два собственных атрибута, объявленных в классе `Message`, приведены в табл. 11.8.

Таблица 11.8. Добавочные атрибуты экземпляров класса `Message`

Атрибут	Описание
<code>lifetime</code>	Указывает, сколько раз должен быть выполнен цикл <code>mainloop()</code> , прежде чем объект самопроизвольно разрушится. При значении 0 объект вообще не разрушается. По умолчанию установлено значение 0
<code>after_death</code>	Функция или метод, который будет вызван после исчезновения объекта. По умолчанию установлено значение <code>None</code>

Подвижные спрайты

В движущихся картинках вся суть игр, да и вообще большинства развлечений. Спрайты позволяют с легкостью перейти от неподвижного к подвижному, ведь у объектов класса `Sprite` есть специальные свойства, позволяющие им передвигаться по графическому экрану.

Знакомство с программой «Летающая пицца»

Эта новая программа — разновидность «Спрайта-пиццы». В ней изображение пиццы передвигается вправо и вниз по графическому экрану, для чего достаточно изменить несколько строк кода — такова мощь спрайтов! Работа программы показана на рис. 11.11.

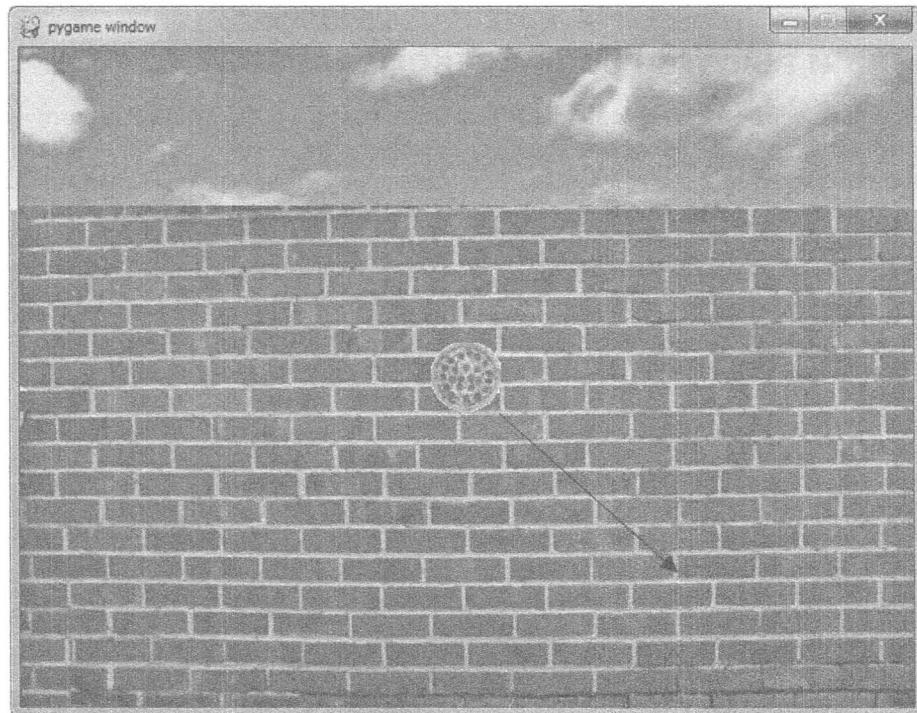


Рис. 11.11. Пицца летит вправо и вниз, в направлении, указанном стрелкой

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется `moving_pizza.py`.

```
# Летающая пицца
# Демонстрирует движущиеся спрайты
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
wall_image = games.load_image("wall.jpg", transparent = False)
games.screen.background = wall_image
pizza_image = games.load_image("pizza.bmp")
the_pizza = games.Sprite(image = pizza_image,
                         x = games.screen.width/2,
                         y = games.screen.height/2,
                         dx = 1,
                         dy = 1)
games.screen.add(the_pizza)
games.screen.mainloop()
```

Настройка скорости движения спрайта

Как оказалось, все, что надо сделать, — изменить код, отвечающий за создание нового спрайта. Методу-конструктору надо добавочно передать значения параметров dx и dy:

```
the_pizza = games.Sprite(image = pizza_image,
                           x = games.screen.width/2,
                           y = games.screen.height/2,
                           dx = 1,
                           dy = 1)
```

У каждого объекта класса `Sprite` или его производных есть свойства `dx` и `dy`, обозначающие горизонтальную и вертикальную скорость объекта соответственно (`d` — сокращение от «дельта», то есть изменения). Итак, `dx` — изменение абсциссы объекта за единицу времени, в качестве которой выступает один цикл `mainloop()`, а `dy` — изменение ординаты объекта за ту же единицу времени. Положительное значение `dx` передвигает спрайт вправо, а отрицательное — влево; положительное значение `dy` передвигает спрайт вниз, а отрицательное — вверх.

В программе «Спрайт-пицца» никаких значений параметрам `dx` и `dy` не передавалось. Хотя у спрайта в этой программе точно так же были свойства `dx` и `dy`, их значение по умолчанию установлено равным 0. А вот теперь, когда я передал параметрам `dx` и `dy` по 1, при каждом очередном обновлении графического окна, то есть при каждом запуске `mainloop()`, картинка с пиццей будет сдвигаться на 1 пиксель по горизонтали (вправо) и на 1 пиксель по вертикали (вниз).

Учет границ экрана

Наблюдая за «Летающей пиццей» хоть сколько-нибудь долго, вы могли заметить, что соприкосновение с границей графического экрана не заставляет спрайт остановиться. Пицца продолжает лететь, как будто ничего не произошло, и исчезает из виду.

Если вы запустили движение спрайта, то вам следует создать также механизм обработки столкновений спрайта с границами экрана. Альтернатив несколько. Движущийся спрайт может, во-первых, просто останавливаться по достижении края окна. Во-вторых, он может исчезать (например, с эффектным взрывом). В-третьих, может упруго отскакивать, как мячик. Наконец, в-четвертых, спрайт может огибать экран так, что, исчезнув с одной стороны, он появится на противоположной. Для пиццы, пожалуй, нет лучше варианта, чем отскакивание.

Программа «Скачущая пицца»

Говоря, что спрайт «отскакивает» от границ графического окна, я имею в виду, что, когда он достигает границы окна, та компонента его скорости, которая привела к столкновению, должна быть обращена. Если, например, скачущая пицца достигает верхней или нижней кромки графического экрана, свойство `dy` должно поменять знак, оставшись прежним по абсолютной величине. Если же спрайт соприкоснется

с боковой стороной экрана, обращена должна быть скорость dx. На рис. 11.12 проиллюстрирована работа программы «Скачущая пицца».

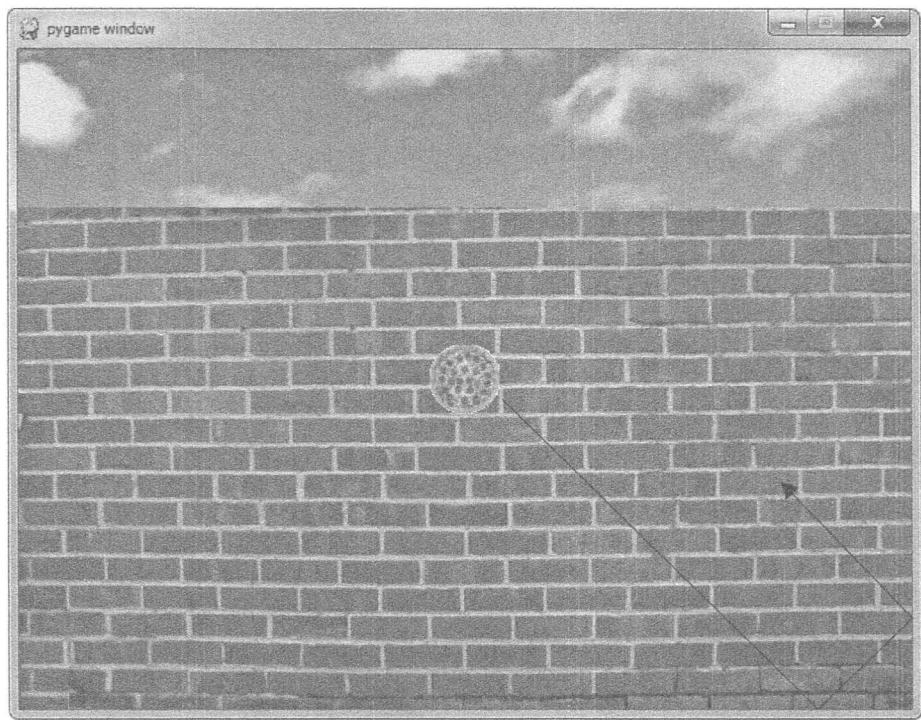


Рис. 11.12. Пицца движется точно по стрелке, хотя этого и нельзя увидеть на рисунке

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется `bouncing_pizza.py`.

Настройка программы

Начало кода — стандартное для программы с графическим окном:

```
# Скачущая пицца
# Демонстрирует обработку столкновений с границами экрана
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Как и ранее, таким образом я получаю доступ к модулю `games` и создаю графический экран.

Создание подкласса `Sprite`

Впервые мне понадобилось «научить» спрайт делать что-то, чего он изначально не умеет: отскакивать. Для этого произведу от `Sprite` дочерний класс. Посколь-

ку его экземплярами будут круги пиццы, этот новый класс я незатейливо назову `Pizza`:

```
class Pizza(games.Sprite):
    """ Скачущая пицца. """
```

Переопределение метода `update()`

Чтобы летящая пицца превратилась в скачущую, в классе `Pizza` надо реализовать лишь один собственный метод. Каждый раз при обновлении графического окна, то есть при запуске `mainloop()`, происходят две вещи:

- меняется положение каждого спрайта исходя из величин `dx` и `dy`;
- вызывается метод `update()` каждого спрайта.

У каждого экземпляра класса `Sprite` есть метод `update()`, но по умолчанию он не делает ничего. Поэтому переопределение `update()` в классе `Pizza` — самый удобный способ обработки столкновений с границами экрана.

```
def update(self):
    """ Обращает одну или обе компоненты скорости, если достигнута граница экрана. """
    if self.right > games.screen.width or self.left < 0:
        self.dx = -self.dx
    if self.bottom > games.screen.height or self.top < 0:
        self.dy = -self.dy
```

Этот метод проверяет, не собирается ли спрайт в следующем цикле `mainloop()` выйти за границы экрана в одном из направлений. Если собирается, то соответствующая величина скорости обращается.

Если свойство объекта `right`, представляющее абсциссу его правого края, превышает `games.screen.width`, то пицца вот-вот вылетит за пределы графического окна вправо. Если свойство `left`, представляющее абсциссу левого края спрайта, оказывается меньше 0, то пицце грозит вылет за левую кромку экрана. В обоих случаях достаточно присвоить противоположный знак горизонтальной скорости `dx`, чтобы пицца «отскочила» от границы окна.

Если свойство объекта `bottom`, представляющее ординату его нижнего края, превышает `games.screen.height`, то пицца начинает свое падение вниз — в никуда. Если же свойство `top`, представляющее ординату верхнего края спрайта, оказывается меньше 0, то пицца собирается вылететь вверх за пределы экрана. В обоих случаях я обращаю `dy` — вертикальную скорость пиццы.

Обертка программы

Поскольку в этой программе я объявил класс, не помешает организовать весь остальной ее код в виде функций:

```
def main():
    wall_image = games.load_image("wall.jpg", transparent = False)
    games.screen.background = wall_image
```

```

pizza_image = games.load_image("pizza.bmp")
the_pizza = Pizza(image = pizza_image,
                   x = games.screen.width/2,
                   y = games.screen.height/2,
                   dx = 1,
                   dy = 1)
games.screen.add(the_pizza)
games.screen.mainloop()
# поехали!
main()1

```

По большей части этот код вам уже знаком. Важное отличие лишь одно: я создал объект уже нового, собственного класса `Pizza`, а не обобщенного `Sprite`. Поэтому метод `update()`, фиксируя соударения с границами окна, обращает скорости, и пицца «скакает».

Обработка ввода с помощью мыши

Хотя вам известны уже многие функции пакета `livewires`, мы только-только еще добираемся до самого интересного: интерактивности и пользовательского ввода. Типичный механизм пользовательского ввода в играх реализуется с использованием мыши. Соответствующий объект в `livewires` позволяет организовать взаимодействие.

Знакомство с программой «Подвижная сковорода»

Объект `mouse` модуля `games` обеспечивает доступ к мыши. Из свойств этого объекта удивительно легко прочесть положение мыши на графическом экране. Пользуясь данными свойствами, я создал программу «Подвижная сковорода», в которой пользователь, перемещая мышь, меняет положение спрайта с изображением сковородки. Работа программы иллюстрируется на рис. 11.13.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется `moving_pan.py`.

Настройка программы

Следующие строки кода вам, конечно, уже знакомы:

```

# Подвижная сковорода
# Демонстрирует ввод с помощью мыши
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)

```

¹ Лучше запуск `main()` прятать внутрь такой конструкции:
`if __name__ == '__main__':
 main()`

Тогда этот сценарий можно будет использовать как модуль, и в процессе импортирования данного модуля не будет выполняться никакого кода. В противном случае `main()` выполнится, даже если кто-то захочет сделать `import` вашего модуля. — Примеч. науч. ред.

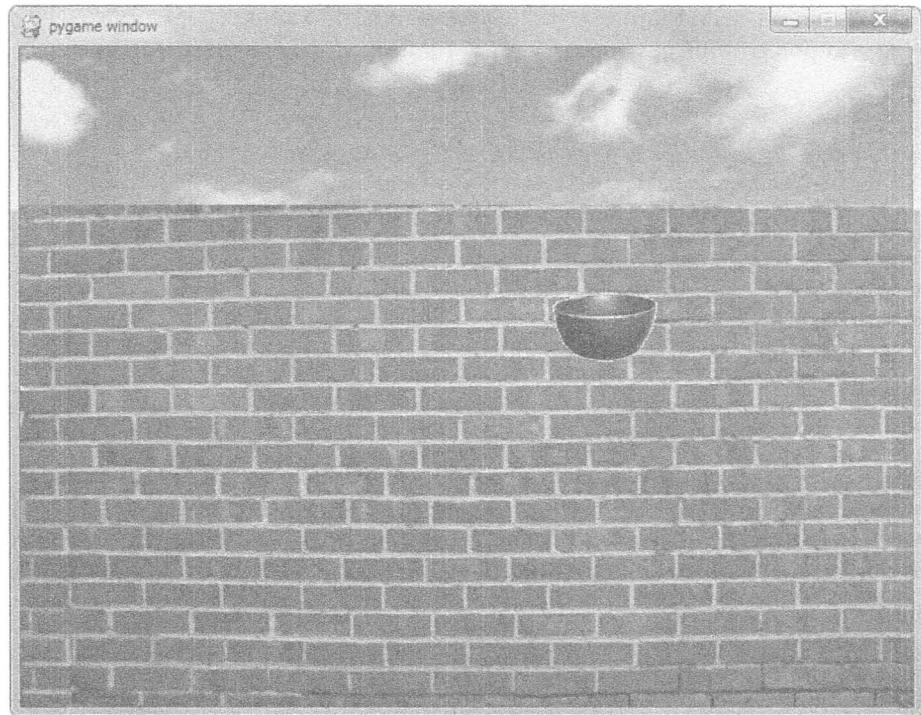


Рис. 11.13. На графическом экране сковорода следует за указателем мыши

Как всегда, импортируется модуль `games` и инициализируется графический экран. Метод `init()` также автоматически создает объект `mouse`, которым мы далее воспользуемся для чтения координат указателя.

Чтение координат указателя

Теперь создам класс `Pan` для спрайта-сковородки:

```
class Pan(games.Sprite):
    """ Перемещаемая мышью сковорода. """
    def update(self):
        """ Перемещает объект в позицию указателя. """
        self.x = games.mouse.x
        self.y = games.mouse.y
```

Как и у экземпляров `Sprite`, у объекта `mouse` есть свойство `x`, описывающее положение указателя по горизонтали, и свойство `y` — для положения указателя по вертикали. Из них можно узнать текущую позицию указателя мыши на графическом экране. В методе `update()` объекту `Pan` приписывается такая же абсцисса `x`, как у указателя (свойство `x` объекта `mouse`), и такая же ордината `y`, как у указателя (свойство `y` объекта `mouse`). Таким образом, сковорода передвигается в позицию указателя мыши.

Теперь я создам функцию `main()` с кодом, общий характер которого вам должен быть хорошо понятен. Здесь будет установлена фоновая картинка и созданы спрайты:

```
def main():
    wall_image = games.load_image("wall.jpg", transparent = False)
    games.screen.background = wall_image
    pan_image = games.load_image("pan.bmp")
    the_pan = Pan(image = pan_image,
                   x = games.mouse.x,
                   y = games.mouse.y)
    games.screen.add(the_pan)
```

Поскольку параметру `x` передается значение `games.mouse.x`, а параметру `y` — `games.mouse.y`, объект `Pan` появится на экране в точке, где расположен указатель.

Настройка видимости указателя

Далее в функции `main()` я пользуюсь свойством `is_visible` объекта `mouse`, чтобы сделать указатель мыши невидимым.

```
games.mouse.is_visible = False
```

Если установить значение этого свойства равным `True`, то указатель мыши появится на экране, а поскольку оно равно `False`, указатель будет невидим: нецелесообразно, чтобы поверх изображения сковороды все время висела стрелка.

Перенаправление ввода в графическое окно

Далее в функции `main()` я пользуюсь свойством `event_grab` объекта `screen`, чтобы перенаправить (или перехватить) весь пользовательский ввод в графическое окно:

```
games.screen.event_grab = True
```

Если сделать это свойство равным `True`, то пользователь не сможет вывести указатель мыши за пределы графического окна, что в нашем случае очень удобно. Если присвоить `event_grab` значение `False`, то пользовательский ввод не будет сосредоточен в графическом окне: указатель сможет его покидать.

ПОДСКАЗКА

Если весь ввод перехватывается графическим экраном, закрыть окно с помощью мыши пользователю не удастся, но это всегда можно сделать нажатием клавиши `Esc`.

Обертка программы

Вслед за этим я, как обычно, заканчиваю `main()` вызовом `mainloop()`, после которого все на экране «оживает», и запускаю `main()`.

```
games.screen.mainloop()
# поехали!
main()
```

Некоторые полезные для программиста свойства объекта mouse перечислены в табл. 11.9.

Таблица 11.9. Полезные свойства объекта mouse

Свойство	Описание
x	Абсцисса указателя мыши
y	Ордината указателя мыши
is_visible	Булев параметр, определяющий, будет ли видимым указатель мыши: True — да, False — нет. По умолчанию — True

Регистрация столкновений

В большинстве игр при столкновении двух предметов результат сразу налицо. В простейшем случае двухмерный персонаж может налетать на препятствие, которое не пустит его пройти дальше; в более сложно реализованной и зрелищной трехмерной игре астероид может пропарывать обшивку космического корабля и т. п. Так или иначе, нужен способ отслеживать столкновения.

Знакомство с программой «Ускользающая пицца»

Эта программа — вариант «Подвижной сковороды». Здесь пользователь, как и ранее, мышью перетаскивает по графическому экрану сковороду, но на экране этот спрайт не единственный: есть еще изображение пиццы. Как только пользователь приблизит сковороду к пицце вплоть до соприкосновения, ускользающая еда вдруг переместится в случайно выбранную точку на экране. Работу программы иллюстрируют рис. 11.14 и 11.15.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется slippery_pizza.py.

Настройка программы

Начальный код взят из «Подвижной сковороды» и содержит только одно маленькое дополнение:

```
# Ускользающая пицца
# Демонстрирует проверку на соприкосновение спрайтов
from livewires import games
import random
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

В новинку здесь только импорт модуля random — нашего доброго знакомого, который после соприкосновения двух спрайтов поможет подыскать новые, случайные координаты для пиццы.

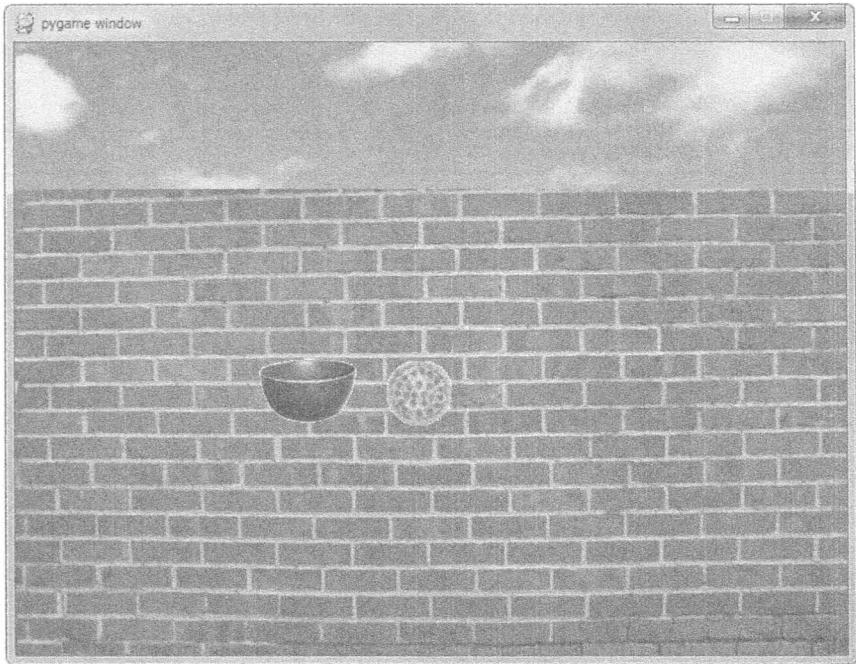


Рис. 11.14. Кажется, вот-вот игрок достигнет цели...

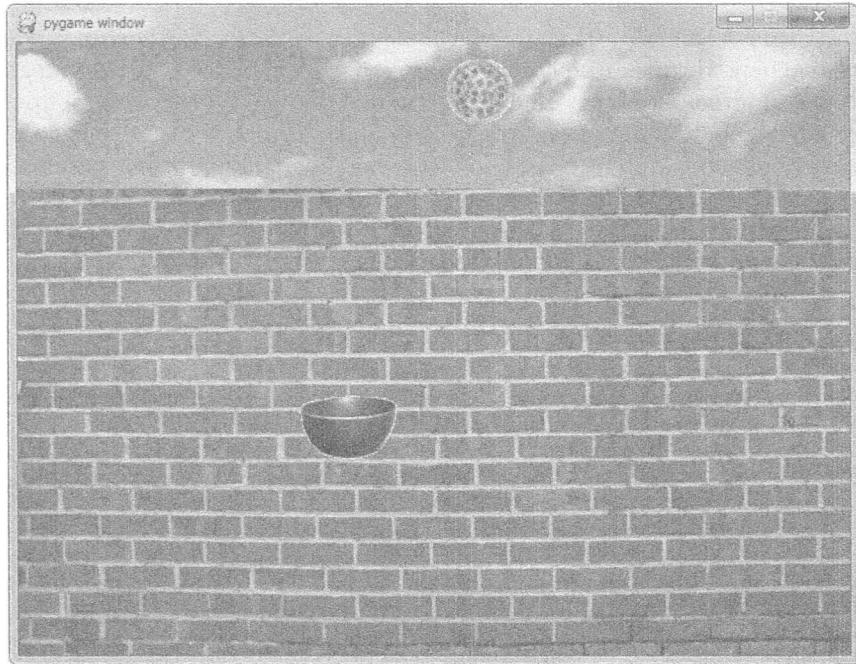


Рис. 11.15. ...но ускользающая пицца вновь и вновь не дается в руки

Регистрация столкновений

Создадим новый класс Pan, в котором часть кода будет посвящена регистрации столкновений:

```
class Pan(games.Sprite):
    """ Сковорода, которую можно мышью перемещать по экрану. """
    def update(self):
        """ Перемещает спрайт в позицию, в которой находится указатель. """
        self.x = games.mouse.x
        self.y = games.mouse.y
        self.check_collide()
    def check_collide(self):
        """ Проверяет, не соприкоснулись ли сковорода и пицца. """
        for pizza in self.overlapping_sprites:
            pizza.handle_collide()
```

В последней строке кода метода update() вызывается метод check_collide(), который, в свою очередь, перебирает список спрайтов, хранимый в свойстве overlapping_sprites объекта Pan. Это спрайты, имеющие визуальное перекрытие с данным экземпляром Pan на экране. По отношению к каждому из них вызывается метод handle_collide(). Попросту говоря, сковорода «велит» соприкасающейся с ней пицце обработать столкновение.

Обработка столкновений

Теперь создадим новый класс Pizza:

```
class Pizza(games.Sprite):
    """ Ускользающая пицца. """
    def handle_collide(self):
        """ Перемещает спрайт в случайную позицию на графическом экране. """
        self.x = random.randrange(games.screen.width)
        self.y = random.randrange(games.screen.height)
```

Я реализовал только один метод handle_collide(), который генерирует случайные координаты и присваивает их объекту Pizza.

Обертка программы

Вот функция main():

```
def main():
    wall_image = games.load_image("wall.jpg", transparent = False)
    games.screen.background = wall_image
    pizza_image = games.load_image("pizza.bmp")
    pizza_x = random.randrange(games.screen.width)
    pizza_y = random.randrange(games.screen.height)
    the_pizza = Pizza(image = pizza_image, x = pizza_x, y = pizza_y)
    games.screen.add(the_pizza)
    pan_image = games.load_image("pan.bmp")
```

```

the_pan = Pan(image = pan_image,
               x = games.mouse.x,
               y = games.mouse.y)
games.screen.add(the_pan)
games.mouse.is_visible = False
games.screen.event_grab = True
games.screen.mainloop()
# поехали!
main()

```

В начале, как всегда, создается фоновая картинка. Вслед за тем появляются два объекта, экземпляры классов `Pizza` и `Pan`. Для пиццы начальные координаты выбираются случайно, а сковорода помещается в позицию указателя. Указатель мыши я сделал невидимым; весь пользовательский ввод перенаправляется в графическое окно. Последним на очереди в функции `main()` стоит вызов `mainloop()`. Программа начинает работу по вызову `main()`.

Вернемся к игре «Паника в пиццерии»

Теперь, владея возможностями мультимедийного пакета `livewires`, вы можете поучаствовать в разработке игры «Паника в пиццерии», общее представление о которой получили в начале этой главы. Значительная часть кода игры позаимствована прямо из демонстрационных программ, однако будет введено также несколько новых понятий. Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 11. Файл называется `pizza_panic.py`.

Настройка программы

Как и во всех программах этой главы, начнем с импорта модулей и инициализации графического экрана:

```

# Паника в пиццерии
# Игрок должен ловить падающую пиццу, пока она не достигла земли
from livewires import games, color
import random
games.init(screen_width = 640, screen_height = 480, fps = 50)

```

Модуль `games` отвечает за всю графику, а `color` избавляет от труда самостоятельно задавать цвета. Импортировать `random` надо затем, чтобы поведение безумного шеф-повара выглядело как можно более реалистичным. Наконец, для инициализации графического экрана и обеспечения доступа к мыши вызывается метод `init()` модуля `games`.

Класс `Pan`

Класс `Pan` — своего рода «чертеж» спрайта с изображением сковородки, которым игрок будет управлять с помощью мыши. Важно, что здесь перемещения могут

происходить только по горизонтали: влево и вправо. Рассмотрим код этого класса отдельными частями.

Загрузка картинки со сковородой

В самом начале класса происходит нечто необычное: изображение для спрайта загружается и связывается с переменной класса — `image`. Это сделано потому, что в игре «Паника в пиццерии» несколько классов и загрузить соответствующую картинку внутри объявления класса — намного изящнее, чем оставить загрузку всех изображений на совести функции `main()`.

```
class Pan(games.Sprite):
    """
    Сковорода, в которую игрок может ловить падающую пиццу.
    """
    image = games.load_image("pan.bmp")
```

Метод `__init__()`

Теперь я напишу метод-конструктор, который будет инициализировать новые экземпляры класса `Pan`:

```
def __init__(self):
    """
    Инициализирует объект Pan и создает объект Text для отображения счета.
    """
    super(Pan, self).__init__(image = Pan.image,
                             x = games.mouse.x,
                             bottom = games.screen.height)
    self.score = games.Text(value = 0, size = 25, color = color.black,
                           top = 5, right = games.screen.width - 10)
    games.screen.add(self.score)
```

Функция `super()` применена для того, чтобы вызвать метод-конструктор `init()` класса `Sprite`. Вслед за тем я объявил атрибут `score` — текстовый объект с начальным значением 0, предназначенный для отображения текущего счета. Конечно, этот новый объект я не забываю добавить на экран.

Метод `update()`

Этот метод нужен для передвижения сковородки:

```
def update(self):
    """
    Передвигает объект по горизонтали в точку с абсциссой, как у указателя мыши.
    """
    self.x = games.mouse.x
    if self.left < 0:
        self.left = 0
    if self.right > games.screen.width:
        self.right = games.screen.width
    self.check_catch()
```

Координата `x` объекта `Pan` становится равной координате `x` указателя, так что игрок может мышью двигать сковороду влево-вправо. Здесь же, взяв свойство `left`

объекта Pan, я проверяю, не принял ли оно значение меньше 0, то есть не вышла ли часть картинки за левый край «графического окна». Если это произошло, свойству left присваивается значение 0 и сковорода возвращается на левый край окна. Далее, взяв свойство right, я проверяю, не принял ли оно значение, большее чем games.screen.width, то есть не вышла ли часть картинки за правый край окна. Если это произошло, то свойству right присваивается значение games.screen.width и сковорода возвращается на правый край окна.

В конце вызывается метод check_catch() объекта. Он проверяет, поймал ли игрок один из падающих кругов пиццы:

```
def check_catch(self):
    """ Проверяет, поймал ли игрок падающую пиццу. """
    for pizza in self.overlapping_sprites:
        self.score.value += 10
        self.score.right = games.screen.width - 10
        pizza.handle_caught()
```

Каждый объект, имеющий визуальное перекрытие со сковородой, увеличивает счет игрока на 10 очков. После этого метод размещает правый край объекта Text ровно в 10 пикселях от правого края графического окна — вне зависимости от того, сколько цифр в счете. Последним идет вызов метода handle_caught() того спрайта, который перекрывается с данным.

Класс Pizza

Экземпляры этого класса представляют собой падающую пиццу, которую игрок должен ловить:

```
class Pizza(games.Sprite):
    """
    Круги пиццы, падающие на землю.

    """
    image = games.load_image("pizza.bmp")
    speed = 1
```

В классе объявлены две переменные: image — для картинки с пиццией и speed — для скорости падения. Последней присваивается значение 1, так что падать пицца будет чрезвычайно медленно. Обе переменные класса использованы в методе-конструкторе объектов Pizza, с которым мы сейчас и ознакомимся.

Метод __init__()

Этот метод инициализирует новый экземпляр класса Pizza:

```
def __init__(self, x, y = 90):
    """ Инициализирует объект Pizza. """
    super(Pizza, self).__init__(image = Pizza.image,
                               x = x, y = y,
                               dy = Pizza.speed)
```

В этом методе я всего лишь вызвал конструктор класса, родительского по отношению к Pizza. Заметьте, что значение у по умолчанию я сделал равным 90, чтобы

каждая следующая пицца возникала вверху как раз на одном уровне с руками обезумевшего кулинара.

Метод update()

Этот метод обрабатывает соприкосновения с границами экрана:

```
def update(self):
    """ Проверяет, не коснулась ли нижняя кромка спрайта нижней границы экрана. """
    if self.bottom > games.screen.height:
        self.end_game()
        self.destroy()
```

Задача этого метода — проверять, не достигла ли пицца условной земли. Если достигла, то будет вызван метод `end_game()`, а сам объект исчезнет с экрана.

Метод handle_caught()

Запомните, что этот метод может быть вызван только объектом `Pan` при соударении с ним экземпляра `Pizza`:

```
def handle_caught(self):
    """ Разрушает объект, пойманный игроком. """
    self.destroy()
```

Когда пицца сталкивается со сковородой, она, как считается, «поймана» и просто перестает существовать. К объекту `Pizza` применяется метод `destroy()` — и тот буквально исчезает.

Метод end_game()

Этот метод заканчивает игру. Он будет вызван после того, как хотя бы одна пицца достигнет нижней кромки окна.

```
def end_game(self):
    """ Завершает игру. """
    end_message = games.Message(value = "Game Over",
                                 size = 90,
                                 color = color.red,
                                 x = games.screen.width/2,
                                 y = games.screen.height/2,
                                 lifetime = 5 * games.screen.fps,
                                 after_death = games.screen.quit)
    games.screen.add(end_message)
```

Этот код создает объект `Message`, извещающий о конце игры. Примерно через 5 секунд сообщение исчезнет, а графическое окно закроется.

ЛОВУШКА

Метод `end_game()` вызывается каждый раз, когда пицца достигает «земли». Но, поскольку сообщение «висит» около 5 секунд, не исключено, что за это время еще одна пицца «упадет», а значит, пользователь будет вторично уведомлен о конце игры. В главе 12 вы узнаете, как создать объект, который будет отслеживать, продолжается текущая игра или уже окончена, и предупреждать разные нежелательные эффекты типа множественных уведомлений.

Класс Chef

Экземпляр класса Chef — это шеф-повар, который, будучи не в себе, взобрался на крышу ресторана и швыряет оттуда пиццу.

```
class Chef(games.Sprite):
    """
    Кулинар, который, двигаясь влево-вправо, разбрасывает пиццу.
    """
    image = games.load_image("chef.bmp")
```

Атрибут класса `image` ссылается на картинку — изображение повара.

Метод `__init__()`

Вот метод-конструктор:

```
def __init__(self, y = 55, speed = 2, odds_change = 200):
    """
    Инициализирует объект Chef.
    """
    super(Chef, self).__init__(image = Chef.image,
        x = games.screen.width / 2,
        y = y,
        dx = speed)
    self.odds_change = odds_change
    self.time_til_drop = 0
```

Здесь сначала вызывается конструктор родительского класса. Параметру `image` передается значение атрибута класса `Chef.image`. Параметр `x` принимает значение, соответствующее середине экрана, а `y` — значение по умолчанию 55, которое располагает фигурку повара как раз над кирпичной стеной. Горизонтальная скорость `dx` движения повара вдоль края крыши оказывается по умолчанию равной 2.

Метод создает также два атрибута у объекта: `odds_change` и `time_til_drop`. Первый из них — `odds_change` — это целое число, обратное вероятности того, что в данный момент повар изменит направление движения. Если, например, параметр `odds_change` равен 200, то существует один шанс из 200, что буквально в данный момент вектор горизонтальной скорости поменяет знак. Как работает этот механизм, вы увидите при разборе метода `update()`.

Целочисленный атрибут `time_til_drop` — это количество циклов `mainloop()`, после которых повар должен сбросить с крыши очередную пиццу. При инициализации этот атрибут равен 0; таким образом, при появлении на экране объект `Chef` сразу же сбрасывает один кружок пиццы. Как работает этот механизм, будет показано при разборе метода `check_drop()`.

Метод `update()`

В этом методе описываются правила, согласно которым фигурка повара перемещается туда-сюда по крыше:

```
def update(self):
    """
    Определяет, надо ли сменить направление.
    """
    if self.left < 0 or self.right > games.screen.width:
        self.dx = -self.dx
```

```
elif random.randrange(self.odds_change) == 0:
    self.dx = -self.dx
    self.check_drop()
```

Повар движется вдоль края крыши в одну сторону до тех пор, пока не достигнет кромки окна или не «решит» (случайным образом) поменять направление. В начале этого кода выполняется проверка на соприкосновение с одной из боковых сторон графического окна. Если край достигнут, то вектор горизонтальной скорости повара обращается: `self.dx = -self.dx`. В противном случае есть один из `odds_change`-шансов на то, что направление движения будет изменено.

Вне зависимости от того, был ли обращен вектор скорости, последнее на очереди — вызвать метод `check_drop()` объекта `Chef`.

Метод `check_drop()`

Этот метод вызывается при каждой итерации цикла `mainloop()`, что, впрочем, отнюдь не значит, что на игрока будет падать по одной пицце 50 раз в секунду:

```
def check_drop(self):
    """ Уменьшает интервал ожидания на единицу или сбрасывает очередную пиццу
    и восстанавливает исходный интервал. """
    if self.time_til_drop > 0:
        self.time_til_drop -= 1
    else:
        new_pizza = Pizza(x = self.x)
        games.screen.add(new_pizza)
    # вне зависимости от скорости падения пиццы "зазор" между падающими кругами
    # принимается равным 30% каждого из них по высоте
    self.time_til_drop = int(new_pizza.height * 1.3 / Pizza.speed) + 1
```

Параметр `time_til_drop`, как вы помните, — это интервал обратного отсчета, в течение которого повар не предпринимает никаких действий. Здесь, если окажется, что этот интервал больше 0, из него вычитается 1; в противном случае появится новый объект `Pizza`, а значение интервала `time_til_drop` будет восстановлено. У вновь созданной пиццы та же абсцисса, что и у объекта `Chef` (конструктору передается соответствующее значение параметра `x`), а значит, этот спрайт начнет двигаться оттуда же, где в данный момент находится спрайт с изображением повара. Значение `time_til_drop` восстанавливается таким образом, чтобы следующая порция пиццы была сброшена при удалении предыдущей порции от верхней кромки экрана на 30% высоты картинки. Скорость падения пиццы не принимается в расчет.

Функция `main()`

Функция `main()` создает объекты и начинает игру:

```
def main():
    """ Собственно игровой процесс. """
    wall_image = games.load_image("wall.jpg", transparent = False)
    games.screen.background = wall_image
    the_chef = Chef()
    games.screen.add(the_chef)
```

```

the_pan = Pan()
games.screen.add(the_pan)
games.mouse.is_visible = False
games.screen.event_grab = True
games.screen.mainloop()
# начнем!
main()

```

Сначала здесь в качестве фона выбирается изображение кирпичной стены. Затем создаются шеф-повар и сковородка. После этого указатель делается невидимым, а весь пользовательский ввод перенаправляется в графическое окно, так что указатель мыши не сможет покинуть его. Чтобы отобразить окно и начать игру, вызывается `mainloop()`. Вызов функции `main()` запускает весь процесс.

Резюме

Из этой главы вы узнали, как с помощью мультимедийного пакета `livewires` украсить свои программы графикой. Вы научились создавать пустое графическое окно и выбирать фоновый рисунок для него. Кроме того, вы увидели, как можно отобразить текст внутри графического окна. Вы познакомились со спрайтами — особыми графическими объектами на основе картинок — и научились их позиционировать и перемещать на графическом экране. Вы теперь знаете, как регистрировать и обрабатывать «столкновения» спрайтов, как получать пользовательский ввод мышью. В конце главы вы применили полученные знания на практике, создав увлекательную видеоигру, противник в которой полностью управляет компьютером.

ЗАДАЧИ

- Доработайте игру «Паника в пиццерии» так, чтобы сложность игрового процесса постепенно возрастила. Задумайтесь о разных способах добиться следующих эффектов: увеличить скорость падения пицы и/или перемещения повара, уменьшить расстояние от крыши до сковороды, наконец, выпустить на экран нескольких сумасшедших кулинаров.
- Напишите игру, в которой на персонаж, управляемый игроком с помощью мыши, сверху (с «неба») будут падать какие-нибудь тяжелые объекты, а он должен будет уворачиваться.
- Создайте простую игру в пинг-понг для одного игрока. В этой игре пользователь должен манипулировать ракеткой, а шарик — отскакивать от трех стенок. Если шарик проскочит мимо ракетки и вылетит за пределы игрового поля, игра должна заканчиваться.

12

Звук, анимация, разработка больших программ. Игра «Прерванный полет»

В этой главе вы овладеете дополнительными навыками мультимедийного программирования и узнаете, как работать со звуком и анимированными картинками. Кроме того, вы обучитесь технике постепенной разработки большого проекта. Вам предстоит освоить, в частности, следующие умения:

- читать клавиатурный ввод в графической игре;
- воспроизводить звук в программе;
- проигрывать музыкальные файлы;
- создавать анимацию;
- разрабатывать программу, последовательно создавая все более и более сложные ее версии.

Знакомство с игрой «Прерванный полет»

Проект, разработке которого посвящена эта глава, называется «Прерванный полет», моя версия классической аркады «Астероиды». Пользователь, играя в «Прерванный полет», управляет космическим кораблем, который движется сквозь пояс смертоносных каменистых астероидов. Корабль может вращаться и линейно ускоряться, а кроме того, он вооружен ракетами. Ракетный удар по астероиду разрушает его. Впрочем, это не снимает разом всех проблем, потому что астероиды крупного и среднего размера, взрываясь, дробятся на две части. Как только игрок справится со всеми астероидами, на корабль налетит новая, еще более мощная волна этих космических убийц. Счет игрока увеличивается с каждым разрушенным астероидом, но как только звездолет столкнется с одним из них, игре наступит конец. Игровой процесс иллюстрируют рис. 12.1¹ и 12.2.

¹ Фоновая картинка с туманностью находится в открытом доступе. Авторство: NASA, The Hubble Heritage Team – AURA/STScI.



Рис. 12.1. Игрок управляет космическим кораблем. Чтобы набирать очки, надо взрывать астероиды



Рис. 12.2. Астероид врезался в корабль игрока и прервал полет

Чтение с клавиатуры

Как сохранять пользовательский клавиатурный ввод в строку с помощью функции `input()`, вы уже знаете. Но совсем другое дело — регистрация отдельных нажатий клавиш. Впрочем, в модуле `games` есть особый объект, функциональность которого в этом и состоит.

Знакомство с программой «Читаю с клавиатуры»

В графическом окне программы «Читаю с клавиатуры» изображен космический корабль на фоне далекой туманности. Нажимая разные клавиши, пользователь может передвигать корабль по экрану. Так, нажатие `W` сдвинет корабль вверх, `S` — вниз, `A` — влево, а `D` — вправо. Можно также использовать сочетания клавиш, чтобы добитьсяся комбинированного эффекта. Например, при одновременном нажатии `W` и `D` корабль сместится по диагонали вправо и вверх. Вид окна программы изображен на рис. 12.3.

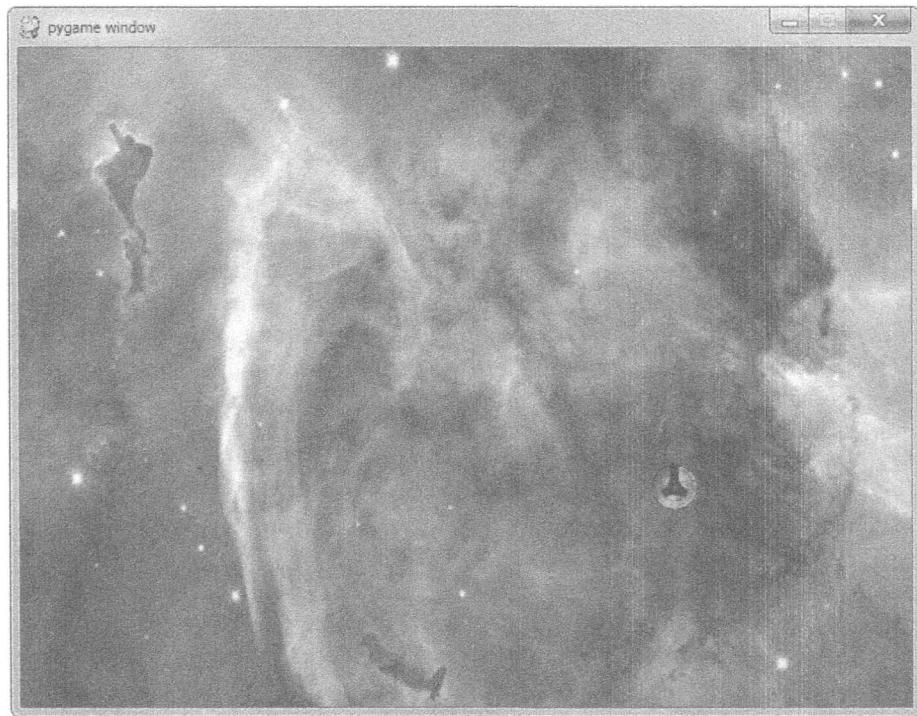


Рис. 12.3. Нажимая клавиши, пользователь передвигает корабль по экрану

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `read_key.py`.

Настройка программы

Как и во всех программах на основе пакета `livewires`, в самом начале кода я импортирую необходимые модули и вызываю метод, инициализирующий окно:

```
# Читаю с клавиатуры
# Демонстрирует чтение клавиатурного ввода
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Регистрация нажатий

Следующее на повестке дня — создать класс для представления корабля. В методе `update()` проверяется, нажата ли хоть одна из клавиш, обрабатываемых программой. Если да, то положение корабля будет соответствующим образом изменено.

```
class Ship(games.Sprite):
    """ Подвижный космический корабль. """
    def update(self):
        """ Перемещает корабль определенным образом, исходя из нажатых клавиш. """
        if games.keyboard.is_pressed(games.K_w):
            self.y -= 1
        if games.keyboard.is_pressed(games.K_s):
            self.y += 1
        if games.keyboard.is_pressed(games.K_a):
            self.x -= 1
        if games.keyboard.is_pressed(games.K_d):
            self.x += 1
```

Здесь я применил новый для вас объект `keyboard` из модуля `games`. С его помощью удобно обнаруживать нажатия определенных (заранее известных) клавиш. Я вызываю метод `is_pressed()` объекта `keyboard`, который возвращает `True`, если пользователь удерживает опрашиваемую клавишу, и `False` — если нет.

Метод `is_pressed()` используется в моем коде несколько раз, в последовательности условных конструкций, призванных регистрировать и обрабатывать нажатия клавиш **W**, **S**, **A** и **D**. Если пользователь нажал **W**, ордината (`y`) объекта уменьшится на 1 и спрайт передвинется по экрану на один пиксель вверх. Если нажата **S**, ордината увеличится на 1 и спрайт передвинется вниз. Точно так же нажатие **A** уменьшает абсциссу (`x`) на 1, отодвигая спрайт влево, а нажатие **D** увеличивает абсциссу на 1, отодвигая спрайт вправо.

Поскольку четыре вызова `is_pressed()` позволяют обнаружить вплоть до четырех одновременных нажатий, пользователю ничто не мешает удерживать несколько клавиш, чтобы добиться смешанного эффекта. Так, например, одновременное нажатие **D** и **S** перемещает корабль вправо и вниз: каждый раз при исполнении `update()`, то есть при каждой итерации основного цикла, абсцисса и ордината объекта `Ship` увеличиваются на единицу.

В модуле `games` задан набор констант, представляющих разные клавиши. Эти константы можно передавать как аргументы методу `is_pressed()`. В данной программе клавишу **W** представляет константа `games.K_w`, клавишу **S** — `games.K_s`, **A** — `games.K_a`

и D — games.K_d. Имена констант вполне понятны. Вообще большинство констант, о которых здесь идет речь, подчиняется следующим правилам.

- Каждое имя клавиатурной константы начинается с games.K_.
- Если рассматривается клавиша с буквой, то за префиксом должна следовать эта буква в нижнем регистре. К примеру, константа, связанная с клавишей A, называется games.K_a.
- Если рассматривается клавиша с цифрой, то за префиксом должна следовать эта цифра. Так, константа, связанная с клавишей 1, называется games.K_1.
- Для большинства остальных клавиш к префикску добавляется их английское название, набранное прописными буквами. Пробелу, например, соответствует константа games.K_SPACE.

Полный список клавиатурных констант приводится в документации livewires в приложении В.

Обертка программы

В конце следует знакомая вам функция main(). Она загружает фоновую картинку с туманностью, создает посередине экрана спрайт с изображением космического корабля и запускает работу графического окна вызовом mainloop().

```
def main():
    nebula_image = games.load_image("nebula.jpg", transparent = False)
    games.screen.background = nebula_image
    ship_image = games.load_image("ship.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)
    games.screen.mainloop()
main()
```

Вращение спрайта

Из главы 11 вы узнали, как перемещать спрайты по экрану. Однако обширная функциональность livewires позволяет также вращать спрайты. За вращение отвечает одно из свойств объекта-спрайта.

Знакомство с программой «Крутящийся спрайт»

В этой программе пользователь, нажимая клавиши, вращает корпус космического корабля. Если удерживать клавишу →, спрайт будет вертеться по часовой стрелке, а если клавишу ←, то против часовой стрелки. При нажатии 1 корабль будет возвращаться в исходное положение с поворотом 0°. Нажатие клавиши 2 заставит его повернуться вправо на 90°, нажатие 3 — перевернуться вверх ногами (180°), а нажатие 4 — совершить поворот на 270°, то есть визуально опрокинуться влево на четверть окружности. Работа программы иллюстрируется на рис. 12.4.

ЛОВУШКА

«Крутящийся спрайт» обрабатывает нажатия тех клавиш с цифрами, которые расположены верхним рядом над буквами, а не те, которые вынесены на отдельную панель.

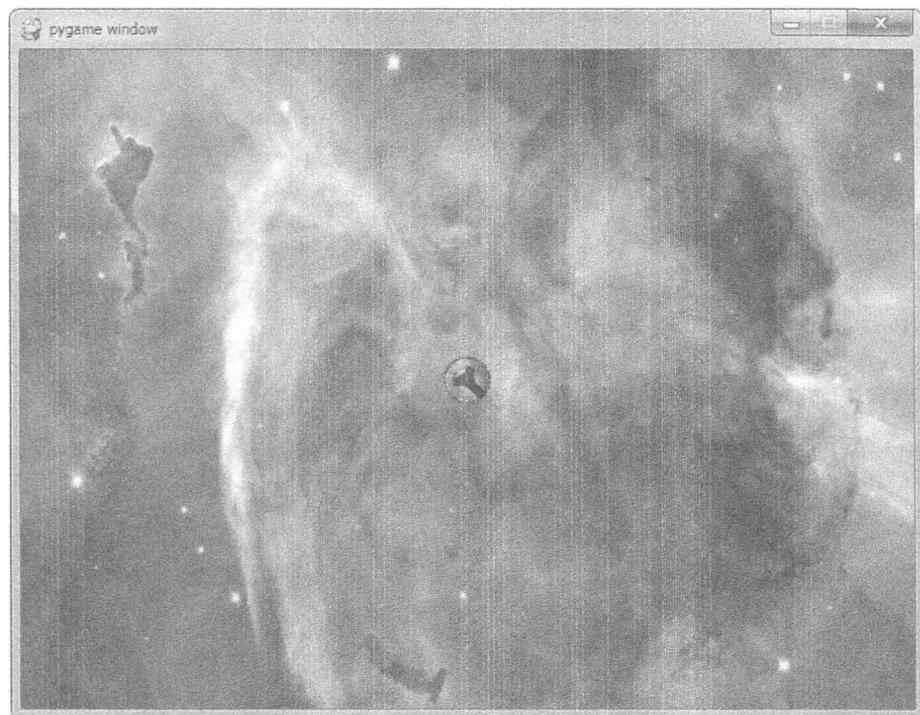


Рис. 12.4. Корабль может вращаться по часовой стрелке, против нее, а также отклоняться от исходного положения на фиксированные углы

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `rotate_sprite.py`.

```
# Крутящийся спрайт
# Демонстрирует вращение спрайта
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
class Ship(games.Sprite):
    """ Вращающийся космический корабль. """
    """ Вращает корабль определенным образом, исходя из нажатых клавиш. """
    def update(self):
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += 1
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= 1
        if games.keyboard.is_pressed(games.K_1):
            self.angle = 0
        if games.keyboard.is_pressed(games.K_2):
            self.angle = 90
```

```
if games.keyboard.is_pressed(games.K_3):
    self.angle = 180
if games.keyboard.is_pressed(games.K_4):
    self.angle = 270
def main():
    nebula_image = games.load_image("nebula.jpg", transparent = False)
    games.screen.background = nebula_image
    ship_image = games.load_image("ship.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)
    games.screen.mainloop()
main()
```

Применение свойства angle у спрайтов

В этой программе в новинку для вас должно быть свойство `angle`. Оно представляет угол наклона спрайта относительно его начального положения (в градусах). Значение этого свойства можно увеличивать, уменьшать, а можно просто присваивать ему новое значение, тем самым изменения угол наклона.

В методе `update()` я первым делом проверил, нажата ли клавиша →. Если да, то значение свойства `angle` объекта `Ship` увеличивается на единицу и, таким образом, корабль поворачивается по часовой стрелке на 1°. Затем проверяется, нажата ли клавиша ←. Если это так, то значение `angle` уменьшается на единицу и корабль тем самым поворачивается на 1° против часовой стрелки.

Дальнейшие строки кода реализуют поворот корабля на заранее назначенный угол: в них свойству `angle` просто присваивается новое значение. При нажатии 1 свойство `angle` становится равным 0 и спрайт возвращается в начальную позицию с углом наклона картинки 0°. После нажатия 2 `angle` получает значение 90, тем самым скачкообразно перемещая спрайт в положение с углом наклона 90°. Если пользователь нажмет 3, то программа сделает `angle` равным 180 и спрайт перевернется. Наконец, если пользователь нажмет 4, программа сделает `angle` равным 270, вызывая поворот спрайта на соответствующий угол.

Создание анимации

Движение и вращение спрайтов добавляет игре занимательности, но по-настоящему оживить игровой процесс может только анимация. Для нее в модуле `games` тоже предусмотрен класс, который так и называется — `Animation`.

Знакомство с программой «Взрыв»

Программа «Взрыв» воспроизводит в центре графического экрана анимированную картину взрыва. Анимация воспроизводится раз за разом, так что вы сможете хорошоенько ее рассмотреть. Когда вы, налюбовавшись эффектом, закроете окно,

программа прекратит работу. Один из моментов воспроизведения анимации показан на рис. 12.5.

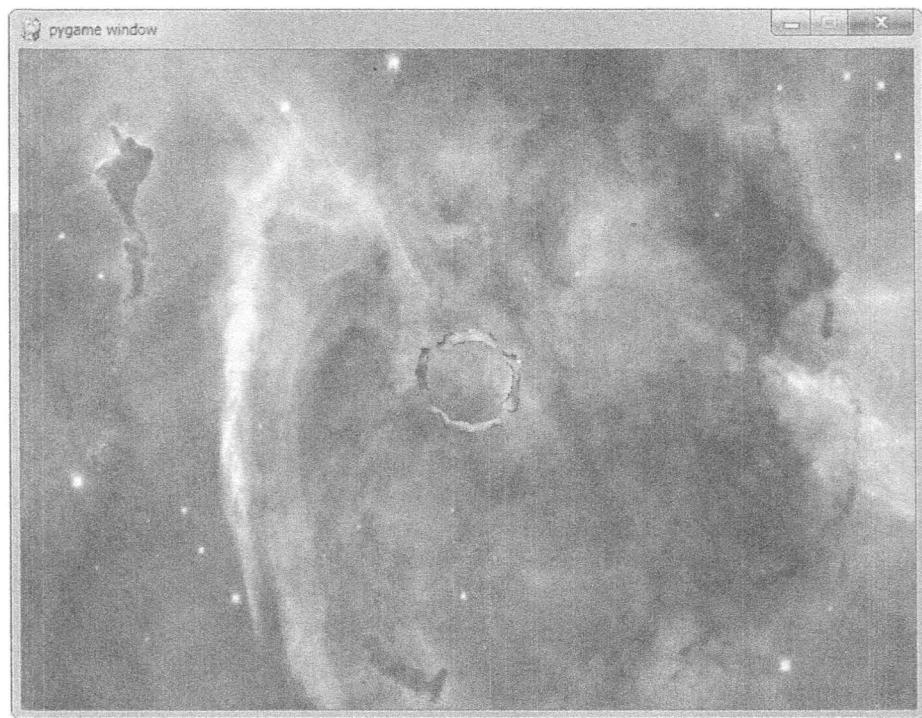


Рис. 12.5. По одному снимку трудно поверить, но здесь, в центре графического окна, происходит настоящий анимированный взрыв

Посмотрим на картинки

Анимация — это последовательность картинок (называемых также *кадрами*), которые отображаются одна за другой. Я создал последовательность из девяти картинок, которые, будучи показанными на экране друг за другом, создают иллюзию взрыва. Все девять картинок представлены на рис. 12.6.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `explosion.py`.

Настройка программы

Как всегда, в начале импортируются нужные модули и вызывается инициализирующий метод:

```
# Взрыв
# Демонстрирует создание анимации
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

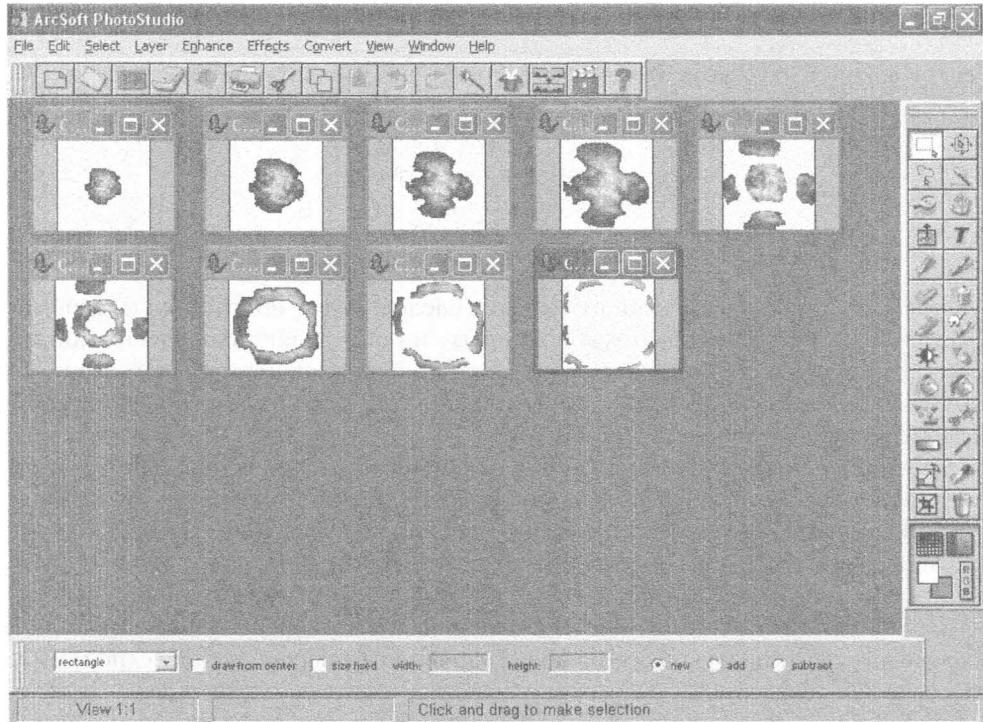


Рис. 12.6. Быстрая смена этих девяти кадров и создает впечатление, будто произошел взрыв

Вслед за тем назначается фоновая картинка графического окна:

```
nebula_image = games.load_image("nebula.jpg", transparent = 0)
games.screen.background = nebula_image
```

Создание списка изображений

Конструктор класса *Animation* принимает список имен графических файлов — картинок, которые будут последовательно отображаться. Следующим шагом я создам такой список. Он будет содержать имена картинок, показанных на рис. 12.6:

```
explosion_files = ["explosion1.bmp",
                    "explosion2.bmp",
                    "explosion3.bmp",
                    "explosion4.bmp",
                    "explosion5.bmp",
                    "explosion6.bmp",
                    "explosion7.bmp",
                    "explosion8.bmp",
                    "explosion9.bmp"]
```

Создание анимированного объекта

Теперь наконец я создам объект класса `Animation` и добавлю его на экран:

```
explosion = games.Animation(images = explosion_files,
                             x = games.screen.width/2,
                             y = games.screen.height/2,
                             n_repeats = 0,
                             repeat_interval = 5)
games.screen.add(explosion)
```

Класс `Animation` произведен от `Sprite` и наследует все его атрибуты, свойства и методы. Как и у всех спрайтов, координаты `x` и `y` определяют размещение анимированного рисунка по горизонтали и вертикали. В предшествующем коде конструктору класса были переданы такие координаты, что анимация окажется ровно посередине графического экрана.

Анимированный объект отличается от спрайта тем, что основан не на одном рисунке, а на нескольких, которые циклически сменяют друг друга. Поэтому вы должны будете перечислить списком все необходимые картинки как строки (имена файлов) или как имена объектов-изображений. Мой список `explosion_files` состоит из строк.

Атрибут объекта `n_repeats` определяет количество повторений анимации (то есть последовательных запусков всего набора картинок, от первой до последней). Если присвоить ему значение 0, то анимация будет повторяться не переставая. По умолчанию атрибут `n_repeats` равен 0. Это же значение передал ему и я; таким образом, мою анимацию графическое окно будет воспроизводить до тех пор, пока пользователь его не закроет.

Атрибут объекта `repeat_interval` определяет интервал (количество обновлений экрана) между двумя последовательными членами анимированного ряда. Чем больше это число, тем длиннее задержка и тем медленнее анимация; чем оно меньше, тем анимация, соответственно, быстрее. Я передал параметру `repeat_interval` значение 5. Полученная скорость, на мой взгляд, убедительно изображает взрыв.

Осталось запустить все это, вызвав метод `mainloop()` объекта `screen`:

```
games.screen.mainloop()
```

Работа со звуком и музыкой

Звуки и музыкальные фрагменты добавят вашей программе новые возможности. Загрузка, воспроизведение, циклизация, а также остановка звука и музыки предельно просто выполняются с помощью модуля `games`. Хотя между звуком и музыкой трудно провести четкую границу, в `livewires` она проводится совершенно недвусмысленно, без каких бы то ни было сомнений.

Знакомство с программой «Звук и музыка»

«Звук и музыка» — программа, которая позволяет пользователю запускать, циклически воспроизводить и останавливать музыкальную тему игры «Прерванный

полет» и звуковой эффект пуска ракет, можно даже одновременно. На рис. 12.7 показан внешний вид программы (к сожалению, без звука).

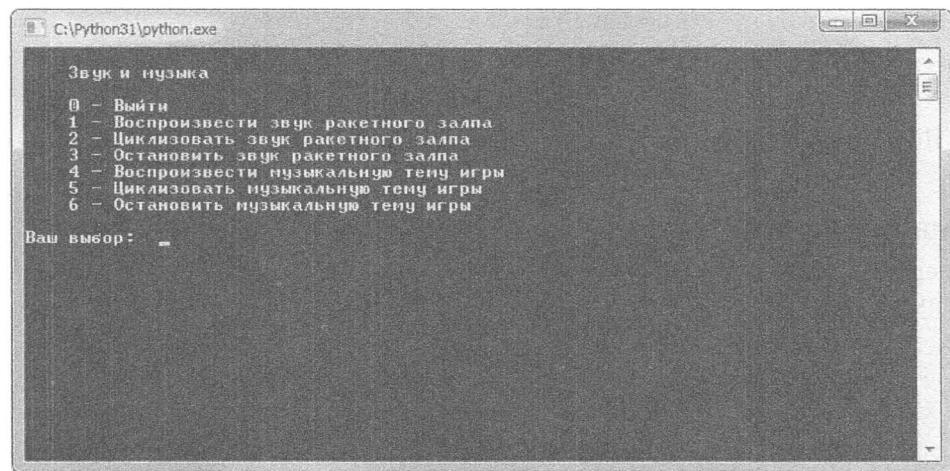


Рис. 12.7. Пользователь может воспроизвести звуковой фрагмент и музикальную тему

ПОДСКАЗКА

После запуска программы пользователь будет взаимодействовать с консольным окном. Чтобы это стало возможным, надо разместить консоль так, чтобы графическое окно не перекрывало доступ к нему. Иными словами, созданный программой графический экран можно свернуть или вовсе закрыть.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `sound_and_music.py`.

Работа со звуком

Для того чтобы создать звуковой объект и пользоваться им далее в программе, надо сначала загрузить WAV-файл. Формат WAV удобен для звуковых эффектов, так как успешно кодирует любой звук, записанный с микрофона.

Загрузка звука

Сначала выполняется обычная настройка программы:

```
# Звук и музыка
# Демонстрирует воспроизведение звуков и музикальных файлов
from livewires import games
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Теперь с помощью функции `load_sound()` модуля `games` я загружу WAV-файл.

```
# загрузка звукового файла
missile_sound = games.load_sound("missile.wav")
```

Эта функция принимает строку — имя звукового файла, который надо загрузить. Я загружаю `missile.wav` и связываю полученный звуковой объект с переменной `missile_sound`.

ЛОВУШКА

WAV-файлы можно загружать исключительно функцией `load_sound()`.

Теперь я загружу музыкальный фрагмент:

```
# загрузка музыкального файла
games.music.load("theme.mid")
```

Почему так, а не иначе и в чем вообще специфика музыки по сравнению со звуком, обсудим несколько позже.

Воспроизведение звука

Теперь я создам текстовое меню по тому образцу, с которым вы впервые познакомились в главе 5:

```
choice = None
while choice != "0":
    print(
        """
Звук и музыка
0 - Выйти
1 - Воспроизвести звук ракетного залпа
2 - Циклизовать звук ракетного залпа
3 - Остановить звук ракетного залпа
4 - Воспроизвести музыкальную тему игры
5 - Циклизовать музыкальную тему игры
6 - Остановить музыкальную тему игры
"""
    )
    choice = input("Ваш выбор: ")
    print()
    # выход
    if choice == "0":
        print("До свидания..")
```

Если пользователь вводит 0, то программа прощается с ним и завершает работу. Следующий код обрабатывает выбор пользователем альтернативы 1:

```
# воспроизведение звука ракетного залпа
elif choice == "1":
    missile_sound.play()
    print("Воспроизвожу звук ракетного залпа..")
```

Чтобы воспроизвести звук всего лишь однократно, язываю метод `play()` звукового объекта. Воспроизводимый звук занимает один из восьми доступных звуковых каналов. Таким образом, для воспроизведения звука нужно, чтобы по крайней мере один звуковой канал был открыт. Если все восемь каналов заняты, по вызову метода `play()` ничего не произойдет.

Если применить метод `play()` к звуковому объекту, который уже проигрываеться, то система попробует воспроизвести тот же звук по любому из свободных каналов, если таковые есть.

Циклизация звука

Звук можно циклизовать, если передать методу `play()` параметр — число, описывающее количество повторных звучаний. Если, например, передать методу `play()` число 3, то соответствующий звук будет воспроизведен в общей сложности четыре раза (один раз сначала и еще три потом). Бесконечный цикл создается значением -1.

Следующий код обрабатывает пользовательский выбор второй альтернативы:

```
# циклизация звука ракетного залпа
elif choice == "2":
    loop = int(input("Сколько еще раз воспроизвести этот звук? (-1 = воспроизводить
не переставая): "))
    missile_sound.play(loop)
    print("Циклизую звук ракетного залпа.")
```

В этом фрагменте кода программа спрашивает у пользователя, сколько еще раз тот хочет услышать ракетный залп, а затем полученное значение передается методу `play()` звукового объекта.

Остановка звука

Для того чтобы остановить проигрывание звукового объекта, достаточно вызвать его метод `stop()`. При этом звук будет остановлен на всех каналах, на которых он в данный момент звучит. Если применить метод `stop()` к звуковому объекту, не запущенному на исполнение, то `livewires` поведет себя милостиво и не сообщит об ошибке.

Итак, если пользователь ввел 3 и в это время звучал ракетный залп, звук будет остановлен:

```
# остановка звука ракетного залпа
elif choice == "3":
    missile_sound.stop()
    print("Останавливаю звук ракетного залпа.")
```

Работа с музыкой

В пакете `livewires` музыка обрабатывается совсем иначе, чем звук. Музыкальный канал всего один, так что в любой момент «актуальная» музыка — это ровно один файл. Впрочем, музыкальный канал проявляет больше гибкости, чем все восемь звуковых: он умеет работать с множеством разных форматов, в том числе WAV, MP3, OGG и MIDI. Наконец, ввиду того факта, что канал единственный, вам не придется создавать для каждого музыкального файла свой объект. Вместо этого существует инструментарий функций, ответственных за загрузку, исполнение и остановку музыки.

Загрузка музыки

Код, с помощью которого загружается музыкальный файл, вы видели в разделе «Загрузка звука». Он пользуется объектом `music` из модуля `games`. Именно через `music` музыкальный трек можно загружать, запускать на воспроизведение и останавливать.

Использованный мною для загрузки файла код `games.music.load("theme.mid")` означает, что будет загружен музыкальный MIDI-файл `theme.mid`. Вообще загрузку музыкального трека обеспечивает функция `games.music.load()`, которой надо передать имя загружаемого файла — строку.

Поскольку, как известно, в момент времени трек всего один, то вновь загруженный музыкальный файл просто заменит более ранний.

Воспроизведение музыки

Следующий код обрабатывает пользовательский выбор четвертого пункта меню:

```
# воспроизведение музыкальной темы
elif choice == "4":
    games.music.play()
    print("Исполняю музыкальную тему игры.")
```

Как результат работы этого кода компьютер воспроизведет загруженный файл `theme.mid`. Если не передавать функции `games.music.play()` никаких значений, то трек будет исполнен однократно.

Циклизация музыки

Для того чтобы воспроизводить музыку циклически, надо передать методу `play()` желаемое количество повторных звучаний. Так, например, если вызвать `games.music.play()` с аргументом 3, то музыка прозвучит четыре раза (один раз сначала и еще три потом). Бесконечный цикл создается значением -1.

Следующий код обрабатывает пользовательский выбор пятой альтернативы:

```
# циклизация музыкальной темы
elif choice == "5":
    loop = int(input("Сколько еще раз воспроизвести эту музыку? (-1 = воспроизводить
не переставая): "))
    games.music.play(loop)
    print("Циклизую музыкальную тему игры.")
```

В этом фрагменте кода программа спрашивает у пользователя, сколько еще раз тот хочет услышать музыкальную тему, а затем полученное значение передается методу `play()`.

Остановка музыки

Если пользователь выберет вариант 6, то играющая музыка будет остановлена благодаря такому коду:

```
# остановка музыкальной темы
elif choice == "6":
    games.music.stop()
    print("Останавливаю музыкальную тему игры.")
```

Для того чтобы текущий трек перестал играть, достаточно вызвать `games.music.stop()`, что я и делаю. Если вызвать функцию в тот момент, когда никакой музыки не воспроизводится, то `livewires` поведет себя милостиво и не сообщит об ошибке.

Обертка программы

В заключение программа обрабатывает ошибочный ввод и ожидает пользователя:

```
# непонятный пользовательский ввод
else:
    print("Извините, в меню нет пункта", choice)
input("\n\nНажмите Enter, чтобы выйти.")
```

Разработка игры «Прерванный полет»

Вернемся к проекту, которому посвящена эта глава, — к игре «Прерванный полет». Будем создавать последовательно все более сложные версии игры до тех пор, пока программа не окажется готова. Прежде чем непосредственно перейти к делу, я считаю уместным обговорить кое-какие подробности: основные функции игры, необходимые для нее классы и мультимедийные ресурсы.

Функциональность игры

Хотя моя разработка основана на классической видеоигре, которую я хорошо знаю (и когда-то даже тщательно изучал), все же не помешает выписать в явном виде функциональность, которую следует реализовать.

- Корабль должен вращаться и пробивать себе путь вперед, направляемый пользовательским вводом с клавиатуры.
- При нажатии игроком определенной клавиши корабль должен выпускать ракеты.
- Астероиды должны перемещаться по экрану с разными скоростями — как правило, маленькие быстрее больших.
- Корабль, выпускаемые им ракеты и все астероиды должны, выходя за пределы графического экрана, «огибать» его, то есть появляться с противоположной стороны.
- Если ракета поражает какой-либо другой объект на экране, то и она, и этот объект должны красиво взрываться.
- Если корабль сталкивается с каким-либо другим объектом на экране, то и он, и этот объект должны красиво взрываться.
- Когда корабль гибнет, игра прекращается.
- При разрушении большого астероида должна появляться пара астероидов среднего размера, при разрушении астероида среднего размера — пара мелких астероидов, а вот эти мелкие астероиды далее уже не делятся.
- Каждый раз, когда игрок разрушает астероид, его счет должен увеличиваться. Чем меньше астероид, тем больше очков начисляется.
- Текущий счет должен быть виден в правом верхнем углу экрана.

- После того как все астероиды будут разрушены, на корабль должна налетать очередная, более массовая «волна» астероидов.
- Для того чтобы не усложнять дело, я в этом перечне не назвал некоторые особенности исходной видеоигры.

Классы игры

Теперь я составлю список классов, которые, как думается, понадобятся мне в разработке:

- Ship — корабль;
- Missile — ракета;
- Asteroid — астероид;
- Explosion — взрыв.

О природе этих классов мне уже кое-что известно. Так, Ship, Missile и Asteroid будут подклассами games.Sprite, а Explosion — подклассом games.Animation. Я не забываю также, что первоначальный список может измениться после того, как я перейду от теории к практике.

Медиаресурсы

Поскольку в игре, как ожидается, будут присутствовать звук, музыка, спрайты и анимация, я понимаю, что надо перечислить соответствующие мультимедийные файлы. Вот какой список получился:

- картинка — изображение корабля;
- картинка — изображение ракеты;
- по одной картинке для представления каждого из трех типов астероидов;
- набор изображений, представляющих взрывы;
- звуковой файл для случая, когда корабль пробивает себе дорогу вперед;
- звуковой файл для запуска ракеты;
- звуковой файл, сопровождающий взрыв чего-либо;
- музыкальная тема.

Создание астероидов

Поскольку в игре есть смертоносные астероиды, пожалуй, с них и начнем разработку. Как мне кажется, это наилучший первый шаг, но я субъективен и мое мнение может не совпасть с мнением коллег-программистов. Это, однако, не проблема. Вам, возможно, захочется сначала создать корабль игрока на экране — и вы тоже будете правы, потому что единственно верного первого шага в разработке не существует. Важно лишь разделить задачу на легко реализуемые части, тогда они, «надстраиваясь» друг над другом, постепенно образуют законченный проект.

Программа «Прерванный полет — 1»

В первой версии «Прерванного полета» появляются графическое окно, фон с изображением туманности, а также восемь астероидов, случайным образом разбросанных по этому фону. Скорость каждого из астероидов вычисляется тоже случайно, но мелкие астероиды склонны перемещаться быстрее крупных. Работу программы иллюстрирует рис. 12.8.



Рис. 12.8. Сначала было поле с движущимися астероидами

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется astrocrash01.py.

Настройка программы

Первые строки кода не блещут новизной:

```
# Прерванный полет-1  
# Только астероиды движутся по экрану  
import random  
from livewires import games  
games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Модуль random импортирован для того, чтобы астероидам можно было присваивать случайные координаты.

Класс Asteroid

Класс Asteroid нужен для того, чтобы создавать движущиеся астероиды:

```
class Asteroid(games.Sprite):
    """ Астероид, прямолинейно движущийся по экрану. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images = {SMALL : games.load_image("asteroid_small.bmp"),
              MEDIUM : games.load_image("asteroid_med.bmp"),
              LARGE : games.load_image("asteroid_big.bmp") }
    SPEED = 2
```

Первое, что я здесь сделал, — объявил три константы класса, `SMALL`, `MEDIUM` и `LARGE`, соответствующие трем возможным размерам астероидов. После этого я создал словарь, в котором этим константам отвечают объекты-изображения астероидов соответствующих размеров. Таким образом, обращаясь к константе размера, можно непосредственно получить доступ к соответствующей картинке. Наконец, в последней строке показанного кода объявляется константа `SPEED`, на основе которой будет определяться случайная скорость каждого из астероидов.

Метод `__init__`

Теперь я создам метод-конструктор:

```
def __init__(self, x, y, size):
    """ Инициализирует спрайт с изображением астероида. """
    super(Asteroid, self).__init__(
        image = Asteroid.images[size],
        x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED * random.random()/size)
    self.size = size
```

Значение, передаваемое параметру `size`, представляет собой размер астероида и должно совпадать с одной из констант размера: `Asteroid.SMALL`, `Asteroid.MEDIUM` или `Asteroid.LARGE`. Исходя из `size` спрайт получает подходящую картинку: она передается параметру `image` конструктора класса `Sprite` (родительского по отношению к `Asteroid`). Значения `x` и `y`, описывающие положение нового астероида внутри графического окна, также передаются конструктору `Sprite`.

Метод-конструктор класса `Asteroid` вычисляет случайные значения горизонтальной и вертикальной компонент скорости нового объекта, которые, в свою очередь, вновь передаются конструктору `Sprite`. Это на самом деле случайные числа, но астероиды меньшего размера, как можно понять из кода, имеют потенциал двигаться быстрее. В заключение конструктор класса `Asteroid` создает и инициализирует у объекта атрибут `size`.

Метод `update()`

Этот метод не позволяет астероиду покинуть графическое окно, заставляя его, если он вышел за границы окна, появиться с противоположной стороны:

```
def update(self):
    """ Заставляет астероид обогнуть экран. """
    if self.top > games.screen.height:
        self.bottom = 0
    if self.bottom < 0:
        self.top = games.screen.height
    if self.left > games.screen.width:
        self.right = 0
    if self.right < 0:
        self.left = games.screen.width
```

Функция main()

Наконец, функция `main()` назначает фоновую картинку с туманностью и создает восемь астероидов в разных местах экрана:

```
def main():
    # назначаем фоновую картинку
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image
    # создаем 8 астероидов
    for i in range(8):
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL, Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)
    games.screen.mainloop()
# поехали!
main()
```

Вращение корабля

Следующая цель — создать звездолет игрока. Скромную первоочередную задачу я вижу в том, чтобы дать пользователю возможность вращать корабль по часовой стрелке и против нее. К реализации других функций корабля я намерен приступить позднее.

Программа «Прерванный полет — 2»

«Прерванный полет — 2» расширяет функциональность «Прерванного полета — 1». В этой новой версии посередине экрана появляется космический корабль, который игрок может вращать. Нажатие клавиши → заставляет корабль повернуться по часовой стрелке, а клавиши ← — против часовой стрелки. Работу программы иллюстрирует рис. 12.9.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash02.py`.



Рис. 12.9. Теперь в дело вступил корабль игрока

Класс Ship

Главное, что здесь надо реализовать, — класс `Ship`, представляющий корабль игрока:

```
class Ship(games.Sprite):
    """ Корабль игрока. """
    image = games.load_image("ship.bmp")
    ROTATION_STEP = 3
    def update(self):
        """ Вращает корабль при нажатии клавиш со стрелками. """
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

По реализации этот класс похож на программу «Крутящийся спрайт», показанную ранее в этой главе. Но все же есть некоторые отличия. Во-первых, объект-изображение корабля связывается с переменной `image` класса. Во-вторых, минимальный угол, на который может одномоментно повернуться корабль, определяет константа класса `ROTATION_STEP`.

Инстанцирование класса `Ship`

Последнее, что осталось сделать в этой новой версии игры, — создать экземпляр класса `Ship` и вывести его на экран. Новый корабль создается в следующем коде функции `main()`:

```
# создаем корабль
the_ship = Ship(image = Ship.image,
                 x = games.screen.width/2,
                 y = games.screen.height/2)
games.screen.add(the_ship)
```

Движение корабля

В очередной версии программы я решил сделать корабль подвижным. Нажатием клавиши ↑ игрок включает главный двигатель, и корабль совершают рывок в том направлении, в котором он ориентирован. Поскольку физическая картина нашей игры не учитывает трения, корабль будет двигаться ровно столько, сколько игрок удерживает нажатой клавишу.

Программа «Прерванный полет — 3»

Когда в «Прерванном полете — 3» игрок включает двигатель корабля, скорость корабля меняется подходящим образом, исходя из угла его наклона, и звучит характерный звук. Работу программы иллюстрирует рис. 12.10.



Рис. 12.10. Теперь корабль может двигаться по экрану

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash03.py`.

Импорт модуля math

Первым делом здесь импортируется новый модуль:

```
import math, random
```

Модуль содержит несколько математических функций и констант, но пусть это вас не пугает: в нашей программе их применяется всего несколько, к тому же не самых сложных.

Добавление переменной и константы в класс Ship

Изменение скорости корабля будет в дальнейшем описывать константа `VELOCITY_STEP` класса `Ship`:

```
VELOCITY_STEP = .03
```

Чем она больше, тем быстрее будет расти скорость корабля после запуска двигателя; чем меньше, тем, соответственно, медленнее.

Я создам также новую переменную `sound`, с которой будет связан звуковой объект — звук ускоряющегося рывка:

```
sound = games.load_sound("thrust.wav")
```

Изменение метода update() объекта Ship

Теперь я добавлю в конец метода `update()` в классе `Ship` код, ответственный за перемещение корабля по экрану. Сначала выполняется проверка: нажал ли игрок клавишу `↑`. Если да, то воспроизводится звуковой эффект:

```
# корабль совершает рывок
if games.keyboard.is_pressed(games.K_UP):
    Ship.sound.play()
```

Теперь, поскольку известно, что игрок нажал клавишу `↑`, надо как-то изменить значения обеих компонент скорости корабля — свойств `dx` и `dy` объекта `Ship`. Итак, каким образом можно, зная угол наклона корабля, вычислить необходимое изменение скорости по каждому из двух перпендикулярных векторов? Решим этот вопрос из тригонометрических соображений (только не спешите, пожалуйста, бросать книгу и бежать с тоскливым воем куда глаза глядят). Как и обещал, я воспользуюсь всем парой математических функций, а все вычисления займут несколько строк кода.

Начну с того, что преобразую угол наклона корабля из градусов в радианы:

```
# изменение горизонтальной и вертикальной скорости корабля с учетом угла поворота
angle = self.angle * math.pi / 180 # преобразование в радианы
```

Радиан — это, как и градус, мера угла. В Python модуль `math` работает с углами, заданными в радианах (тогда как `livewires` — с углами, заданными в градусах), что и вынуждает выполнить это преобразование, в котором, кстати, используется константа `pi` модуля `math`, представляющая число «пи».

Теперь, зная угол в радианах, с помощью функций `sin()` и `cos()` модуля `math` я могу вычислить, каким должно быть изменение скорости по горизонтальной и вертикальной компонентам соответственно¹. Следующие строки кода вычисляют новые значения `dx` и `dy` нашего объекта:

```
self.dx += Ship.VELOCITY_STEP * math.sin(angle)
self.dy += Ship.VELOCITY_STEP * -math.cos(angle)
```

Попросту говоря, `math.sin(angle)` характеризует долю ускорения, которая должна быть передана горизонтальной скорости корабля, а `-math.cos(angle)` — долю ускорения, сообщаемую вертикальной скорости корабля².

Осталось описать только взаимодействие с границами экрана. Прибегну к той же стратегии, что и в случае с астероидами: корабль будет «огибать» экран. По сути, я просто копирую и вставляю код из метода `update()` класса `Asteroid` в конец одноименного метода класса `Ship`:

```
# корабль будет "огибать" экран
if self.top > games.screen.height:
    self.bottom = 0
if self.bottom < 0:
    self.top = games.screen.height
if self.left > games.screen.width:
    self.right = 0
if self.right < 0:
    self.left = games.screen.width
```

Хотя это и рабочий вариант, вообще копирование и вставка больших порций кода — признак плохо спроектированной программы. Далее мы еще вернемся к этому коду и найдем более элегантное решение.

ЛОВУШКА

Повторение отдельных фрагментов кода запутывает программу и затрудняет ее доработку. Если только у вас в программе появился повторяющийся код, это значит, что пришло время задуматься о новой функции или классе. Направьте свои усилия на то, чтобы свести повторяющийся код куда-либо в одно место и просто вызывать его из всех частей программы, в которых он сейчас дублируется.

Стрельба ракетами

Теперь я сделаю так, чтобы корабль мог стрелять ракетами. При нажатии клавиши Пробел бортовое орудие звездолета будет выпускать ракету в том же направлении, в котором ориентирован сам корабль. Ракета должна разрушать любой объект, встреченный ею на пути; но пока, чтобы не усложнять код преждевременно, я не стану реализовывать механизм разрушения.

¹ Именно в таком порядке, потому что при нулевом повороте корабль ориентирован строго вверх. — *Примеч. пер.*

² Минус при косинусе нужен потому, что ординаты отсчитываются сверху вниз, а не снизу вверх, как обычно в школьной математике. — *Примеч. пер.*

Программа «Прерванный полет — 4»

В программе «Прерванный полет — 4» игрок может, нажимая Пробел, выпускать ракеты. Есть, впрочем, одна проблема: если нажать Пробел и удерживать, ракеты будут вылетать неудержимым потоком, с частотой около 50 штук в секунду. Понятно, что темп стрельбы надо как-то ограничить, однако всему свое время — дождемся следующей версии игры. Как выглядит «Прерванный полет — 4», можно видеть на рис. 12.11.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется astrocrash04.py.

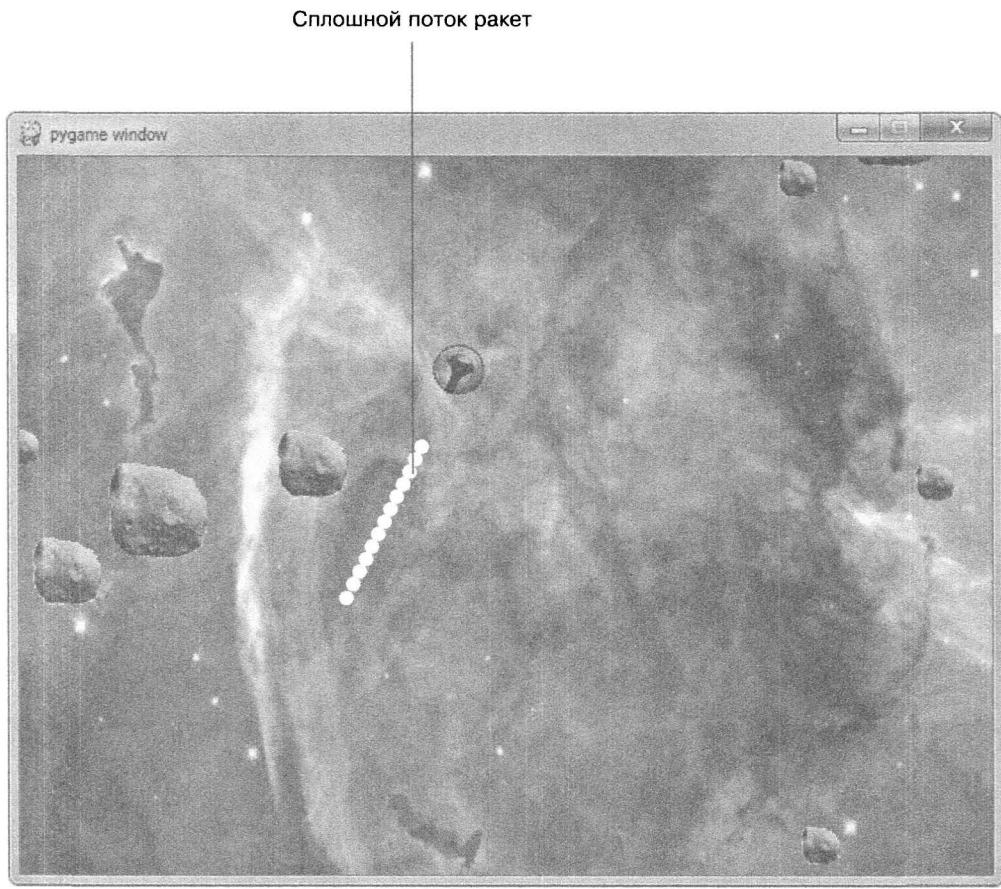


Рис. 12.11. Плотность огня слишком высока

Изменение метода update() объекта Ship

Я изменил метод update() в классе Ship, добавив код, который позволяет кораблю вести огонь. Как только игрок нажимает Пробел, создается новая ракета:

```
# если нажат Пробел, выпустить ракету
if games.keyboard.is_pressed(games.K_SPACE):
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
```

Чтобы по вызову конструктора `Missile(self.x, self.y, self.angle)` появился новый объект-ракета, не помешает создать класс `Missile`.

Класс `Missile`

Объекты класса `Missile` представляют ракеты, выпускаемые кораблем. Я начал с того, что объявил переменные и константы в этом классе:

```
class Missile(games.Sprite):
    """ Ракета, которую может выпустить космический корабль игрока. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40
```

Переменная `image` связана с картинкой — непрозрачным алым кружком, графически представляющим ракету. В переменную `sound` загружен звук, которым сопровождается ракетный залп. Константа `BUFFER` характеризует начальное удаление вновь созданной ракеты от корабля (это нужно для того, чтобы в момент запуска кружок появлялся не поверх изображения корабля, а около). Константа `VELOCITY_FACTOR` скавывается на скорости полета ракет. Наконец, `LIFETIME` — это продолжительность существования ракеты до ее самопроизвольного исчезновения (а исчезать ракеты должны, чтобы не летать беспорядочно по экрану, сколько им заблагорассудится).

Метод `__init__()`

Конструктор класса начинается со следующих строк:

```
def __init__(self, ship_x, ship_y, ship_angle):
    """ Инициализирует спрайт с изображением ракеты. """
```

Вас может удивить, что конструктор запрашивает о местоположении корабля и угле его наклона: соответствующие значения передаются параметрам `ship_x`, `ship_y` и `ship_angle`. Но все просто: метод должен определить, во-первых, где точно появится ракета, а во-вторых, какими будут горизонтальная и вертикальная компоненты ее скорости. Место появления ракеты зависит от местонахождения корабля, а ее дальнейший путь — и от угла наклона.

Здесь первый на очереди — звуковой эффект:

```
Missile.sound.play()
```

Затем, выполнив необходимые вычисления, я нахожу координаты точки, в которой появится ракета:

```
# преобразование в радианы
angle = ship_angle * math.pi / 180
```

```
# вычисление начальной позиции ракеты
buffer_x = Missile.BUFFER * math.sin(angle)
buffer_y = Missile.BUFFER * -math.cos(angle)
x = ship_x + buffer_x
y = ship_y + buffer_y
```

Угол поворота корабля конвертируется в радианы. После этого вычисляются начальные абсцисса и ордината ракеты; в основу этих вычислений положены уже известный нам угол и константа Missile.BUFFER. Полученные координаты x и y задают точку впереди корабля с учетом его ориентации в пространстве.

Теперь я рассчитываю горизонтальную и вертикальную компоненты скорости ракеты — по той же системе, что и в классе Ship:

```
# вычисление горизонтальной и вертикальной скорости ракеты
dx = Missile.VELOCITY_FACTOR * math.sin(angle)
dy = Missile.VELOCITY_FACTOR * -math.cos(angle)
```

Вызываю конструктор Sprite надкласса:

```
# создание ракеты
super(Missile, self).__init__(image = Missile.image,
                               x = x, y = y,
                               dx = dx, dy = dy)
```

Осталось только создать для ракеты (экземпляра класса Missile) атрибут lifetime, ограничивающий время ее странствий по космосу.

```
self.lifetime = Missile.LIFETIME
```

Метод update()

Теперь разработаем метод update(). Вот его первая часть:

```
def update(self):
    """ Перемещает ракету. """
    # если "срок годности" ракеты истек, она уничтожается
    self.lifetime -= 1
    if self.lifetime == 0:
        self.destroy()
```

Этот код всего-навсего ведет обратный отсчет, убавляя значение атрибута lifetime по единице. Когда оно окажется равным 0, объект Missile перестанет существовать.

Во второй части метода update() я реализовал уже знакомый механизм «огибания» графического окна:

```
# ракета будет огибать экран
if self.top > games.screen.height:
    self.bottom = 0
if self.bottom < 0:
    self.top = games.screen.height
if self.left > games.screen.width:
    self.right = 0
if self.right < 0:
    self.left = games.screen.width
```

Вот и в третий раз повторился один и тот же код. Определенно, надо с этим что-то делать.

Управление плотностью огня

В предыдущем примере было показано, что наш космический корабль может выпускать до 50 ракет в секунду. Даже для заядлого любителя легких побед это, пожалуй, многовато. Поэтому в очередной версии программы я ограничил плотность огня.

Программа «Прерванный полет — 5»

В «Прерванном полете — 5» интенсивность стрельбы снижена: после выстрела начинается обратный отсчет и, пока он не закончится, игрок не может выпустить следующую ракету. Работу программы иллюстрирует рис. 12.12.



Рис. 12.12. Теперь интервал между пусками ракет более приемлемый

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash05.py`.

Добавление константы в класс Ship

Для того чтобы заставить ракеты вылетать с определенным интервалом, я первым делом добавлю в класс `Ship` константу:

```
MISSILE_DELAY = 25
```

`MISSILE_DELAY` — это задержка. Представляет собой количество обновлений экрана, в течение которых игрок не может выпустить ракету после очередного запуска. Эта константа понадобится мне для того, чтобы заставить игрока стрелять более упорядоченно, с перерывами.

Создание метода-конструктора в классе Ship

Теперь я создам конструктор объектов-кораблей:

```
def __init__(self, x, y):
    """ Инициализирует спрайт с изображением космического корабля. """
    super(Ship, self).__init__(image = Ship.image, x = x, y = y)
    self.missile_wait = 0
```

Принимая значения абсциссы и ординаты вновь создаваемого корабля, этот метод передает их конструктору надкласса — `games.Sprite`. Затем у нового объекта объявляется атрибут `missile_wait`. С помощью `missile_wait` будет отсчитываться задержка, упреждающая запуск очередной ракеты.

Изменение метода update() объекта Ship

В классе `Ship` в метод `update()` я добавил обратный отсчет — последовательное уменьшение значения `missile_wait` до нуля.

```
# если запуск следующей ракеты пока еще не разрешен, вычесть 1 из длины оставшегося интервала ожидания
```

```
if self.missile_wait > 0:
    self.missile_wait -= 1
```

Код, который в предшествующей версии осуществлял запуск ракет, я тоже переписал:

```
# если нажат Пробел и интервал ожидания истек, выпустить ракету
if games.keyboard.is_pressed(games.K_SPACE) and self.missile_wait == 0:
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
    self.missile_wait = Ship.MISSILE_DELAY
```

Теперь, кроме нажатия **Пробела**, требуется, чтобы атрибут `missile_wait` был равным 0, то есть промежуток ожидания между двумя последовательными запусками ракет истек. Каждый следующий выстрел вновь приравнивает `missile_wait` к `MISSILE_DELAY`, и отсчет начинается заново.

Обработка столкновений

До сих пор игрок мог маневрировать среди астероидов и давать ракетные залпы, но взаимодействия между объектами не было. Совсем иначе стало в новой версии игры. Теперь ракета, сталкиваясь с каким-либо другим объектом, разрушает его, и себя. Корабль при столкновении тоже разрушает соприкоснувшийся с ним объект и гибнет сам. Астероиды ведут себя более пассивно: игра не предусматривает, чтобы астероиды, сталкиваясь, самоизвестно исчезали с экрана.

Программа «Прерванный полет — 6»

В «Прерванном полете — 6» все актуальные для нас столкновения регистрируются с помощью свойства `overlapping_sprites`, которым обладает каждый спрайт. Кроме того, здесь особым образом обрабатывается разрушение астероидов: крупные и средних размеров космические глыбы, распадаясь, делятся на пары более мелких.

ЛОВУШКА

Ввиду того, что астероиды в начале игры разбросаны по экрану совершенно случайным образом, может оказаться так, что один из них по расположению совпадет с кораблем пользователя, так что игра закончится, едва начавшись. Пока эту недоработку можно счесть простительной, но в заключительном варианте игры ее придется исправить.

Работу программы иллюстрирует рис. 12.13. Ее код вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash06.py`.

Изменение метода `update()` объекта `Missile`

В конец метода `update()` в классе `Missile` я добавил следующий код:

```
# проверка на перекрытие с другими объектами
if self.overlapping_sprites:
    for sprite in self.overlapping_sprites:
        sprite.die()
    self.die()
```

Если есть визуальное перекрытие ракеты с какими-либо иными объектами (по крайней мере одним), то и эти объекты, и сама ракета будут разрушены методом `die()`. Это новый метод, который я сейчас добавлю в классы `Asteroid`, `Ship` и `Missile`.

Добавление метода `die()` объекту `Missile`

Объект `Missile` нуждается в методе `die()` в той же мере, что и остальные классы текущей версии игры. Этот метод может быть реализован чрезвычайно просто:

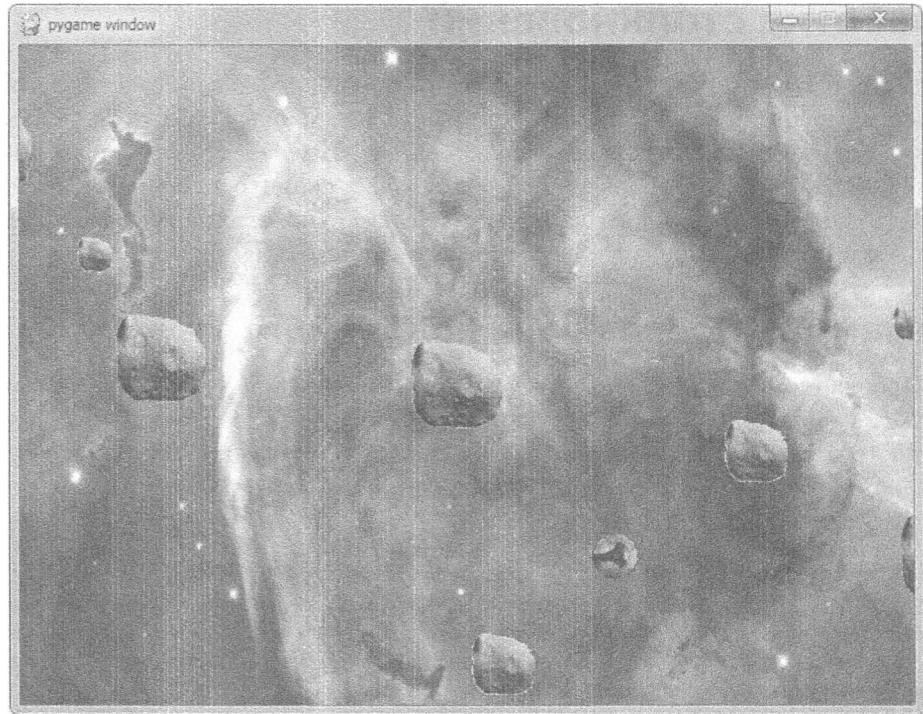


Рис. 12.13. Теперь ракеты, выпускаемые кораблем, разрушают астероиды.
Но будьте осторожны: астероид может врезаться в корабль

```
def die(self):
    """ Разрушает ракету. """
    self.destroy()
```

Когда у объекта `Missile` срабатывает метод `die()`, этот объект самопроизвольно разрушается.

Изменение метода `update()` объекта `Ship`

В конец метода `update()` в классе `Ship` я добавил следующий код:

```
# проверка на перекрытие с другими объектами
if self.overlapping_sprites:
    for sprite in self.overlapping_sprites:
        sprite.die()
    self.die()
```

Если возникает визуальное перекрытие корабля с какими-либо другими спрайтами, то и эти спрайты, и корабль будут разрушены методом `die()`. Заметьте, что тот же самый код дублируется в обновленном методе `update()` в классе `Missile`. Не забывайте, что любой повторяющийся код надо устранивать, или «консолидировать». В следующей версии игры этот и другие избыточные фрагменты кода будут реорганизованы.

Добавление метода die() объекту Ship

Это тот же самый метод, что и в классе Missile:

```
def die(self):
    """ Разрушает корабль. """
    self.destroy()
```

Когда у объекта Ship срабатывает метод die(), этот объект самопроизвольно разрушается.

Добавление константы в класс Asteroid

В классе Asteroid я создал одну новую константу:

```
SPAWN = 2
```

Значение константы SPAWN — количество новых астероидов, на которые распадается один взорванный.

Добавление метода die() объекту Asteroid

Метод die() в классе Asteroid реализован сравнительно сложно:

```
def die(self):
    """ Разрушает астероид. """
    # если размеры астероида крупные или средние, заменить его двумя более мелкими
    # астероидами
    if self.size != Asteroid.SMALL:
        for i in range(Asteroid.SPAWN):
            new_asteroid = Asteroid(x = self.x,
                                    y = self.y,
                                    size = self.size - 1)
            games.screen.add(new_asteroid)
    self.destroy()
```

Особенность, которую я здесь реализовал, заключается в том, что метод Asteroid.SPAWN позволяет создавать новые экземпляры класса Asteroid. Сначала выполняется проверка размеров астероида. Если это космическое тело крупного или среднего размера, то на его месте появляются два новых астероида — одним размером меньше. Вне зависимости от того, были ли они созданы, разрушенный астероид прекращает существовать и метод останавливается.

Добавление взрывов

В предыдущей версии игрок мог взрывать астероиды ракетами, но все это выглядело как-то несерьезно. Добавим в игру эффектные взрывы.

Программа «Прерванный полет — 7»

«Прерванный полет — 7» содержит новый класс на основе `games.Animation`, который я создал для изображения анимированных взрывов. Кроме того, выполнена реорганизация, в результате которой избыточный код был собран воедино. Игрок этого, конечно, не оценит, но все же новая структура кода — немаловажная вещь. Работу программы иллюстрирует рис. 12.14.



Рис. 12.14. Теперь все происходящие взрывы и столкновения сопровождаются яркими вспышками

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash07.py`.

Класс Wrapper

Начну с изменения, которое внешне не будет заметным: произведу от `games.Sprite` новый класс `Wrapper`.

Метод `update()`

В классе `Wrapper` я реализую метод `update()`, который автоматически вернет объект на противоположную границу, когда он пересечет одну из границ окна:

```
class Wrapper(games.Sprite):
    """ Огибатель. Спрайт, который, двигаясь, огибает графический экран. """
    def update(self):
        """ Переносит спрайт на противоположную сторону окна. """
        if self.top > games.screen.height:
            self.bottom = 0
        if self.bottom < 0:
            self.top = games.screen.height
        if self.left > games.screen.width:
            self.right = 0
        if self.right < 0:
            self.left = games.screen.width
```

Этот код вы уже видели, и неоднократно. Новизна нынешнего подхода в том, что, если произвести прочие классы игры от `Wrapper`, им не понадобятся собственные методы `update()`, а поведение объектов на экране останется прежним.

Метод `die()`

В этом классе недостает лишь метода `die()`, разрушающего объект:

```
def die(self):
    """ Разрушает объект. """
    self.destroy()
```

Класс `Collider`

Устранил и еще один элемент избыточности в коде. Выше было отмечено, что объекты `Ship` и `Missile` одинаково обрабатывают столкновения. Поэтому от `Wrapper` можно произвести новый класс `Collider`, объекты которого будут, выходя за пределы графического окна, возвращаться с противоположной стороны, а при столкновении с другими объектами — прекращать свое существование.

Метод `update()`

Вот метод `update()`, обрабатывающий столкновения:

```
class Collider(Wrapper):
    """ Погибатель. Огибатель, который может сталкиваться с другими объектами и гибнуть. """
    def update(self):
        """ Проверяет, нет ли спрайтов, визуально перекрывающихся с данным. """
        super(Collider, self).update()
        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()
```

Сначала в методе `update()` класса `Collider` язываю метод `update()` его родительского класса, то есть `Wrapper`, чтобы таким образом удержать объект внутри окна. Вслед за тем выполняется проверка на столкновения. Если объект перекрывается

с какими-либо другими на экране, то сначала ко всем этим соприкоснувшимся объектам применяется их метод `die()`, а затем разрушается и сам объект.

Метод `die()`

В этом классе я тоже реализую метод `die()`, потому что все объекты `Collider` гибнут одинаково — со взрывом и саморазрушением:

```
def die(self):
    """ Разрушает объект со взрывом. """
    new_explosion = Explosion(x = self.x, y = self.y)
    games.screen.add(new_explosion)
    self.destroy()
```

В этом методе, как видите, был создан объект `Explosion`. Экземпляры нового класса `Explosion`, с которым мы подробнее познакомимся чуть позже, — это анимированные взрывы.

Изменение класса `Asteroid`

Класс `Asteroid` я сделал производным от `Wrapper`:

```
class Asteroid(Wrapper):
```

Теперь `Asteroid` наследует от `Wrapper` метод `update()`, так что его собственный метод `update()` можно удалить. Уничтожение избыточного кода началось!

Еще одно маленькое изменение заключается в том, что в методе `die()` класса `Asteroid` я заменил последнюю строку `self.die()` такой:

```
super(Asteroid, self).die()
```

Теперь, если в метод `die()` класса `Wrapper` будет внесена правка, то `Asteroid` автоматически унаследует новое поведение.

Изменение класса `Ship`

Класс `Ship` я сделал производным от `Collider`:

```
class Ship(Collider):
```

А в метод `update()` добавил строку:

```
super(Ship, self).update()
```

Теперь можно удалить еще часть кода, ставшего ненужным. Поскольку метод `update()` в классе `Collider` обрабатывает столкновения, то можно лишний раз не поручать ту же задачу одноименному методу в классе `Ship`. Кроме того, метод `update()` «Погибателя» вызывает, в свою очередь, метод `update()` «Огibателя», а значит, одноименный метод в классе `Ship` больше не нуждается в коде, который отвечал за огибание окна спрайтом. Я также удалил метод `die()`: достаточно того варианта, который унаследован из `Collider`.

Изменение класса Missile

Изменив класс `Missile`, я сделал его производным от `Collider`:

```
class Missile(Collider):
```

В метод `update()` я добавил строку:

```
super(Missile, self).update()
```

Как и в случае с `Ship`, можно удалять избыточный код. Поскольку `update()` класса `Collider` справляется с обработкой столкновений, то обладающий той же функциональностью код можно удалить из `update()` класса `Missile`. По изложенным выше причинам класс перестал нуждаться также в коде, который отвечает за огибание окна (унаследован от `Wrapper`), и в методе `die()` (унаследован от `Collider`).

ПОДСКАЗКА

Чтобы лучше понять те изменения, которые я описываю, не стесняйтесь и изучите исходный код всех версий «Прерванного полета», доступный на сайте-помощнике по адресу www.courseptr.com/downloads.

Класс Explosion

Поскольку мне нужны анимированные взрывы, произведу класс `Explosion` от `games.Animation`.

```
class Explosion(games.Animation):
    """ Анимированный взрыв. """
    sound = games.load_sound("explosion.wav")
    images = ["explosion1.bmp",
              "explosion2.bmp",
              "explosion3.bmp",
              "explosion4.bmp",
              "explosion5.bmp",
              "explosion6.bmp",
              "explosion7.bmp",
              "explosion8.bmp",
              "explosion9.bmp"]
```

С переменной класса `sound` связан звуковой эффект взрыва. Другая переменная класса — `images` — ссылается на список из девяти строк — имен тех файлов, последовательность которых и образует анимированный взрыв.

Следующее по порядку — создать метод-конструктор `Explosion`.

```
def __init__(self, x, y):
    super(Explosion, self).__init__(images = Explosion.images,
                                    x = x, y = y,
                                    repeat_interval = 4, n_repeats = 1,
                                    is_collideable = False)
    Explosion.sound.play()
```

Этот конструктор принимает значения в параметры `x` и `y`, которыми описывается место взрыва на экране. Вызвав конструктор надкласса (`games.Animation`), я передаю эти значения его параметрам `x` и `y`, так что анимация появится именно там, где и ожидается. Кроме того, конструктору надкласса в параметр `images` будет передан список имен графических файлов — `Explosion.images`. Параметр же `n_repeats` я сделал равным 1, чтобы анимация воспроизводилась лишь один раз. Ради визуального правдоподобия взрыва параметр `repeat_interval`, характеризующий скорость анимации, получает значение 4. Наконец, параметру `is_collideable` я передал значение `False`, чтобы картинки, изображающие взрывы, не «сталкивались» с другими спрайтами, которые могут перекрываться с ними на экране. Осталось только воспроизвести звуковой эффект — `Explosion.sound.play()`.

ХИТРОСТЬ

Помните, что конструктору `games.Animation` можно передать в качестве «кадров» анимации или список имен файлов, или список уже загруженных объектов-изображений.

Уровни, ведение счета, музыкальная тема

Для того чтобы игра оставляла ощущение законченного продукта, надо в нее еще кое-что добавить. Напоследок я решил создать несколько уровней, чтобы игрок, разрушив все астероиды на экране, не чувствовал себя в безопасности и должен был противостоять еще более многочисленному налету. Кроме того, ради целостности пользовательских впечатлений от программы я создал функцию счета очков и сопроводил игру напряженной музыкальной темой.

Программа «Прерванный полет — 8»

Кроме системы уровней, игровой статистики и музыкальной темы, я также добавил в код несколько решений, менее очевидных с точки зрения игрока, но от того не менее важных. Окончательную версию программы, которая получилась в результате всех манипуляций, изображает рис. 12.15.

Код этой программы вы можете найти на сайте-помощнике (www.courseptr.com/downloads) в папке Chapter 12. Файл называется `astrocrash08.py`.

Импорт модуля `color`

Первое внесенное дополнение элементарно. Наряду с `games` из пакета `livewires` импортируется `color`:

```
from livewires import games, color
```

Модуль `color` нужен мне для того, чтобы сообщение `Game Over` отображалось на экране ярко-красными буквами.



Рис. 12.15. Нанесены последние штрихи, и теперь полет не прервется, пока звездолет игрока успешно лавирует между астероидами

Класс Game

Ближе к концу программы я создал класс `Game` — новый класс, объект которого представляет игру как таковую. На первый взгляд это странная идея, но если вдуматься, то окажется, что ничего неестественного в ней нет. Игра — точно такой же объект, у которого могут быть свои методы: например, `play()` — для начала игры, `advance()` — для перехода на следующий уровень, `end()` — для завершения игрового эпизода.

Принятое проектировочное решение позволяет сделать так, чтобы объекту `game` отправляли сообщения другие объекты. Например, если на текущем уровне остался один, последний астероид и игрок его взрывает, то этот астероид, прежде чем исчезнуть, может послать объекту `game` сообщение, которое инициирует переход на следующий уровень. Или же корабль, разрушаясь, может сообщить объекту `game` об окончании игры.

Разбираясь в коде класса `Game`, вы заметите, что сюда перекочевала значительная часть кода функции `main()`.

Метод `__init__()`

Первым делом в классе `Game` я объявляю конструктор:

```
class Game(object):
    """ Собственно игра. """
```

```

def __init__(self):
    """ Инициализирует объект Game. """
    # выбор начального игрового уровня
    self.level = 0
    # загрузка звука, сопровождающего переход на следующий уровень
    self.sound = games.load_sound("level.wav")
    # создание объекта, в котором будет храниться текущий счет
    self.score = games.Text(value = 0,
                           size = 30,
                           color = color.white,
                           top = 5,
                           right = games.screen.width - 10,
                           is_collideable = False)
    games.screen.add(self.score)
    # создание корабля, которым будет управлять игрок
    self.ship = Ship(game = self,
                     x = games.screen.width/2,
                     y = games.screen.height/2)
    games.screen.add(self.ship)

```

Атрибут `level` содержит номер текущего уровня игры. Переменная `sound` связана со звуковым эффектом, который сопровождает переход на следующий уровень. Атрибут `score` связан с текстовым объектом, который появится в правом верхнем углу экрана и будет отображать текущий счет. Свойство `is_collideable` этого объекта я установил равным `False`, чтобы корабль игрока не мог случайно взорваться, «врезавшись» в информационное табло. Наконец, атрибут `ship` — переменная, связанная с кораблем игрока.

Метод play()

Теперь я создам метод `play()`, ответственный за начало игры.

```

def play(self):
    """ Начинает игру. """
    # запуск музыкальной темы
    games.music.load("theme.mid")
    games.music.play(-1)
    # загрузка и назначение фоновой картинки
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image
    # переход к уровню 1
    self.advance()
    # начало игры
    games.screen.mainloop()

```

Этот метод загружает музыкальную тему и циклизует ее так, что она будет воспроизводиться неопределенно долго. Вслед за тем загружается изображение туманности, которое становится фоновой картинкой графического окна. Потом из кода метода вызывается другой метод объекта `Game` — `advance()`, который переводит игру на следующий уровень, в данном случае первый (как он работает, мы рассмотрим совсем скоро). Наконец, запуск всей игры осуществляется по вызову `games.screen.mainloop()`.

Метод advance()

Метод `advance()` отвечает за переход на очередной уровень игры. Он увеличивает на единицу значение атрибута `level`, создает очередную «волну» астероидов, краткое время отображает номер достигнутого уровня на экране и воспроизводит звук перехода к следующему уровню.

Реализация метода начинается несложно — с увеличения порядкового номера уровня:

```
def advance(self):
    """ Переводит игру на очередной уровень. """
    self.level += 1
```

Интересное и важное начинается дальше. Как создается волна астероидов? Надо иметь в виду, что на каждом уровне стартовое количество астероидов совпадает с порядковым номером этого уровня. На первом уровне, например, всего один астероид, на втором — два и т. д. Это в общем довольно просто; надо только проследить, чтобы ни один вновь создаваемый астероид не совпал по своему местоположению с кораблем игрока. Иначе, как только уровень начнется, корабль потерпит крушение.

```
# зарезервированное пространство вокруг корабля
BUFFER = 150
# создание новых астероидов
for i in range(self.level):
    # вычислим x и y, чтобы от корабля они отстояли минимум на BUFFER пикселях
    # сначала выберем минимальные отступы по горизонтали и вертикали
    x_min = random.randrange(BUFFER)
    y_min = BUFFER - x_min
    # исходя из этих минимумов, сгенерируем расстояния от корабля по горизонтали и вертикали
    x_distance = random.randrange(x_min, games.screen.width - x_min)
    y_distance = random.randrange(y_min, games.screen.height - y_min)
    # исходя из этих расстояний, вычислим экранные координаты
    x = self.ship.x + x_distance
    y = self.ship.y + y_distance
    # если необходимо, вернем объект внутрь окна
    x %= games.screen.width
    y %= games.screen.height
    # создадим астероид
    new_asteroid = Asteroid(game = self,
                            x = x, y = y,
                            size = Asteroid.LARGE)
    games.screen.add(new_asteroid)
```

Константа `BUFFER` определяет размеры безопасного пространства вокруг корабля. После ее задания запускается цикл, в каждой итерации которого наном удалении от корабля создается один астероид. Параметр `x_min` хранит наименьшее расстояние по оси абсцисс, на которое может отстоять астероид от корабля; `y_min` — такое же наименьшее расстояние по оси ординат. Разнообразие в эти параметры вносит модуль `random`, но в сумме `x_min` и `y_min` всегда остаются равными `BUFFER`.

Переменная `x_distance` – это расстояние от корабля до вновь созданного астероида по оси абсцисс. Это число, выбранное случайно, но не совсем: оно гарантирует, что астероид будет удален от корабля по меньшей мере на `x_min`. В свою очередь, `y_distance` – расстояние между кораблем и астероидом по оси ординат. И это тоже случайное число, гарантирующее, что от астероида до корабля будет не менее чем `y_min` пикселов.

Для того чтобы вычислить абсциссу x нового астероида, я сложил абсциссу корабля с величиной $x_distance$. Затем я проконтролировал, чтобы по горизонтали x не вылетала за пределы экрана: для этого находится остаток от целочисленного деления x на ширину окна. Ордината y нового астероида вычисляется подобным же образом: для этого достаточно сложить ординату корабля с величиной $y_distance$. Чтобы y тоже не вылетала за пределы экрана, ее новым значением становится остаток от ее целочисленного деления на высоту окна. Теперь наконец пара координат x и y описывает корректное размещение астероида.

Заметьте, что конструктор класса Asteroid принимает новый параметр game. Поскольку у любого из астероидов должна быть возможность вызвать один из методов объекта Game, как мы решили ранее, то всем им следует передавать ссылку на текущий объект Game. В качестве такой ссылки здесь выступает self.

После этого в методе `advance()` нужно отобразить порядковый номер очередного уровня и сопроводить переход звуковым эффектом:

```
# отображение номера уровня
level_message = games.Message(value = "Уровень " + str(self.level),
                               size = 40,
                               color = color.yellow,
                               x = games.screen.width/2,
                               y = games.screen.height/10,
                               lifetime = 3 * games.screen.fps,
                               is_collideable = False)
games.screen.add(level_message)
# звуковой эффект перехода (кроме 1-го уровня)
if self.level > 1:
    self.sound.play()
```

Метод end()

Метод end() выводит в центре графического экрана сакраментальные слова Game Over, набранные крупным красным шрифтом. Надпись продержится примерно 5 секунд. После этого игра будет окончена и окно закроется.

```

        after_death = games.screen.quit,
        is_collideable = False)
games.screen.add(end_message)

```

Добавление переменной и константы в класс Asteroid

В класс Asteroid я внес мелкую правку, связанную с добавлением уровней и ведением игровой статистики. Так, например, я создал константу класса POINTS:

```
POINTS = 30
```

Эта константа — исходная величина, на основе которой будет рассчитываться количество очков за уничтожение отдельного астероида. Эта величина будет тем больше, чем меньше астероид.

Для того чтобы можно было перейти к следующему уровню, программа должна знать, все ли астероиды на текущем уровне разрушены. Для этого надо следить за общей численностью астероидов, с чем успешно справится новая переменная класса — total, объявленная здесь же, в начале класса:

```
total = 0
```

Изменение метода-конструктора в классе Asteroid

В методе-конструкторе появилась строка, увеличивающая Asteroid.total на единицу:

```
Asteroid.total += 1
```

Чтобы астероид мог посылать сообщение объекту Game, я передам ему ссылку на требуемый объект. Принимать эту ссылку будет новый параметр в конструкторе класса Asteroid:

```
def __init__(self, game, x, y, size):
```

В параметр game будет передана переменная, связанная с объектом Game. Затем она же будет сохранена в одноименном атрибуте нового объекта Asteroid:

```
self.game = game
```

Итак, у всех вновь создаваемых экземпляров класса Asteroid теперь есть атрибут game — ссылка на текущую игру как таковую. С его помощью объект Asteroid может вызывать методы объекта Game, например advance().

Изменение метода die() объекта Asteroid

Метод die() в классе Asteroid я несколько расширил. Во-первых, появилась строка, уменьшающая Asteroid.total на единицу:

```
Asteroid.total -= 1
```

Затем исходя из размера астероида и значения константы Asteroid.POINTS вычисляется сумма очков, которую игрок получает за взрыв этого астероида (чем меньше размеры, тем выше номинал). Я контролирую также положение счета на экране — выравнивание по правому краю. Свойство right объекта score делается на 10 пикселов меньше координаты правого края окна.

```
self.game.score.value += int(Asteroid.POINTS / self.size)
self.game.score.right = games.screen.width - 10
```

При создании каждого из двух новых астероидов (осколков взорванного) надо передать им ссылку на объект Game. Для этого вызов конструктора класса Asteroid переписывается в виде:

```
new_asteroid = Asteroid(game = self.game,
```

Ближе к концу метода die() выполняется проверка Asteroid.total на совпадение с нулем: все ли астероиды разрушены? Если да, то последний при разрушении вызовет метод advance() объекта Game, тем самым переводя игру на следующий уровень уже с другим количеством астероидов.

```
# если больше астероидов не осталось, переходим на следующий уровень
if Asteroid.total == 0:
    self.game.advance()
```

Добавление константы в класс Ship

В классе Ship я сделал несколько изменений. Во-первых, была создана константа VELOCITY_MAX, ограничивающая максимальную скорость корабля:

```
VELOCITY_MAX = 3
```

Изменение метода-конструктора в классе Ship

Как и объект Asteroid, экземпляр Ship должен иметь доступ к объекту Game, чтобы вызывать его методы. Без какого-либо отличия от Asteroid я изменяю конструктор класса Ship:

```
def __init__(self, game, x, y):
```

Новый параметр game принимает ссылку на объект Game, которая затем сохраняется в атрибуте объекта Ship:

```
self.game = game
```

Таким образом, у каждого объекта Ship теперь будет атрибут game — ссылка на игру как таковую. С его помощью экземпляр Ship может вызывать объектные методы Game, например end().

Изменение метода update() объекта Ship

Метод update() в классе Ship претерпел следующее изменение: скорость корабля по каждому из двух векторов, то есть dx и dy, ограничена с помощью MAX_VELOCITY, константы класса:

```
# ограничение горизонтальной и вертикальной скорости
self.dx = min(max(self.dx, -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
self.dy = min(max(self.dy, -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
```

Этот код гарантирует, что `dx` и `dy` никогда не будут меньше чем `-Ship.VELOCITY_MAX` и больше чем `Ship.VELOCITY_MAX`. Для достижения такого эффекта применены функции `min()` и `max()`. Функция `min()` возвращает наименьшее из двух чисел, а `max()` — наибольшее. Ограничить скорость надо, в частности, для того, чтобы корабль не столкнулся, чрезмерно разогнавшись, с собственными ракетами, лежащими впереди.

Добавление метода `die()` объекту `Ship`

После разрушения корабля игрока прерывается полет, а вместе с ним и игра. В классе `Ship` я создал метод `die()`, который, в свою очередь, вызывает метод `end()` объекта `Game`, завершающий игру.

```
def die(self):
    """ Разрушает корабль и завершает игру. """
    self.game.end()
    super(Ship, self).die()
```

Функция `main()`

Теперь, когда реализован класс `Game`, функция `main()` стала очень короткой. Весь ее смысл в том, чтобы создать объект `Game` и вызвать его метод `play()`, который запускает игровой процесс.

```
def main():
    astrocrash = Game()
    astrocrash.play()
# поехали!
main()
```

Резюме

В этой главе вы глубже познакомились с мультимедийным программированием, освоив технику применения звуков, музыки и анимации. Вы узнали, как загружать, воспроизводить и останавливать звуковые и музыкальные файлы, как создавать анимированные картинки. Кроме того, вы ознакомились с технологией разработки больших приложений, которая заключается в том, чтобы писать последовательно все более сложные работающие версии конечного продукта. Вы увидели, как, решая одну задачу за другой, проложить путь от наброска к окончательной реализации. Наконец, все эти новые сведения и приемы пригодились вам в разработке динамичной экшен-игры со звуковыми эффектами, анимацией и собственной музыкальной темой.

Задачи

- Доработайте «Прерванный полет» таким образом, чтобы кроме астероидов в игре появился какой-нибудь другой тип опасных космических обломков. Необходимо, чтобы какое-либо качество отличало эту новую разновидность от астероидов: например, для разрушения обломков такого типа может понадобиться два ракетных залпа вместо одного.
- Напишите вариант детской игры «Саймон говорит — запоминайт» (Simon Says). В ней игрок должен с клавиатуры дублировать все удлиняющуюся случайную последовательность звуков и цветов, которые воспроизводит и демонстрирует компьютер.
- Напишите свою версию какой-либо другой знаменитой видеоигры, например «Космических захватчиков» или «Пакмана».
- Создайте собственную программистскую задачу. И помните: главное — никогда не переставать стремиться к новому и неизвестному.

Приложение А

Сайт-помощник

Сайт-помощник к этой книге доступен по адресу www.courseptr.com/downloads. Там вы можете скачать исходный код всех программ, вспомогательные файлы и пакеты, описанные в книге.

Архивы. Для скачивания доступно два архива:

- `py3e_source.zip` — содержит исходный код всех законченных программ, которые представлены в этой книге, и вспомогательные файлы к ним;
- `py3e_software` — включает в себя файлы всех программных пакетов, упомянутых в книге, в том числе установочный файл Python 3.1.1 для Windows.

В табл. А.1 описано содержимое архива `py3e_source.zip`, а в табл. А.2 — содержимое архива `py3e_software.zip`.

Таблица А.1. Содержимое архива `py3e_source.zip`

Название папки	Описание
<code>chapter01</code>	Исходный код к главе 1
<code>chapter02</code>	Исходный код к главе 2
<code>chapter03</code>	Исходный код к главе 3
<code>chapter04</code>	Исходный код к главе 4
<code>chapter05</code>	Исходный код к главе 5
<code>chapter06</code>	Исходный код к главе 6
<code>chapter07</code>	Исходный код и файлы данных к главе 7
<code>chapter08</code>	Исходный код к главе 8
<code>chapter09</code>	Исходный код к главе 9
<code>chapter10</code>	Исходный код к главе 10 и соответствующие пакетные файлы для запуска программ
<code>chapter11</code>	Исходный код к главе 11, пакетные файлы, графика и звук
<code>chapter12</code>	Исходный код к главе 12, пакетные файлы, графика и звук

Таблица А.2. Содержимое архива `py3e_software.zip`

Название папки	Описание
<code>python</code>	Дистрибутив Python 3.1.1 для установки под Windows
<code>pygame</code>	Пакет <code>pygame</code> версии 1.9.1, совместимый с Python 3.1.x под Windows
<code>livewires</code>	Пакет (игровой движок) <code>livewires</code>

СОВЕТ

Установочный файл `pygame` для Windows в архиве `py3e_software.zip` совместим с Python 3.1.x, то есть с любым вариантом Python 3.1 (от 3.1.0 до 3.1.9).

Приложение В

Справка по пакету livewires

В этом приложении собрано почти все то, что вы хотели узнать о модифицированном пакете `livewires`, но о чём боялись спросить. Кое-что ради простоты я не излагаю; самую свежую версию документации всегда можно найти в исходном коде модуля `livewires` на его сайте.

Пакет `livewires`

Пакет `livewires` — это набор модулей, предназначенных для создания игр с графикой, звуком, анимацией. Два из этих модулей названы в табл. В.1. Если вы хотите пользоваться пакетом `livewires`, необходимо, чтобы был установлен мультимедийный пакет `pygame`.

Таблица В.1. Модули пакета `livewires`

Модуль	Описание
<code>games</code>	Содержит функции и классы, облегчающие разработку игр
<code>color</code>	Содержит набор цветовых констант

Классы модуля `games`

В `games` содержится несколько классов и функций, предназначенных для программирования игр. Классы перечислены в табл. В.2.

Таблица В.2. Классы модуля `games`

Класс	Описание
<code>Screen</code>	Объект этого класса представляет собой графический экран, или графическое окно
<code>Sprite</code>	Объект этого класса оснащен картинкой и может отображаться на графическом экране
<code>Text</code>	Объект этого класса представляет собой текст на графическом экране. <code>Text</code> — подкласс <code>Sprite</code>
<code>Message</code>	Объект этого класса представляет собой сообщение, которое появляется на графическом экране и через определенное время исчезает. <code>Message</code> — подкласс <code>Text</code>

Класс	Описание
Animation	Объект этого класса представляет собой несколько картинок, последовательно выводимых на графический экран. Animation — подкласс Sprite
Mouse	Объект этого класса обеспечивает доступ к мыши
Keyboard	Объект этого класса обеспечивает доступ к клавиатуре
Music	Объект этого класса обеспечивает доступ к музыкальному каналу

Класс Screen

Объект Screen представляет собой графический экран. Функция `games.init()` создает объект `screen` класса `Screen`; принято пользоваться этим объектом вместо того, чтобы создавать свой собственный экземпляр класса. В табл. В.3 описываются свойства объектов `Screen`, а в табл. В.4 — методы.

Таблица В.3. Свойства объектов Screen

Свойство	Описание
<code>width</code>	Ширина экрана
<code>height</code>	Высота экрана
<code>fps</code>	Количество обновлений экрана в секунду
<code>background</code>	Фоновая картинка экрана
<code>all_objects</code>	Список всех спрайтов, которые есть на экране
<code>event_grab</code>	Булево свойство, определяющее, перенаправлен ли ввод в графическое окно. Если да, то принимает <code>True</code> , если нет — <code>False</code>

Таблица В.4. Методы объекта Screen

Метод	Описание
<code>get_width()</code>	Возвращает ширину экрана
<code>get_height()</code>	Возвращает высоту экрана
<code>get_fps()</code>	Возвращает количество обновлений экрана в секунду
<code>get_background()</code>	Возвращает фоновую картинку экрана
<code>set_background(new_background)</code>	Назначает <code>new_background</code> в качестве фоновой картинки экрана
<code>get_all_objects()</code>	Возвращает список всех спрайтов, которые есть на экране
<code>get_event_grab()</code>	Сообщает, перенаправлен ли ввод в графическое окно. Если да, то принимает <code>True</code> , если нет — <code>False</code>
<code>set_event_grab(new_status)</code>	Делает статус перенаправления ввода в графическое окно равным <code>new_status</code> . <code>True</code> — ввод будет перенаправлен, <code>False</code> — нет
<code>add(sprite)</code>	Добавляет на графический экран спрайт — объект <code>Sprite</code> или одного из его подклассов
<code>remove(sprite)</code>	Удаляет с графического экрана спрайт — объект <code>Sprite</code> или одного из его подклассов
<code>clear()</code>	Удаляет с графического экрана все спрайты
<code>mainloop()</code>	Запускает основной цикл графического экрана
<code>quit()</code>	Закрывает графическое окно

Класс Sprite

Объекты Sprite оснащены картинками и могут отображаться на графическом экране. В табл. В.5 описываются свойства объектов Sprite, а в табл. В.6 — их методы.

Таблица В.5. Свойства объекта Sprite

Свойство	Описание
image	Картинка спрайта — объект-изображение
width	Ширина картинки
height	Высота картинки
angle	Угол наклона в градусах
x	Абсцисса
y	Ордината
position	Положение спрайта на экране. Двухэлементный кортеж (x, y)
top	Ордината верхней границы спрайта
bottom	Ордината нижней границы спрайта
left	Абсцисса левой границы спрайта
right	Абсцисса правой границы спрайта
dx	Горизонтальная скорость
dy	Вертикальная скорость
velocity	Скорость движения спрайта. Двухэлементный кортеж (dx, dy)
overlapping_sprites	Список других объектов, перекрывающихся с данным
is_collideable	Булево свойство, определяющее, может ли спрайт сталкиваться с другими объектами. Если True, то столкновения будут регистрироваться, если False — нет
interval	Определяет интервал между вызовами метода tick() объекта

Таблица В.6. Методы объекта Sprite

Метод	Описание
<code>__init__(image [, angle] [, x] [, y] [, top] [, bottom] [, left] [, right] [, dx] [, dy] [, interval] [, is_collideable])</code>	Инициализирует новый спрайт. У параметра image нет значения по умолчанию, поэтому его обязательно передавать. Параметры angle, x, y, dx и dy по умолчанию равны 0. Параметры top, bottom, left и right по умолчанию имеют значение None. Если не передать им никаких значений, то при инициализации соответствующие свойства останутся неопределенными. Параметр interval по умолчанию равен 1, а is_collideable — True
<code>get_image()</code>	Возвращает объект-изображение, связанный со спрайтом
<code>set_image(new_image)</code>	Назначает new_image в качестве картинки спрайта
<code>get_height()</code>	Возвращает высоту картинки спрайта
<code>get_width()</code>	Возвращает ширину картинки спрайта
<code>get_angle()</code>	Возвращает текущий угол наклона спрайта в градусах
<code>set_angle(new_angle)</code>	Устанавливает угол наклона равным new_angle
<code>get_x()</code>	Возвращает абсциссу спрайта
<code>set_x(new_x)</code>	Устанавливает абсциссу спрайта равной new_x
<code>get_y()</code>	Возвращает ординату объекта

Метод	Описание
set_y(new_y)	Устанавливает ординату спрайта равной new_y
get_position()	Возвращает положение спрайта в виде двухэлементного кортежа
set_position(new_position)	Устанавливает спрайт в позицию, заданную двухэлементным кортежем new_position
get_top()	Возвращает ординату верхней кромки спрайта
set_top(new_top)	Устанавливает ординату верхней кромки спрайта равной new_top
get_bottom()	Возвращает ординату нижней кромки спрайта
set_bottom(new_bottom)	Устанавливает ординату нижней кромки спрайта равной new_bottom
get_left()	Возвращает абсциссу левой кромки объекта
set_left(new_left)	Устанавливает абсциссу левой кромки спрайта равной new_left
get_right()	Возвращает абсциссу правой кромки объекта
set_right(new_right)	Устанавливает абсциссу правой кромки спрайта равной new_right
get_dx()	Возвращает горизонтальную скорость спрайта
set_dx(new_dx)	Устанавливает горизонтальную скорость спрайта равной new_dx
get_dy()	Возвращает вертикальную скорость спрайта
set_dy(new_dy)	Устанавливает вертикальную скорость спрайта равной new_dy
get_velocity()	Возвращает скорость спрайта в виде двухэлементного кортежа
set_velocity(new_velocity)	Устанавливает скорость спрайта равной двухэлементному кортежу new_velocity
get_overlapping_sprites()	Возвращает список всех спрайтов, которые способны сталкиваться и при этом перекрываются с данным объектом
overlaps(other)	Возвращает True, если объект визуально перекрывается с other, и False в противном случае. Возвращает False, если хотя бы у одного из объектов атрибут is_collideable равен False
get_is_collideable()	Сообщает, способен ли данный спрайт сталкиваться. True значит, что его столкновения с другими объектами регистрируются, False — что не регистрируются
set_is_collideable(new_status)	Делает статус «столкновимости» спрайта равным new_status. Если последний равен True, то столкновения с другими объектами будут регистрироваться, если False — не будут
get_interval()	Возвращает интервал между вызовами метода tick() спрайта
set_interval(new_interval)	Устанавливает интервал между вызовами метода tick() спрайта равным new_interval
update()	Обновляет спрайт. По умолчанию ничего не делает; вызывается при каждой итерации mainloop(). В классе, производном от Sprite, может быть переопределен
tick()	Вызывается через каждые interval итераций mainloop(). По умолчанию ничего не делает. В классе, производном от Sprite, может быть переопределен
destroy()	Удаляет спрайт с экрана

Класс Text

Text — подкласс Sprite. Объект Text представляет собой текст на графическом экране. Конечно, Text наследует атрибуты, свойства и методы от Sprite. В табл. В.7 перечислены собственные свойства Text, а в табл. В.8 — его собственные методы.

Таблица В.7. Свойства объекта Text

Свойство	Описание
value	Значение, отображаемое как текст
size	Размер текста
color	Цвет текста. Может быть равен одной из констант, заданных в модуле color

На основе text, size и color конструктор класса создает графический объект — текст на экране.

Таблица В.8. Методы объекта Text

Метод	Описание
<code>__init__(value, size, color [, angle] [, x] [, y] [, top] [, bottom] [, left] [, right] [, dx] [, dy] [, interval] [, is_collideable])</code>	Инициализирует новый объект. Параметр value — значение, отображаемое как текст, size — размер, color — цвет текста. Параметры angle, x, y, dx и dy по умолчанию равны 0. Параметры top, bottom, left и right по умолчанию имеют значение None. Если не передать им никаких значений, то при инициализации соответствующие свойства останутся неопределенными. Параметр interval по умолчанию равен 1, а is_collideable — True
<code>get_value()</code>	Возвращает значение, отображаемое как текст
<code>set_value(new_value)</code>	Устанавливает new_value в качестве нового отображаемого значения
<code>get_size()</code>	Возвращает размер текста
<code>set_size(new_size)</code>	Устанавливает new_size в качестве размера текста
<code>get_color()</code>	Возвращает цвет текста
<code>set_color(new_color)</code>	Устанавливает new_color в качестве цвета текста. Можно приравнять цвет к одной из констант, заданных в модуле color

Класс Message

Message — подкласс Text. Объект Message представляет собой сообщение, которое появляется на графическом экране и через определенное время исчезает. Message может также назначать событие, которое должно произойти после исчезновения текста с экрана.

Message наследует атрибуты, свойства и методы от класса Text. В добавок к ним у объекта Message есть новый атрибут after_death. Этот атрибут может ссылаться на код (например, имя функции или метода), который следует выполнить после исчезновения сообщения. Значение по умолчанию — None.

Message объявляет новый метод `__init__(): __init__(value, size, color [, angle] [, x] [, y] [, top] [, bottom] [, left] [, right] [, dx] [, dy] [, lifetime] [, is_collideable] [, after_death])`. Этот метод инициализирует новый объект. Параметр value — значение, отображаемое как текст, size — размер, color — цвет текста. Параметры angle, x, y, dx и dy по умолчанию равны 0. Параметры top, bottom, left и right по умолчанию имеют значение None. Если не передать им никаких значений, то при инициализации соответствующие свойства останутся неопределенными. Параметр lifetime определяет продолжительность пребывания текста на экране — сколько

итераций `mainloop()` должно пройти, прежде чем сообщение исчезнет. Если передать этому параметру 0, то объект вообще не исчезнет с экрана. По умолчанию `lifetime` имеет значение 0, а `is_collideable` — True. Параметр `after_death` по умолчанию равен None.

СОВЕТ

Атрибуты или свойства `lifetime` объекта `Message` не имеют, поэтому значение, переданное параметру `lifetime`, сохраняется в прежнем свойстве `interval`.

Класс Animation

`Animation` — подкласс `Sprite`. Объект `Animation` представляет собой некоторое количество картинок, последовательно выводимых на графический экран. `Animation` наследует атрибуты, свойства и методы от `Sprite` и объявляет несколько собственных атрибутов, которые описываются в табл. В.9.

Таблица В.9. Атрибуты объекта Animation

Атрибут	Описание
<code>images</code>	Список объектов-изображений
<code>n_repeats</code>	Число, которое определяет, сколько раз должен повториться цикл прокрутки анимации. Если 0, то анимация будет воспроизводиться не переставая. По умолчанию — 0

`Animation` объявляет новый метод `_init_(): _init_(images [. angle] [. x] [. y] [. top] [. bottom] [. left] [. right] [. dx] [. dy] [. repeat_interval] [. n_repeats] [. is_collideable])`. Этот метод инициализирует новый объект. Параметру `images` можно передать как список объектов-изображений, так и список имен файлов, которые еще предстоит загрузить. Параметры `angle`, `x`, `y`, `dx` и `dy` по умолчанию равны 0. Параметры `top`, `bottom`, `left` и `right` по умолчанию имеют значение `None`. Если не передать им никаких значений, то при инициализации соответствующие свойства останутся неопределенными. Параметр `repeat_interval` определяет интервал между вызовами метода `tick()` объекта, а значит, и скорость анимации. По умолчанию он равен 1. По умолчанию `n_repeats` имеет значение 0, а `is_collideable` — True.

СОВЕТ

Атрибуты или свойства `repeat_interval` объект `Animation` не имеет, поэтому значение, переданное параметру `repeat_interval`, сохраняется в прежнем свойстве `interval`.

Класс Mouse

Объект `Mouse` обеспечивает доступ к мыши. Функция `games.init()` создает объект `mouse` класса `Mouse`, следящий за позицией указателя и регистрирующий нажатия клавиш. Принято пользоваться этим объектом вместо того, чтобы создавать свой собственный экземпляр класса. Свойства `Mouse` описываются в табл. В.10, а методы — в табл. В.11.

Таблица В.10. Свойства объекта Mouse

Свойство	Описание
x	Абсцисса указателя мыши
y	Ордината указателя мыши
position	Положение указателя мыши. Двухэлементный кортеж (x, y)
is_visible	Булево свойство, определяющее, видим ли указатель мыши на экране. Если True — видим, если False — нет. По умолчанию True

Таблица В.11. Методы объекта Mouse

Метод	Описание
get_x()	Возвращает абсциссу указателя мыши
set_x(new_x)	Устанавливает абсциссу указателя мыши равной new_x
get_y()	Возвращает ординату указателя мыши
set_y(new_y)	Устанавливает ординату указателя мыши равной new_y
get_position()	Возвращает положение указателя мыши в виде двухэлементного кортежа
set_position(new_position)	Устанавливает указатель мыши в позицию, заданную двухэлементным кортежем new_position
set_is_visible(new_visibility)	Настраивает видимость указателя мыши. Если значение new_visibility равно True, указатель видим, а если False — невидим
is_pressed(button_number)	Проверяет, нажата ли одна из кнопок мыши. Возвращает True, если кнопка номер button_number удерживается пользователем, и False в противном случае

Класс Keyboard

Объект Keyboard обеспечивает доступ к клавиатуре. Функция games.init() создает объект keyboard класса Keyboard, регистрирующий нажатия клавиш. Принято пользоваться этим объектом вместо того, чтобы создавать собственный экземпляр класса.

В классе реализован единственный метод is_pressed(key), который возвращает True, если проверяемая клавиша в данный момент нажата пользователем, и False — в противном случае. Как аргументы этого метода можно использовать константы, объявленные в модуле games. Их полный перечень приводится далее в этом приложении, в табл. В.15.

Класс Music

Объект Music обеспечивает доступ к единственному музыкальному каналу, позволяя таким образом загружать, воспроизводить и останавливать музыкальные файлы. Принято пользоваться этим объектом вместо того, чтобы создавать свой собственный экземпляр класса. Музыкальный канал допускает много разных форматов, в том числе WAV, MP3, OGG и MIDI. Методы Music охарактеризованы в табл. В.12.

Таблица В.12. Методы объекта Music

Метод	Описание
load(filename)	Загружает файл filename в музыкальный канал. Если какой-либо файл уже загружен, он будет заменен новым
play([loop])	Воспроизводит музыку loop+1 раз. Если передать значение -1, то музыка будет играть не переставая. По умолчанию передается значение 0
fadeout(millisec)	Постепенно, в течение millisec миллисекунд приглушает играющую музыку
stop()	Останавливает воспроизведение музыки по музыкальному каналу

ФУНКЦИИ МОДУЛЯ games

В модуле games есть также функции для работы с изображениями и звуком. Они описаны в табл. В.13.

Таблица В.13. Функции games

Функция	Описание
init([screen_width,] [screen_height,] [fps])	Инициализирует графический экран шириной screen_width и высотой screen_height, вид которого обновляется fps раз в секунду. Объект screen — экземпляр класса games.Screen, обеспечивающего доступ к графике. Объект mouse — экземпляр класса games.Mouse, ответственного за доступ к мыши. Объект keyboard — экземпляр класса games.Keyboard, ответственного за доступ к клавиатуре. Наконец, объект music, экземпляр класса games.Music, дает доступ к единственному каналу воспроизведения музыки
load_image(filename [, transparent])	Возвращает объект-изображение, загруженный из файла filename. Если параметр transparent равен True (что задано по умолчанию), у картинки будет прозрачный фон
scale_image(image, x_scale [, y_scale])	Возвращает новый объект-изображение, который представляет собой картинку image, растянутую по горизонтали в x_scale раз, а по вертикали в y_scale раз. Если не передать параметру y_scale никакого значения, то масштаб x_scale будет применен к обеим координатным осям. Исходное изображение остается прежним
load_sound(filename)	Возвращает звуковой объект, загруженный из WAV-файла filename

У звукового объекта, возвращаемого методом load_sound(), доступно несколько собственных методов. Они перечислены в табл. В.14.

Таблица В.14. Методы объекта Sound

Метод	Описание
play([loop])	Воспроизводит звук loop+1 раз. Если передать значение -1, то звук будет играть не переставая. По умолчанию передается значение 0
fadeout(millisec)	Постепенно, в течение millisec миллисекунд приглушает звук
stop()	Останавливает воспроизведение звука по всем каналам

Константы модуля games

В модуле colog доступно некоторое количество констант, соответствующих клавишам на клавиатуре. Их полный список приводится в табл. В.15.

Таблица В.15. Клавиатурные константы модуля games

Константа	Клавиша
K_RETURN	Return
K_PAUSE	Pause
K_ESCAPE	Esc
K_SPACE	Пробел
K_EXCLAIM	!
K_QUOTEDBL	"
K_HASH	#
K_DOLLAR	\$
K_AMPERSAND	&
K_QUOTE	'
K_LEFTPAREN	(
K_RIGHTPAREN)
K_ASTERISK	*
K_PLUS	+
K_COMMMA	,
K_MINUS	-
K_PERIOD	.
K_SLASH	/
K_0	0
K_1	1
K_2	2
K_3	3
K_4	4
K_5	5
K_6	6
K_7	7
K_8	8
K_9	9
K_COLON	:
K_SEMICOLON	;
K_LESS	<
K_EQUALS	=
K_GREATER	>
K_QUESTION	?
K_AT	@
K_LEFTBRACKET	[
K_BACKSLASH	\

Константа	Клавиша
K_RIGHTBRACKET]
K_CARET	Возврат каретки
K_UNDERSCORE	=
K_a	À
K_b	Б
K_c	С
K_d	Д
K_e	Е
K_f	Ғ
K_g	҂
K_h	Ҥ
K_i	Ӣ
K_j	Ҋ
K_k	Ҝ
K_l	Ӆ
K_m	Ӎ
K_n	Ң
K_o	Ӯ
K_p	Ӱ
K_q	Ӳ
K_r	Ӯ
K_s	Ӯ
K_t	Ӯ
K_u	Ӯ
K_v	Ӯ
K_w	Ӯ
K_x	Ӯ
K_y	Ӯ
K_z	Ӯ
K_DELETE	Delete
K_KP0	Keypad ¹ 0
K_KP1	Keypad 1
K_KP2	Keypad 2
K_KP3	Keypad 3
K_KP4	Keypad 4
K_KP5	Keypad 5
K_KP6	Keypad 6
K_KP7	Keypad 7
K_KP8	Keypad 8
K_KP9	Keypad 9

Продолжение в

¹ Имеются в виду клавиши правой цифровой клавиатуры. — Примеч. пер.

Таблица В.15 (продолжение)

Константа	Клавиша
K_KP_PERIOD	Keypad Period
K_KP_DIVIDE	Keypad Divide
K_KP_MULTIPLY	Keypad Multiply
K_KP_MINUS	Keypad Minus
K_KP_PLUS	Keypad Plus
K_KP_ENTER	Keypad Enter
K_KP_EQUALS	Keypad Equals
K_UP	Up Arrow
K_DOWN	Down Arrow
K_RIGHT	Right Arrow
K_LEFT	Left Arrow
K_INSERT	Insert
K_HOME	Home
K_END	End
K_PAGEUP	Page Up
K_PAGEDOWN	Page Down
K_F1	F1
K_F2	F2
K_F3	F3
K_F4	F4
K_F5	F5
K_F6	F6
K_F7	F7
K_F8	F8
K_F9	F9
K_F10	F10
K_F11	F11
K_F12	F12
K_NUMLOCK	Num Lock
K_CAPSLOCK	Caps Lock
K_SCROLLLOCK	Scroll Lock
K_RSHIFT	Правый Shift
K_LSHIFT	Левый Shift
K_RCTRL	Правый Ctrl
K_LCTRL	Левый Ctrl
K_RALT	Правый Alt
K_LALT	Левый Alt
K_LSUPER	Левая клавиша Windows
K_RSUPER	Правая клавиша Windows
K_HELP	Help
K_PRINT	Print Screen
K_BREAK	Break

Константы модуля color

В модуле `color` доступно несколько констант, которыми можно пользоваться при создании игр как цветообозначениями:

- `red` — красный;
- `green` — зеленый;
- `blue` — голубой;
- `black` — черный;
- `white` — белый;
- `dark_red` — багровый;
- `dark_green` — темно-зеленый;
- `dark_blue` — насыщенно-синий;
- `dark_gray` — темно-серый;
- `gray` — серый;
- `light_gray` — светло-серый;
- `yellow` — желтый;
- `brown` — коричневый;
- `pink` — розовый;
- `purple` — фиолетовый.

Алфавитный указатель

- __additional_cards(), метод 274
(escape-символ одиночной кавычки) 50, 51
__pass_time(), метод 241
@property, декоратор 238
Python
 независимость от платформы 33
 (звук системного динамика) 50, 51
 (обратный слеш) 49, 52
 (оператор вычитания) 55
 // (оператор деления с остатком) 55
 / (оператор истинного деления) 55
 % (оператор нахождения остатка от деления) 55
 * (оператор повторения) 53
 + (оператор сложения) 55
 > (оператор сравнения «больше») 78
 >= (оператор сравнения «больше или равно») 78
 < (оператор сравнения «меньше») 78
 <= (оператор сравнения «меньше или равно») 78
!= (оператор сравнения «не равно») 77
== (оператор сравнения «равно») 77
+ (оператор склеивания) 52, 65, 129
* (оператор умножения) 55
/ (правый слеш) 214
(символ комментария) 41
(символ новой строки) 49, 51, 199, 202
_ (символ подчеркивания) 234
- A**
- ab 205
ab+ 205
add(sprite), метод объекта screen 313, 389
- add(), метод
программа «Карты» 251
спрайты 319
advance(), метод 381
after_death, атрибут сообщения 325
all_objects, свойство объекта screen 389
and, логический оператор 96
angle, свойство 320, 349, 390
Animation, класс 351, 389, 393
append(), списочный метод 142, 144
Application
 класс, объявление 288
 объект, создание 289
ASCII-графика 48
ask_number(), функция 184, 185, 264, 265
ask_yes_no(), функция 172, 184, 185, 264
Asteroid.SPAWN, метод 373
Asteroid, класс 360
 die(), метод 373, 383
 __init__(), метод 360
 update(), метод 360
 изменение 376
 метод-конструктор 383
 общая характеристика 373
 переменная и константа 383
- B**
- background, свойство 313, 389
BJ_Card, класс 267, 269
BJ_Dealer, класс 268, 272
BJ_Deck, класс 268, 270
BJ_Game, класс
 __additional_cards(), метод 274
 __init__(), метод 273

play(), метод 274
 still_playing, свойство 273
 общая характеристика 268, 273
BJ_Hand, класс 268, 270
BJ_Player, класс 268, 272
BooleanVar, объект 300
bottom, свойство 320, 390
bust(), метод 273

C

capitalize(), метод 63
cards, модуль
 импорт 269
Card, класс
 объект Hand как сочетание его экземпляров 251
 применение 251
 создание 250

checkCatch(), метод 338
checkDrop(), метод 341
Chef, класс
 check_drop(), метод 341
 __init__(), метод 340
 update(), метод 340
 общая характеристика 340
clear(), метод объекта screen 251, 313, 389
close(), функция 198, 203
Collider, класс

die(), метод 376
 update(), метод 375
 общая характеристика 375
color, модуль

импорт 321, 324, 378
 константы 399
 общая характеристика 388
color, свойство 322, 392
colspan, параметр 293
column, параметр 293
command, параметр 291
computerMove(), функция 184, 188

configure(), метод 287
congrat_winner(), функция 184, 191
cos(), функция 365
count(), списочный метод 144
createWidgets(), метод 289, 291, 304
Critter, класс 241
 eat(), метод 242
mood, свойство 241
 __pass_time(), метод 241
play(), метод 242
talk(), метод 242
 метод-конструктор 241
 создание 242
C#, язык 32

D

deal(), метод 257
Deck, класс 253
delete(), метод 296
del, служебное слово 140
destroy(), метод 320, 339, 391
displayBoard(), функция 184, 186
displayInstruct(), функция 184, 185
display(), функция 170
dump, функция консервации 206
dx, свойство 320, 390
dy, свойство 320, 390

E

eat(), метод 242
elif, условие 80
else, условие 79, 83, 212
endGame(), метод 339
end(), метод 382
end, параметр 47
Escape-последовательности
 вставка кавычек 50
 вставка пустой строки 49
 вставка табуляционного отступа 49

вывод обратного слеша 49
 звук системного динамика 50
 общая характеристика 48
event_grab, свойство 313, 389
except, оператор 210, 212
Explosion, класс 377

F

fadeout(), метод 395
False, значение 89
flip_first_card(), метод 273
float(), функция 68
fps, свойство объекта screen 313, 389

G

Game Over 2.0, программа
 общая характеристика 44
Game Over, программа
 в интерактивном режиме 35
print(), функция 36
 генерирование ошибки 37
 написание программы 36
 подсветка синтаксиса 38
 терминология 37
 в сценарном режиме 38
 запуск программы 39
 написание программы 38
 сохранение программы 38
 вывод строки на экран 42
 комментарии 41
 общая характеристика 30
 ожидание пользователя 42
 пустые строки 41
games.init(), метод 336
games.init(), функция 312
games.load_image(), функция 314
games, модуль
 Animation, класс 393
 Keyboard, класс 394
 Message, класс 392
 Mouse, класс 393

Music, класс 394
 Screen, класс 389
 Sprite, класс 390
 Text, класс 391
 импорт 269, 311
 константы 396
 общая характеристика 388
 функции 395
Game, класс 379
 advance(), метод 381
 end(), метод 382
__init__(), метод 379
 play(), метод 380
 geometry(), метод 283
 get_all_objects(), метод объекта screen 389
 get_angle(), метод 390
 get_background(), метод объекта screen 389
 get_bottom(), метод 391
 get_color(), метод 392
 get_dx(), метод 391
 get_dy(), метод 391
 get_event_grab(), метод объекта screen 389
 get_fps(), метод объекта screen 389
 get_height(), метод 390
 get_image(), метод 390
 get_interval(), метод 391
 get_is_collideable(), метод 391
 get_left(), метод 391
 get_overlapping_sprites(), метод 391
 get_position(), метод 391, 394
 get_right(), метод 391
 get_size(), метод 392
 get_top(), метод 391
 get_value(), метод 392
 get_velocity(), метод 391
 get_width(), метод 389
 get_x(), метод 390, 394
 get_y(), метод 390, 394
 get(), метод 155, 158, 295
 give_me_five(), функция 171

give(), метод 251
grid(), метод 285, 293

H

handle_caught(), метод 339
handle_collide(), метод 335
Hand, класс
 как сочетание объектов Card 252
height, свойство 313, 390
Hello World, программа 30
human_move(), функция 184, 188

I

IDLE 35

интерактивный режим
 print(), функция 36
написание программ 35
подсветка синтаксиса 38
создание ошибок 37
терминология 37
сценарный режим
 запуск программ 39
 написание программ 38
 сохранение программ 38

images, атрибут 393

image, свойство 320, 390

import, команда 265

IndexError, тип исключения 210

index(), списочный метод 144

init(), функция 395

input(), функция 280

insert(), метод 144, 296

instruction(), функция 168

interval, свойство 390

IOError, тип исключения 210

is_busted(), метод объекта BJ_Hand 271

is_collideable, свойство 320, 390

is_hitting(), метод 272

is_pressed(), метод 346, 394

is_visible, свойство 333, 394

items(), словарный метод 159

J

Java 32

K

Keyboard, класс 389, 394
keyboard, объект 312
KeyError, тип исключения 210
keys(), словарный метод 158

L

left, свойство 320, 390
legal_moves(), функция 184, 186
lifetime, атрибут сообщения 325
livewires, пакет
 color, модуль 399
 games, модуль 388
 общая характеристика 308, 388
load_image(), функция 314
load_sound(), функция 395
load(), метод 395
load, функция консервации 206
lose(), метод 272
lower(), метод 62, 63

M

mainloop(), метод 283, 306
 «Новое графическое окно»,
 программа 313
math, модуль 364
MAX_VELOCITY, константа класса 384
Message, класс 312
 добавление на экран 325
 общая характеристика 388, 392
 создание 324
Missile, класс
 die(), метод 371
 __init__(), метод 367
 update(), метод 368, 371
 изменение 372
 общая характеристика 367

mood, свойство 241
 moon_weight, функция 70
 Mouse, класс 389, 393
 mouse, объект 312
 Music, класс 389, 394

N

NameError, тип исключения 210
 new_board(), функция 184, 186
 next_block(), функция 215, 217
 next_line(), функция 215
 next_turn(), функция 184, 191
 not, логический оператор 96
 n_repeats, атрибут 352, 393

O

odds_change, атрибут объекта 340
 open_file(), функция 214
 open(), функция 197
 or, логический оператор 97
 overlapping_sprites, свойство 320, 390
 overlaps(), метод 391

P

Pan, класс
 check_catch(), метод 338
 __init__(), метод 337
 update(), метод 337
 загрузка изображения сковороды 337
 pickle.dump(), функция 205
 pickle.load(), функция 205
 pickle, модуль 204
 pieces(), функция 184, 186
 Pizza, класс 338
 end_game(), метод 339
 handle_caught(), метод 339
 __init__(), метод 338
 update(), метод 339
 player.is_hitting(), метод 274

Player, класс 264
 populate(), метод 256
 pop(), списочный метод 144
 position, свойство 390, 394
 private_method(), метод 235
 public_method(), метод 235
 pygame, пакет 310
 Python 31
 достоинства
 бесплатность и открытость 33
 возможность интеграции с другими языками 32
 легкость в использовании 31
 сообщество 33
 установка
 в Windows 33
 в других операционных системах 34
 Python Tutor, рассылка 33

Q

quit(), метод объекта screen 313

R

randint(), функция 74, 101
 random.choice(), функция 130, 162
 random.randrange(), функция 114
 random.shuffle(), метод 257
 random, модуль 102
 randint(), функция 74
 randrange(), функция 75
 импорт 74, 114, 359
 общая характеристика 74
 randrange(), функция 75, 265
 range(), функция 108
 rb 205
 rb+ 205
 read_global(), функция 180
 read_it.txt, файл-аргумент 197
 readlines(), метод файлового объекта 203

readline(), метод 199
 readline([]), метод файлового объекта 203
 read(), метод файлового объекта 198
 read([число]), метод файлового объекта 203
 remove(sprite), метод объекта screen 389
 remove(), списочный метод 143, 144
 replace(), метод 63
 reveal(), метод 295
 reverse(), списочный метод 144
 right, свойство 320, 390
 rowspan, параметр 294
 row, параметр 293

S

scale_image(), функция 395
 Screen, класс 388, 389
 mainloop(), метод 389
 screen, объект 312
 self.update_text(), метод 299
 self, параметр 223, 227
 set_angle(), метод 390
 set_background(), метод объекта screen 389
 set_bottom(), метод 391
 set_color, метод 392
 set_dx(), метод 391
 set_dy(), метод 391
 set_event_grab(), метод объекта screen 389
 set_image(), метод 390
 set_interval(), метод 391
 set_is_collideable(), метод 391
 set_is_visible(), метод 394
 set_left(), метод 391
 set_position(), метод 391, 394
 set_right(), метод 391
 set_size(), метод 392
 set_top(), метод 391
 set_value, метод 392
 set_velocity(), метод 391
 set_x(), метод 390, 394

set_y(), метод 391, 394
 shadow_global(), функция 181
 shelve.open(), функция 206
 shelve, модуль 204
 Ship, класс
 die(), метод 385
 update(), метод 364, 370, 372, 384
 изменение 376
 метод-конструктор 370, 384
 общая характеристика 362, 370
 переменная и константа 364, 384
 sin(), функция 365
 size, свойство 392
 sort(), списочный метод 143, 144
 Sprite, класс 312, 320, 388, 390
 status(), метод 231
 sticky, параметр 294
 still_playing, свойство 273
 stop(), метод 355, 395
 StringVar, класс 302
 strip(), метод 63
 str(), функция 68
 sun_weight, функция 70
 super(), функция 337
 swapcase(), метод 63
 sync(), метод 207
 SyntaxError, тип исключения 210
 sys.exit(), функция 214

T

talk(), метод 223, 224, 234, 242
 tell_story(), метод 305
 text_file, переменная 197
 Text, класс 312, 391
 Text, объект
 добавление на экран 322
 создание 322
 tick(), метод 391
 time_til_drop, атрибут объекта 340
 title(), метод 63, 283

`tkinter`, модуль 279, 282, 288, 303
`top`, свойство 320, 390
`total`, атрибут 231
`True`, значение 83, 85, 88
`TypeError`, тип исключения 210

U

UML (Унифицированный язык моделирования) 248
`update_count()`, метод 291
`update_text()`, метод 299
`update()`, метод
 спрайты 320, 391
`upper()`, метод 62, 63

V

ValueError, тип исключения 210
`values()`, словарный метод 159
`value`, свойство 322
`variable`, параметр 302
VELOCITY_MAX, константа 384
`velocity`, свойство 390

W

WAV, формат 353
`wb` 205
`wb+` 205
`welcome()`, функция 216
`width`, свойство 313, 389
`winner()`, функция 184, 187
Wrapper, класс 374
 `die()`, метод 375
 `update()`, метод 374
`writelines()`,
 метод файлового объекта 202, 203
`write()`, метод файлового
 объекта 201, 203

Z

ZeroDivisionError, тип исключения 210

A

Абстракция 169, 172
Алгоритмы
 определение 98
 пошаговая доработка 99
 создание на псевдокоде 99
Анимация
 на основе статических изображений 350
 общая характеристика 349
 создание объекта *Animation* 352
 создание списка изображений 351
Аргумент исключения 212
Аргументы
 определение 37
Архивные файлы 387
Атрибуты
 вывод на экран 228
 доступ 228
 закрытые
 доступ 234
 общая характеристика 233
 создание 234
 инициализация 227
класса
 доступ 231
 общая характеристика 229
 присвоение новых значений 231
 создание 231
 общая характеристика 226
ООП 221
 открытые 233
 управление доступом
 доступ к свойствам 239
 создание свойств 238
Аудио
 звуки
 воспроизведение 354
 загрузка 353
 остановка 355
 системный динамик 50
 циклизация 355

музыка 355
воспроизведение 356
загрузка 355
музыкальная тема 380
остановка 356
циклизация 356
общая характеристика 352

Б

Базовое окно
вход в событийный цикл 283, 285, 287
изменение вида 283
импорт модуля tkinter 282
общая характеристика 281

Базовый класс
вызов методов 260
наследование 255
переопределение методов 259
создание 258

Бесконечный цикл
намеренное создание 91
конструкции с break 93
конструкции с continue 93
общая характеристика 86
трассировка программ 88
условия, которые могут становиться ложными 88

Блок
определение 78
создание начального 101
тело цикла 86, 93

В

Ввод с помощью мыши
настройка видимости указателя 332
общая характеристика 330
перенаправление ввода в графическое окно 332
чтение координат указателя 331

Векторы 126
Ветвление 75

Видимость указателя мыши 332
Вложенные вызовы функций 67
Вложенные последовательности
доступ к вложенным элементам 147
обработка ошибочного выбора 149
общая характеристика 145
ожидание пользователя 149
распаковка 147
рекорды 148
создание 146
Возврат значений 37
Возвращаемые значения, способ обмена информацией 171
Вызов функции, определение 37
Вызываемая переменная 70
Выражения, определение 37, 55

Г

Генерация псевдослучайных чисел 72
Глобальная область видимости 178
Глобальные константы 182
Глобальные переменные
затенение локальными функциями 180
изменение локальными функциями 181
когда пользоваться 182
общая характеристика 177
определение 178
чтение локальными функциями 180

Границы
общая характеристика 327
переопределение метода update() 329
создание подклассов Sprite 328
Графический пользовательский интерфейс (GUI) 277
Графическое окно
запуск основного цикла 312
импорт модуля games 311
инициализация 312
общая характеристика 310
перехват ввода 332

Д

Декораторы 232

Деление 55

Деление нацело 55

Десятичные дроби 55

Документирующая строка,
определение 168

Доступ на запись для закрытых
атрибутов 239

Доступ на чтение для закрытых
атрибутов 239

З

Закрытие файлов 198

Запись в текстовые файлы

запись списка строк в файл 202

запись строк в файл 201

общая характеристика 200

Звук 352, 353, 355, 357

Значения

в словаре 152

вывод нескольких 46

параметров по умолчанию 176

установка начальных 101

Значения параметров

по умолчанию 174, 176

И

Изменяемость, определение 115

Именованные аргументы 175

Индексация, определение 111

Инкапсуляция

и абстракция 172

объектов 232

определение 172

Инструкция 167, 168, 169

Итерации цикла, определение 106

К

Кавычки

вставка 50

вывод нескольких значений 46

общая характеристика 44

применение внутри строк 45

строки в тройных кавычках 47

указание заключительных символов

строки 47

Кадры

анимация 350

Клавиатурные константы 347

Классы

импорт 265

наследование 253

объявление 222

определение 221

Клиент функции 233

Ключи

общая характеристика 152

применение для доступа к значениям
словаря 154

проверка с помощью оператора in 154

Кнопки

вход в событийный цикл
базового окна 287

общая характеристика 285

создание 286

Код, определение 37

Количество кадров в секунду (fps) 312

Комментарии 41

Консервация данных 204

Консольное окно 30

Константы

модуля color 399

модуля games 396

определение 117

создание 159

Конструктор надкласса 378

Конструкторы

общая характеристика 224

создание 225

создание нескольких объектов 225

Конструкции if

общая характеристика 77

- операторы сравнения 77
- построение 79
- создание блоков с помощью отступов 78
- создание условий 77
- Конструкция с break**
 - в каких случаях пользоваться 93
 - выход из цикла 93
- Конструкция с continue**
 - в каких случаях пользоваться 93
 - выход из цикла 93
- Конструкция с from** 311
- Конструкция с try** 208, 212
- Кортежи**
 - вложенные 149
 - вывод на экран 125
 - индексация 128
 - как условия 125
 - когда лучше пользоваться списками 144
 - неизменяемость 129
 - общая характеристика 126
 - определение 104
 - перебор элементов 126
 - применение оператора in 128
 - создание 123
 - пустых 124
 - с элементами 125
 - резы 128
 - сплление 129

Л

- Логические операторы**
 - and 96
 - not 96
 - or 97
 - общая характеристика 93
- Логические ошибки, борьба с ними 65
- Логические переменные 298
- Локальные переменные, определение 178

М

- Манипуляции 62
- Массивы 126

- Математические операторы 55
- Менеджер размещения Grid**
 - grid(), метод 285
 - общая характеристика 291
 - размещение элемента управления 293
 - создание элемента управления Entry 294
 - создание элемента управления Text 295
 - текстовые элементы управления 295

Метки

- вход в событийный цикл базового окна 285
- общая характеристика 284
- создание 285
- создание рамок 284

Методы

- вызов 62, 224
- закрытые
 - доступ 235
 - общая характеристика 233
 - создание 235
- объявление 223
- определение 221
- переопределение унаследованных 258
- экземпляра 223

Модули

- написание 263
- общая характеристика 262
- определение 74
- применение импортированных функций и классов 265

Н

- Надклассы** 261
- Нажатия клавиш, регистрация 346
- Наследование**
 - изменение поведения унаследованных методов
 - вызов методов базового класса 260
 - использование подклассов 261
 - общая характеристика 258
 - переопределение методов базового класса 259
 - создание базового класса 258

множественное 252
 расширение классов 253
 создание классов 252
Неизменяемость, определение 115

О

Области видимости 178
Обновление значения переменной цикла 86
Обработка исключений
 добавление условия `else` 212
 обработка разных типов 211
 общая характеристика 208
 прием аргумента 212
 применение конструкций `try/except` 208
 указание типа 209

Обработчики событий
 общая характеристика 290
 определение 280
 связывание 291
 создание 291

Объектно-ориентированное программирование (ООП)
 изменение поведения унаследованных методов 258

общая характеристика 32
 полиморфизм 262
 сообщения 246
 сочетание объектов 249

Ограничители 50

Оператор `in`
 использование с кортежами 128
 использование со списками 137
 общая характеристика 111
 проверка на существование ключей 154

Операторы и функции для работы с последовательностями

общая характеристика 109
 оператор `in` 111
 функция `len()` 110

Операторы сравнения 77

Отрицательные номера позиций 113
Отступы, создание блоков кода 78
Ошибки 37
Ошибкачный выбор, обработка 144, 149

П

Пакетные файлы 282
Параметры
 определение 170
 прием значений 171
Пары «ключ – значение»
 добавление 156
 замена 157
 удаление 157
Перегрузка операторов 65
Переключатели
 доступ к значениям 303
 общая характеристика 300
 создание 301
Переменная цикла 200
Переменные
 именование 57
 инициализация 162
 использование 57
 общая характеристика 56
 определение 56
 создание 57
Переменные-счетчики 108
Перенос текста 214, 215, 295
Перехват исключений 210
Переход по табуляции вперед 49
Печать на экране
 имен 69
 кортежей 125
 нескольких значений 46
 обратного слеша 49
 объектов 228
 указание заключительных символов
 строки 47
Пиксели 312
Повторное использование кода 173

Повторяющийся код 365
 Подсветка синтаксиса 38
 Позиции среза 121
 Позиционные аргументы 175
 Позиционные параметры 174
 Полиморфизм 262, 274
 Положительные номера позиций 112
 Пользовательский ввод 58
 Последовательность, определение 105
 Последовательный доступ,
 определение 111
 Пошаговая доработка, определение 99
 Присваивающие конструкции,
 определение 57
 Программы на основе Tkinter 281
 Прозрачность, загрузка изображений
 с этим свойством 318
 Производные классы
 определение 255
 применение 256, 261
 расширение 255
 Псевдокод
 написание для игрового цикла 268
 определение 98
 создание алгоритмов с его помощью 99

P

Рамки, создание 284
 Расконсервация данных 206
 Распределенные ссылки 149
 Режимы доступа к бинарным файлам 205
 Родительский элемент, определение 285

C

Сайты
 официальный сайт Python 34
 сайт-помощник 34, 387
 Самодокументирующий код 58
 Свойства
 доступ 239
 создание 238

Связывание событий
 с обработчиками 280
 Символы
 чтение из строки 199
 чтение из файла 198
 Система координат графики 315
 Системы управления базами данных
 (СУБД) 96
 Словари
 доступ к значениям 153
 общая характеристика 152
 пары «ключ – значение» 156
 поиск значения 156
 создание 152
 требования 158
 Событийно-ориентированное
 программирование 280
 Событийный цикл
 базовое окно 281
 определение 280
 Создание экземпляра класса 221, 223, 362
 Сообщения
 общая характеристика 246
 отображение 323
 отправка 248
 прием 249
 Составные операторы присвоения 68
 Списки
 изменяемость 138
 изображений 351
 индексация 138
 общая характеристика 135
 определение 108
 применение оператора in 137
 применение функции len() 137
 создание 137
 срезы 138
 сцепление 138
 функций 183
 элементы 139
 Списочные методы
 выход из программы 142

обработка ошибочного выбора 144
 общая характеристика 140
 ожидание пользователя 144
 отображение меню 141
 рекорды 142

Спрайты

вращение 347
 движение 325
 добавление на экран 319
 загрузка изображений 318
 определение 316
 отображение 315
 производные классы 328
 создание 318

Срезы

определение 119
 создание 122
 списков 139

Статические методы

вызов 232
 общая характеристика 229
 создание 232

Столкновения

обнаружение 335
 обработка 335
 общая характеристика 333

Строки

вывод на экран 42
 запись в файлы 201
 неизменяемость 115
 повторение 53
 преобразование в целые числа 67
 пустые 41
 создание строковыми методами 62
 скеление 52

Строки в тройных кавычках 47, 169

Строки, определение 37

Строковые методы

общая характеристика 61
 создание новых строк 62

T

Текстовые файлы 194
 Текст, отображение
 добавление объекта Text на экран 322
 импорт модуля color 321
 общая характеристика 320
 создание объекта Text 322
 Терминал Python 36
 Точечная нотация 74, 265
 Трассировка программы 88

У

Управляющая переменная цикла
 изменение значения 86
 инициализация 85
 проверка значения 86
 Условия
 интерпретация любого значения как
 истинного или ложного 91
 которые могут быть ложными 88
 общая характеристика 89
 определение 77
 простые 93
 создание 77
 составные 94
 Установочный файл Python
 для Windows 34

Ф

Файлы
 запись
 общая характеристика 200
 списков строк в файлы 202
 строк в файлы 201
 открытие 197
 хранение структурированных данных
 консервация данных 204
 общая характеристика 203
 полки 206
 расконсервация данных 205

чтение
 общая характеристика 194
 открытие и закрытие 197
 посимвольное 198
 циклический перебор строк 200
 чтение всех строк в список 200

Флажки
 общая характеристика 296
 проверка статуса 299
 создание 298

Фоновые изображения
 загрузка 313
 назначение 315
 общая характеристика 313

Функции
 документирование 168
 импортированные 265
 модуля games 395
 нестандартные 169
 общая характеристика 165
 объявление 168
 абстракция 169
 определение 37
 принимаемые и возвращаемые
 значения 169
 создание списка 183
Функций консервации 206

X

Хранение, определение 150

Ц

Целые числа
 определение 55
 преобразование строк в них 67

Циклизация
 звука 355
 музыки 356
 перебор строк файла 200
 перебор элементов кортежа 126

Цикл отгадывания, создание 102

Циклы for
 в других языках 106
 общая характеристика 106
 перебор строк файла 200
 создание 107
 счет
 общая характеристика 107
 по возрастанию 108
 по убыванию 109
 с интервалом 109

Циклы while
 общая характеристика 84
 управляющая переменная цикла
 инициализация 85
 обновление значения 86
 проверка 86

Ч

Числа
 математические операторы 55
 общая характеристика 53
 случайные
 randint(), функция 74
 randrange(), функция 75
 импорт модуля random 74
 общая характеристика 72
 типы данных 55

Чтение с клавиатуры
 общая характеристика 345
 регистрация нажатий клавиш 346

Чувствительность к регистру 36

Э

Экземпляры 221

Элементы
 вложенные 147
 кортежа 125
 создание непустых кортежей 125

списка
 присвоение новых значений
 по индексу 139

- удаление 140
строки 126
- Элементы управления
Button (кнопка) 280
Checkbutton (флажок) 280
Entry (текстовое поле) 280
Frame (рамка) 279
Label (метка) 279
Radiobutton (переключатель) 280
Text (текстовая область) 280
извлечение и вставка текста 295
- и обработчики событий
общая характеристика 290
связывание 291
создание 291
объявление создающего метода 289
определение 284
позиционирование с помощью менеджера
размещения Grid 293

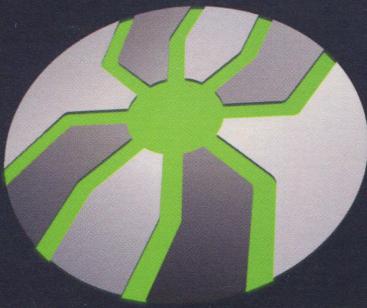
Я

Языки высокого уровня 32

М. Доусон
Программируем на Python

Перевел с английского В. Порицкий

Заведующий редакцией	<i>Д. Виницкий</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>В. Керимов</i>
Литературный редактор	<i>Н. Гринчик</i>
Корректор	<i>Е. Павлович</i>
Верстка	<i>А. Барцевич</i>



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

Эта книга — идеальное пособие для начинающих изучать Python. Руководство, написанное опытным разработчиком и преподавателем, научит фундаментальным принципам программирования на примере создания простых игр. Вы приобретете необходимые навыки для разработки приложений на Python и узнаете, как их применять в реальной практике.

Для лучшего усвоения материала в книге приведено множество примеров программного кода. В конце каждой главы вы найдете проект полноценной игры, иллюстрирующий ключевые идеи изложенной темы, а также краткое резюме пройденного материала и задачи для самопроверки. Прочитав эту книгу, вы всесторонне ознакомитесь с языком Python, усвойте базовые принципы программирования и будете готовы перенести их на почву других языков, за изучение которых возьметесь.

Научитесь программировать на Python играючи!

С помощью этой книги вы освоите принципы объектно-ориентированного программирования и сможете

- Формировать и индексировать строки, выделять срезы.
- Писать функции.
- Выполнять чтение из текстовых файлов и запись в них.
- Создавать спрайты и управлять ими.
- Создавать графические интерфейсы.
- Работать со звуком и музыкой, создавать анимацию.

 COURSE TECHNOLOGY
CENGAGE Learning®

Тема: Программирование. Языки и среды разработки/Python

Уровень пользователя: начинающий

 ПИТЕР®

Заказ книг:

Санкт-Петербург

тел.: (812) 703-73-74, postbook@piter.com

www.piter.com — вся информация о книгах и веб-магазин